A Multi-View Software Infrastructure for Hybrid Immersive Environments

Alessandro Febretti (afebre2@uic.edu)
Ph.D. Advisor: Andrew Johnson
Committee Members: Michael Papka, Robert V. Kenyon, Leilah Lyons, Luc Renambot

Executive Summary

In the domain of large-scale visualization instruments, Hybrid Reality Environments (HREs) are a recent innovation that combines the best-in-class capabilities of immersive environments, with the best-in-class capabilities of ultra-high-resolution display walls. HREs create a seamless 2D/3D environment that supports both information-rich analysis as well as virtual-reality simulation exploration at a resolution matching human visual acuity. Co-located research groups in HREs tend to work on a variety of tasks during a research session (sometimes in parallel), and these tasks require 2D data views, 3D views, linking between them and the ability to bring in (or hide) data quickly as needed. Addressing these needs requires a matching software infrastructure that fully leverages the technological affordances offered by these new instruments.

In this proposal I detail the requirements of such infrastructure and outline the model of an operating system for Hybrid Reality Environments: I present an implementation of core components of this model, called Omegalib. Omegalib is a framework and runtime that facilitates application development on HREs. Omegalib is designed to support dynamic reconfigurability of the display environment: areas of the display can be interactively allocated to 2D or 3D workspaces as needed. Compared to existing frameworks and toolkits, Omegalib makes it possible to have multiple immersive applications running on a cluster-controlled display system, have different input sources dynamically routed to applications, and have rendering results optionally redirected to a distributed compositing manager. I present examples of applications developed with Omegalib for the CAVE2™ HRE, show that dynamic reconfigurability has little impact on application performance and discuss how a Hybrid Reality Environment proved effective in supporting work for a co-located environmental sciences research group.

Based on the aforementioned results, I propose to further extend the existing implementation, to fully realize the vision for an operating system for Hybrid Reality Environments. The extension will integrate Omegalib with the Scalable Adaptive Graphics Environment (SAGE), a widely used window management system for cluster-based display walls, and will add support for user-centered immersive viewports that can be moved and resized at runtime. The proposed extension is twofold. First, a distributed rendering resource allocation technique that allows immersive windows to be resizable up to the full HRE display size, without sacrificing resolution or application performance. Second, an improved multiuser interaction system, offering dynamic device/window association based on heterogeneous pointing techniques (2D, 3D ray, 3D frustum).

Intellectual Merit

The work outlined in this proposal will extend the state of the art in large display software infrastructures, will solve several limitations in current systems (lack of immersion support in multi-window environments, single-application-only immersive environments) and will provide an advanced platform for application development on Hybrid Reality Environments.

Broader Impact

An operating system for Hybrid Reality Environments will accelerate research using these novel instruments in two ways. It will simplify and speed up the creation of domain-specific applications that make full use of HREs. And it will support co-located user groups tackling modern research, analysis or planning tasks requiring multiple heterogeneous data displays involving immersive and non-immersive views.

Table of Contents

1 Introduction

Today, most research involves the analysis of scientific phenomena of ever-increasing scale and complexity, and requires the concentrated effort of interdisciplinary teams: scientists from different backgrounds whose work involves large, heterogeneous data sources. As the scale and complexity of data continue to grow, large scale visualization instruments like display walls and immersive environments become increasingly essential to researchers, letting them transform raw data into discovery. In particular, immersive systems are an attractive option for exploring 3D spatial data such as molecules, astrophysical phenomena, and geoscience datasets [1]. On the other hand, display walls with their high resolution help interpret large datasets, offering both overview and resolution, or can be used to lay out a variety of correlated visual data and documents for collaborative analysis [2], [3].

Current technology trends lead to the production of larger, affordable thin-bezel LCD monitors with built-in support for stereoscopic 3D [4]. Such recent advancements made it conceivable to merge the best-in-class capabilities of immersive Virtual Reality systems, such as CAVEs, with the best-in-class capabilities of Ultra-high-resolution Display Environments, such as OptIPortals [5] thus creating conceptually new immersive environments which we refer to as Hybrid Reality Environments (HREs), such as the CAVE2 system [6], [7]

Naturally, hardware is only half of the equation. To fully take advantage of these novel technological affordances we need a matching software infrastructure that facilitates scientific work within HREs. This infrastructure should be tailored to the real-world needs of research teams, taking into account the way users interact with the instrument, with their applications and with co-located or remote members of their team.

So far, software development for display walls and immersive systems has followed independent paths. On display walls, the focus has been on supporting scalable ultra-high resolution visualization, enabling and managing multiple views of heterogeneous datasets and co-located collaboration. On immersive environments, the effort is to provide low latency viewer centered stereo, naturalistic interaction and remote collaboration.

What is now needed is a convergence in software design for display walls and immersive environments: I propose the integration of display wall software, an immersive environment framework and additional components into an "operating system" for Hybrid Reality Environments (Figure 1) to promote the creation of hybrid reality visualizations that make the full use of the capabilities of HREs. This operating system aims to solve two major challenges with alternative approaches:

1) *Static allocation of 3D and 2D views:* although an HRE is capable of displaying 2D and 3D content at the same time, software relies on static configurations describing how the physical display space should be split into 2D and 3D views. Multiple predefined configurations give the end users some flexibility, but require restarting and resetting all running applications.
2) *Lack of unified interaction*: the 2D and 3D portions on an HRE often rely on inconsistent interaction schemes: for instance Pointing semantics may use absolute movement on the 3D half and relative movement on the 2D half. Physically separate interaction devices may be required. Or, when a single device is used, interaction may lead to conflicting results (i.e. navigating a 3D view moves 2D views).

The HRE operating system also needs to satisfy the requirements of two distinct, but often overlapping, categories of users: scientific application users (i.e. research teams) and scientific application developers.

Application users need access to a system that lets them easily and seamlessly manage display space and content. We can for instance consider a structural engineering task: Engineers want to compare different designs of a building, load a digital model, and explore it in full-scale in the Hybrid Reality environment. They then decide to compare two variants of the design side-by-side, splitting the available screen space in two. Different users from the team explore the structures separately, while discussing their design. The group then chooses to look at pictures of the target site: they hide one of the 3D visualizations, and use the now available screen space to share photographs. The team splits again, with one group discussing the site while another user navigates the building model. The user notices a flaw in the design and interrupts the rest of the team. He uses a virtual slicing plane to generate a section of the building that gets displayed on a separate 2D view. The team now brings up the alternate design again, observes the two 3D

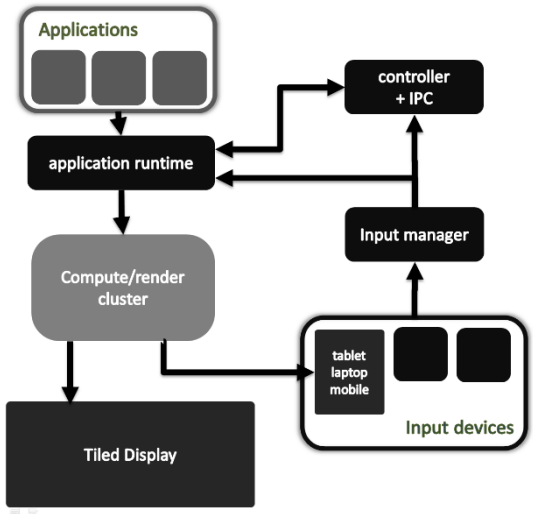

Figure 1. The high level model for the proposed multi-view operating system for Hybrid Reality Environments. The core components of the operating system are: the distributed application runtime; a controller and inter-process communication manager that handles application lifetime and communication between application instances; a distributed input manager capable of handling heterogeneous devices.

Figure 2. An overview of the CAVE2 Hybrid Reality Environment. CAVE2 is based on a cylindrical setup composed by 18 columns of 4 displays each. This arrangement provides a panoramic view of 320 degrees. Each display pair is driven by a separate computer, for a total of a 36 computer cluster, plus a head node. CAVE2 uses an optical motion tracking system arranged in a circular configuration above the displays

visualizations and a set of 2D sections, until they agree on one of the design variants. Before closing the meeting, they display this variant on the full display again, and mark a few points for future revision.

This kind of work pattern can be observed in research groups in a variety of disciplines: co-located collaboration often entails multiple phases that alternate full group work with individuals or sub-groups working in parallel: moreover the work may focus on a single issue, on different views into that single issue, or on independent issues [8]–[10]: the ability to easily re-configure the display space is fundamental to support such heterogeneous tasks [11].

An HRE operating systems also needs to provide an easy, yet powerful application programming interface (API) that developers can leverage to implement custom software. Immersive applications are often created to serve interdisciplinary research teams whose members may have limited programming experience. Applications also have a great variance in their complexity: some only need basic visualization and interaction with 3D models. Others require the implementation of complex and custom visualization techniques, need low-level access to the graphics API or need to use specialized visualization libraries. A way to satisfy these requirements is to provide a layered API, with multiple levels of access that offer a tradeoff between complexity and control.

2 Background

The benefits of co-located collaborative work have been analyzed in a variety of contexts like design [11], software development [8], and engineering [9][12]. Advantages of co-location include reduced communication friction (users more than 30 meters away are effectively remote [13]) and distributed cognition: teammates exploit features of the social and physical world as resources for accomplishing a task [14].

Effective visualization in large scale environments is an open research area, and its requirements have been investigated in the past. In terms of data size and format, some scientists need to display data so massive that it exceeds the screen resolution of even a large display [15]. Others find a large display ideal for comparing different data sets of the same phenomenon. Lastly, large display surfaces can help aggregate heterogeneous data about a specific phenomenon [16]. This third scenario (heterogeneous multiple views) is the most common[10], [17], [18].

In [17], the authors note how large display surfaces let researchers organize large sets of information and provide a physical space to organize group work. Part of the work is spent with each user individually working on a separate part of the display. Another part is dedicated to collaborative data consolidation and analysis, so everybody in the group can aggregate and discuss partial findings. During this kind of work, researchers often need to bring in new data and documents from their laptops or from the web: thus, a single pre-packaged application running on a large display rarely covers the full needs of a research group. Important factors for an efficient co-located collaborative space are, among others, the physical design of the space (space should support both the work of the entire team, and separate sub-teams), and its reconfigurability (both physical and at the software level).

2.1 Hybrid Visualization

The effectiveness of presenting data in different modalities has been the subject of previous research. 2D views have been found to be better when used to establish precise relationships between data, and for visual search ([19], [20])**,** while 3D is very effective for approximate 3D manipulation and navigation, especially with the use of appropriate cues, like shadows. In [21] it is suggested that combining both views leads to good or better analysis and navigation performance than using 2D or 3D alone. These findings are confirmed in [21], where in an air traffic control simulation 2D displays proved to be better for checking aircraft speed and altitudes while 3D was best used to perform collision avoidance

Bowman et al. have conducted extensive research on the advantages, challenges and best practices on hybrid 2D/3D visualization, proposing a taxonomy for Information-Rich Virtual Environments (IRVEs). In [22] the authors categorize visualizations depending on the physical and logical positioning of 2D and 3D displays, and in [23] they present a hybrid molecular visualization application for the CAVE system [24] that statically assigns 2D and 3D content to CAVE walls.

3 Prior Work

Existing software frameworks for large displays and immersive systems can be classified in a variety of ways. In [25], Chung et al. point out how a single taxonomy is insufficient to fully describe frameworks for cluster-based large functionalities. They propose a multi-faceted taxonomy that organized frameworks based on three dimensions: task distribution, input handling and programming model. In this review of prior work, I use a similar taxonomy, enriched with two additional dimensions: multiview support and immersion. The added dimensions allow to better characterize different approaches of software frameworks on display walls and immersive environments (Table 1).

The following review of current software frameworks organizes the existing approaches into several classes, based on a few defining characteristics shared by multiple solutions. Each class can be described by one or more dimensions of the taxonomy presented in Table 2: for instance, all the frameworks classified as Transparent Primitive Distribution Frameworks have the same task distribution model (Distributed Render) and the same programming model (No API). In general, frameworks in the same class can be seen as different solutions to the same problem (porting legacy applications to a large display, improving cluster render performance, supporting immersion, etc.). Although the classification presented in this proposal is not the only possible one, it helps in clustering similar approaches together and providing a better description of the state of the art.

3.1 Transparent Primitive Distribution Frameworks

Transparent Primitive Distribution Frameworks are designed mainly to port legacy applications to large displays, have a fully transparent API (matching the underlying graphics system API, like OpenGL) and use graphic primitives as the unit of distribution.

3.1.1 *Chromium*

Based on the earlier WireGL library [26], Chromium is a widely used transparent OpenGL framework for large display applications [27]. Chromium works by intercepting and replacing OpenGL library calls, so OpenGL

Dimension	Taxonomy	Description
Task Distribution Model	**Distributed application**	An instance of the application runs on each cluster node. Head node takes care of synchronization and message passing. Reduces network usage, but duplicates part of the workload on all nodes.
	Distributed Renderer	Application runs on the head node. The head node distributes drawing primitives to the cluster nodes, which act as lightweight renderers. Requires more bandwidth but consumes less processing resources on the rendering nodes.
Input Handling	**No Input Handling**	Input handling is completely left to the application developer.
	2D Event Handling	The framework provides functionality for handling 2D-type input (mouse pointers, touch, etc.). The input is typically used to handle 2D content or interact with a classic 2D graphical user interface.
	3D Event Handling	The framework provides functionality for handling 3D-type input (tracked devices with six degrees of freedom, wands, 3D gesture interfaces etc.). The input is typically used to control manipulation and navigation in 3D environments, but can also be re-mapped to 2D input on virtual of physical surfaces.
	Hybrid/Extensible Event Handling	The framework supports both 2D and 3D input capability, usually through an abstracted input event syntax. The event handling machinery is extensible to new event types and to new logical or physical input devices.
Programming Model	**Transparent/No API**	The framework is designed as a transparent layer that handles application execution and / or rendering on the display cluster. It simplifies porting of legacy applications, usually at the cost of efficiency and / or flexibility.
	Low-level API	A minimal API is provided, usually consisting of a set of simple callback-style interfaces that developers use to implement application logic. And low-level primitive-based drawing. Makes it moderately easy to port legacy applications and gives developer some control over the environment, but provides no utility APIs (rendering, scene, input handing, etc.)
	Medium-level API	The framework provides one or more APIs that assist application development using higher-level concepts like a scene graph, user interface widgets, etc. Legacy applications cannot easily be ported.
	High-level API	In addition to the medium-level API, the framework provides access to higher level functionality like a scripting engine, visual scene editor, pre-made application modules and plugins, etc. Frameworks with high-level APIs may force a specific application model and sacrifice some flexibility in exchange for ease of creating new applications.
Multiview Support	**No Multiview**	Only one application at a time can run and control the entire display surface. Switching applications or display configurations typically requires restarting the framework runtime.
	Static Multiview	Multiple views (from the same or multiple applications) can share the display, but their viewports are fixed at configuration or startup-time. Viewports cannot be moved or resized.
	Dynamic Multiview	Multiple views can share the display and can be moved / resized freely like classic windows on a desktop.
Immersion Support	**No Immersion**	The framework is only capable of displaying 2D views, or non-immersive 3D views (similar to 3D graphics on a standard display)
	Basic Immersion	The framework supports the display of stereoscopic 3D views.
	Advanced Immersion	The framework can display user-centered stereoscopic 3D views and supports advanced immersive capabilities like head tracking, tracked navigation/manipulation, positional audio, etc.

Table 1. The multi-dimensional taxonomy used to describe the existing large display / immersive software frameworks.

applications can be executed through the Chromium runtime without modifications. When an OpenGL application runs on a cluster head node, Chromium intercepts OpenGL calls using Stream Processing Units (SPUs). SPUs are used to manipulate the stream of OpenGL primitives. On a tiled display configuration, the *tilesort* SPU, performs a sort-first partition of the geometry and passes the OpenGL command stream to the display nodes. Another Chromium component running on display nodes (crServer) receives the command stream and performs the actual rendering. A configuration server running on the head node, called the mothership, manages information about the configuration of the display and controls execution.

Chromium is a practical and simple tool for transparently running legacy OpenGL applications on a large display, but introduces significant overheads that make it unsuitable for dynamic, large data visualization. Intercepting rendering commands in the head node is an expensive operation, especially when OpenGL is used in immediate mode and primitives are forwarded on a vertex-by-vertex basis. Chromium's performance also heavily depends on network bandwidth availability due to the large amount of graphics commands and primitives that must be transmitted for every new frame. Additionally, since Chromium must intercept OpenGL function calls, it only work for applications using functions wrapped by the Chromium/OpenGL API. Since the OpenGL API is extensive and constantly evolving, Chromium has not maintained feature parity with OpenGL and lacks a number of modern features including OpenGL Shading Language (GLSL) and Vertex Buffer Object (VBO).

3.1.2 *ClusterGL*

The ClusterGL middleware [28] uses a transparent wrapping approach similar to Chromium's, but uses several optimizations to reduce network usage such as multicast, frame differencing and compression. Frame differencing and compression make it possible to use intra-frame coherency to reduce data transfers by creating delta representations of the changes between frames. The tradeoff of this approach is the increased load on the head node due to computation of deltas. Benchmarks by ClusterGL authors show that, for most applications, ClusterGL outperforms Chromium applications. The performance difference increased with more complex scene geometries and more display nodes.

3.2 Distributed Compositing Managers

Distributed Compositing Managers make it possible to use a cluster-based display as a large seamless "desktop environment", where multiple applications can run in separate windows whose layout is controlled by the user. These frameworks usually expose a minimal API and use pixel streams as the unit of distribution. They also offer some form of 2D input support, to make window management possible.

3.2.1 *SAGE*

SAGE is the de-facto standard middleware for driving multiview large scale displays. [29]. Prior to SAGE, software to drive tiled displays operated like DOS in the 1980s, running only a single application at a time, and occupying the entire wall. SAGE turns a tiled display wall into a multitasking environment that enables multiple users to maximally take advantage of the vast resolution and screen size and operate multiple applications simultaneously. It permits them to juxtapose multiple visualization and data products side by side for analysis. SAGE has three components: a window manager called the Free Space Manager (FSManager), SAGE Receivers that drive the individual tiles in the display, and an API called the SAGE Application Interface Library (SAIL). Applications use SAIL to generate an additional frame buffer. One drawing for a frame is done, the application is in charge of copying the frame data to the SAIL frame buffer. Pixels in a SAIL frame buffer are divided in blocks and streamed to the SAGE receivers on the nodes corresponding to the output viewport of the application on the tiled display. A single application could also be running on a distributed system, with separate instances generating separate parts of the frame buffer. The Free Space Manager takes care of routing and organizing the pixel streams accordingly, to generate a final consistent image on the display nodes. The SAGE distribution includes several standard applications like picture and video viewers, remove desktop streaming tools and mouse pointer sharing applications. SAGE support 2D input from multiple devices and users. For instance, it is possible to have two users controlling window placement on a display simultaneously, with one user interacting with windows directly through a touch overlay and another controlling them using the mouse pointer on his or her laptop. While direct use of SAIL is not fully transparent, there is a collection of tools for SAGE, which makes the environment easier to use. For instance, it supports an OpenGL wrapper that allows existing OpenGL applications to be run on the display with minimal modification.

3.2.2 *DisplayCluster*

DisplayCluster is an interactive visualization environment for cluster-driven tiled displays [16]. It provides a desktop-like windowing system similar to SAGE. In addition to managing window content based on pixel streams, DisplayCluster provides a built-in media viewer optimized for the display of ultra-high resolution images with dynamic zooming. Similar to SAGE, DisplayCluster provides a python-based scripting interface that can be used to program and control interaction.

3.3 Parallel Rendering Frameworks

Parallel Rendering Frameworks can be used to create applications that make efficient use of a cluster rendering and computation resources. They tend to expose a low-level generic API that gives users a lot of control and flexibility on implementation. They may use a Distributed Application distribution model but they are typically flexible enough to let the developer control the details of distribution and parallelization. Parallel Rendering Frameworks can be used to create 2D or 3D applications.

3.3.1 *Equalizer*

Equalizer is a framework for scalable parallel OpenGL rendering [30]. It provides an API to create OpenGL multi-pipe applications for cluster-based large displays and immersive systems. Equalizer provides a runtime environment that applications can use to distribute their execution. The runtime consists of a messaging layer, a basic display management layer capable of creating graphic viewports and managing contexts, and a multi-threaded set of callbacks that developers use to implement application logic and rendering across cluster nodes. In contrast to Chromium which runs a full application including rendering tasks at the head node, the rendering part of an Equalizer application is executed solely in the display nodes and all rendering tasks are executed locally to the OpenGL context on each display node, rather than being driven by the head node. Therefore, Equalizer application can reduce computation steps and workload in the head node and instead of transmitting low-level graphics commands, the head node can send higher level of graphics calls to reduce network traffic.

A dedicated configuration server manages the utilization and load balancing of the display cluster resources. An extended configuration syntax makes it possible to define pixel-generating channels (the application rendering nodes) and output channels (the display nodes). Channels can be linked by compound trees that specify how frames should be generated, and can be used to define various task decomposition techniques for parallel rendering (sort-first, sort-last, tile load balancing, pixel compounds). To implement scalable applications with Equalizer's API, developers define callback functions like GLUT and implement sub-classes that present abstraction of physical and logical entities for rendering, such as display node, GPU, window and view. In addition to the parallel programming interfaces, the framework provides users with useful libraries including a network library and a library for multi-threaded programming. Equalizer also supports up-to-date OpenGL advanced features such as VBO, GLSL, and CUDA.

3.3.2 *CGLX*

The Cross-Platform Cluster Graphics Library (CGLX) provides a minimally invasive OpenGL programming model aimed at supporting high-performance rendering on a display cluster [31]. Nodes in the display cluster maintain each an independent OpenGL context, and use a custom communication layer provided by CGLX to exchange synchronization messages. CGLX exposes most of the standard OpenGL and GLUT APIs, and works by intercepting and re-implementing some draw callbacks and view-related OpenGL functions.

Nodes in a display cluster using CGLX can be connected to multiple displays. CGLX uses multiple thread to make efficient use of the CPU and GPU resources of each node. Since the framework allows users to configure how a separate thread performs rendering on each display/window in parallel or in a serial way, users can optimize the visualization performance based on different display configurations. A tool included in the CGLX distribution provides a graphical interface to simplify display configuration (a process typically done through manual editing of configuration files). With this tool, developers can configure the display system by adjusting various display parameters including size of bezels, resolution of displays, and arrangement of the display array. The tool also allows to preview an application switching between different configurations. The framework can handle input event streams

from multiple input servers and handles a variety of 2D input devices An extension of the framework called CGLXTouch adds support for multitouch displays such as tabletops and hand-held displays [32].

3.3.3 *MPIglut*

Similar to CGLX, MPIglut supports running applications on cluster based large displays providing an OpenGL/GLUT-compatible API. Unlike CGLX, which uses its own communication layer, MPIglut uses the standard Message Passing Interface (MPI) library for network communication [33]. While fully transparent approaches like Chromium work by intercepting function calls at runtime, with MPIglut users need to recompile OpenGL code and explicitly link it against the MPIglut library.

3.3.4 *FlowVR Render*

FlowVR [34] is a parallel rendering framework that supports a sort-first parallel rendering algorithm for cluster-based display systems. Instead of distributing raw OpenGL commands streams over a network, FlorVR distributes resources in a higher level format based on assets (vertex buffers, textures, shaders, shader properties) and simple draw commands using those assets. This distribution model reduces bandwidth requirements, simplifies the sort-first step (which works on vertex buffer bounds instead of filtering single OpenGL commands) and exploits the GPU's shader capabilities. A FlowVR application is made of two separate components: a Viewer and a Renderer. The Viewer component run the application logic, creates assets and issues draw commands to be distributed to Renderers. The Renderer component receives assets and commands from Viewers and is in charge of rendering content destined to a specific tile. Depending on different types of large display applications, the programmer can choose among different distributed rendering strategies according to how the Renderer and Viewer modules are distributed among cluster nodes.

3.4 Distributed Scene Graphs

Distributed Scene Graphs are a "specialized" subset of Parallel Rendering Frameworks that specifically target 3D application development. Distributed Scene Graph frameworks expose a medium or high-level API and may include support for immersion or 3D input.

3.4.1 *OpenSG*

The OpenSG framework implements a scene graph interface that explicitly supports multithreading and multi-node distribution [35], [36]. Each display node maintains a copy of the scene graph in a serializable data format based on *fields* and *field containers.* Any change in the scene graph (node transformations, visual property changes, node additions / deletions) can be represented as one or more field changes in a set of containers. A thread in each display node concurrently handles and synchronizes the scene graph according to the change list of the fields from the head node. User applications can also synchronize custom data by implementing additional classes that use the field container specification. Due to the overhead in synchronization and update of the distributed scene graph, very dynamic visualizations may not run efficiently with OpenSG. However, large scene graph changes can be compressed to reduce network bandwidth. OpenSG also supports parallel rendering algorithms such as sort-first and sort-last. Multicast transmits the change lists and rendering data across the cluster. OpenSG is built on top of OpenGL and its API consists of the original sets of libraries but some of the OpenGL and GLUT functions can be used within an OpenSG application, when finer control over drawing operations is needed.

3.4.2 *Garuda*

Garuda is built on top of the OpenSceneGraph (OSG) toolkit [37], a widespread scene graph rendering and utility library [38]. Similar to what Chromium does for OpenGL, Garuda enables legacy OSG applications to run on cluster-based display without the need to recompile or alter their code. Garuda takes care of detecting and synchronizing changes in the scene graph using multicast communication to the display nodes. The framework replaces the standard OpenSceneGraph cull draw and swap operations with specialized versions that optimize tile-based frustum culling [39].

3.5 Wall Application Frameworks

Wall Application Frameworks are designed to simplify the creation of 2D applications for large, cluster-driven displays. They may offer some multiview capability like Distributed Compositing Managers, but sacrifice some of their flexibility to provide a higher-level (and possibly domain-specific) API.

3.5.1 *jBricks*

jBricks is a Java framework for 2D drawing on large scale cluster driven display walls [40]. jBricks supports interaction with heterogeneous input devices and integrates the 2D graphics rendering API of ZVTM [41], a powerful toolkit for developing information visualization applications. jBricks requires small changes to a single-display ZVTM program, mostly related to initialization and callback management. Several standard 2D graphic primitives supported by Java (bitmaps, text, widgets) are available in jBricks, embedded in objects called *glyphs*. Glyphs are distributed through JGroups multicast communication, and visualized on viewports that span one or more tiles in a display wall. Viewports can also contain other viewports to increase the flexibility of visualization layouts. Each display node performs 2D culling on the viewport area within its tile or tiles. jBricks also provides a specialized input server called jBIS, which supports a variety of 2D input devices such as tablets and game controllers, and can connect to other input data sources using the TUIO protocol [42].

3.5.2 *Shared Substance*

Shared Substance is a middleware for developing flexible interactive multi-surface applications [43]. One of the objectives of shared substance is to make it possible to create dynamically reconfigurable environments where software and interaction is distributed across heterogeneous devices (display walls, smartphones, multitouch tables), and where hardware and software features can be added and removed at runtime. Shared substance utilizes a tree representation of applications, display and input resources. Nodes represent data, while the connections between nodes represent message exchange paths between different components in the environment. Nodes support the connection to *facets*, which represent the programmable entry-points of the application setup. Facets can be used to run scripts, launch applications, setup data sharing points and so on. Adding, removing and modifying nodes or facets in the three reconfigures the application or its execution environment at runtime. Multiple shared substance processes running on different machines, can be linked together through a discovery process and can share their node/facet trees through a process akin to mounting a drive or filesystem in Linux. Mounted trees can be replicated locally, or can exist as proxies that exchange messages with the remote tree location.

The authors of shared substance show how the proposed model can be used to implement both a distributed scene graph and synchronize multiple instances of a non-distributed application. The Shared Substance middleware is created in python. It should be underscored that although Shared Substance uses a high level language and programming concepts, it does not expose a high-level API in the sense discussed in Table 1. Shared substance provides an environment to launch applications, coordinate their execution and share data/input, but developers are in charge of choosing the appropriate tools or libraries to develop the applications themselves.

3.5.3 *MediaCommons*

MediaCommons is a middleware designed to simplify the creation of complex visualization applications for tiled display walls [44]. MediaCommons makes it possible to display and arrange 2D content on a display wall, but does so without using the pixel streaming approach of SAGE. Rendering happens directly at the visualization nodes, through the use of replicated *renderable data objects* that synchronize their content and are also aware of the boundaries of the local display. MediaCommons' approach is similar to the one use in jBricks and partially DisplayCluster, but offers the additional capability of displaying 3D data, since the middleware is based on OpenSceneGraph. A limitation of this approach compared to distributed compositors like SAGE, is that rendering for all viewports happens within a single process: a resource-intensive view with a slow framerate would slow down the framerate of the entire wall display.

3.6 Immersive Application Frameworks

Immersive Application Frameworks are aimed at creating 3D applications for immersive environments. These frameworks support advanced immersion features (as discussed in table 2) and are designed to handle 3D input. Immersive Application Frameworks expose heterogeneous APIs. Some have low level APIs made of simple callbacks, designed to be as generic as possible. Others implement full scene graph APIs or ad-hoc visualization pipelines to allow for the rapid development of immersive applications in specific areas.

3.6.1 *CAVELib*

CAVELib was originally developed as a dedicated API for the original CAVE system [45], [46]. The initial implementation of CAVELib was targeted at shared-memory system, but it has been extended to support fully distributed display clusters. At startup, an application is replicated and runs a separate instance on each node. As in other distributed application frameworks, application logic and rendering tasks are split and run in separate threads. CAVELib supports immersive interactive 3D application features like head tracking and off-axis projection. More advanced 3D functionality like navigation and object manipulation is left for application developers to implement. Altough CAVELib only provides an OpenGL API for drawing, it supports integration with higher level libraries such as Performer and OpenSceneGraph.

3.6.2 *FreeVR*

FreeVR is an open-source virtual reality interface/integration library [47][48]. It has been designed to work with a wide variety of input and output hardware, with many device interfaces already implemented. One of the design goals was for FreeVR applications to be easily run in existing virtual reality facilities, as well as newly established VR systems. The other major design goal is to make it easier for VR applications to be shared among active VR research sites using different hardware from each other. FreeVR is designed to be API-compatible with CAVELib. FreeVR is not a VR content library. It does not provide a scene graph layer, or other features often associated with such libraries like intersection testing and collision detection. It does not provide a physical simulation for objects in the virtual world. FreeVR is intended to work with many scene graph and other libraries. While SGI's Performer library was the primary alternative for many years, it is no longer a primary focus (though if you have a machine with Performer, it will still work). More recently, the OpenSceneGraph (OSG) library has been used with several FreeVR applications, including some that make use of the Delta3D system [49].

3.6.3 *VR Juggler*

The VR Juggler virtual reality toolkit [50] is based on a virtual platform framework. The virtual platform is constructed of two main components: the draw manager and the kernel. Every application acts as an application object in the form of a C++ object and the manager and kernel provide the application with interfaces to the graphics API and hard- ware devices respectively. The kernel executes application objects and controls the run-time system by brokering communications among the managers. The manager provides abstraction of multiple input devices, displays, networking, and windowing systems. Programmers access new devices by simply creating a new manager in the API. Developers can easily extend the system during run time, and can make use of various existing graphics APIs, including OpenGL, OpenSG, OSG, VTK [51], etc. Developers can write applications with a set of predefined kernel interface functions and existing graphics APIs. VR Juggler achieves a consistent and stable frame rate since it is not affected by certain performance factors incurred by other frameworks, such as network traffic and dividing rendering tasks in the single head node. An additional feature of VR Juggler is a GUI-based performance analysis tool, VjControl, for debugging VR Juggler codes. This tool gathers and provides important performance data such as rendering times, wait time for buffer swaps, etc.

3.6.4 *CalVR*

CalVR is a virtual reality middleware system based on OpenSceneGraph [52]. CalVR implements similar core functionality present in other immersive environment frameworks like CAVELib FreeVR and VR Juggler and adds a menu system, multiple user interaction and remote collaboration support. Applications are programmed using the OpenSceneGraph C++ API, plus several additional classes exposed by CalVR itself that handle 3D object manipulation, input and menu support. CalVR applications are implemented as plugins that can be loaded and

Name	Distribution Model	Input Handling	API	Multiview Support	Immersion Support
Chromium, ClusterGL	Render	No Input Handling	No API	No Multiview	No Immersion
SAGE	Render	2D	Low Level	Dynamic	Basic Immersion
DisplayCluster	Render	2D	Low Level	Dynamic	No Immersion
Equalizer	Both	2D	Low Level	Static	Advanced Immersion
CGLX	Application	No Input Handling	Low Level	Static	No Immersion
MPIglut	Application	No Input Handling	Low Level	No Multiview	No Immersion
FlowVR Render	Render	No Input Handling	Low Level	No Multiview	No Immersion
OpenSG, Garuda	Render	2D	Low + Medium Level	No Multiview	No Immersion
jBricks	Application	2D	Medium Level	No Multiview	No Immersion
Shared Substance	Application	Hybrid/Extensible	Low Level	Dynamic	No Immersion
MediaCommons	Application	2D	Low + Medium Level	Dynamic	No Immersion
CAVELib, FreeVR, VR Juggler	Both(Default=Application)	3D	Low Level	No Multiview	Advanced Immersion
CalVR	Both(Default=Application)	Hybrid/Extensible	Low + Medium Level	No Multiview	Advanced Immersion
AVANGO	Application	3D	High Level	No Multiview	Advanced Immersion
HRE OS	Both (Default=Application)	Hybrid/Extensible	Low + Medium + High Level	Dynamic	Advanced Immersion

Table 2. A summary of the reviewed software frameworks and middlewares, categorized based on the multi-dimensional taxonomy presented in Table 1. Features marked in green represent desirable features in a Hybrid Reality Operating System, based on the requirements described in the introduction. Note: frameworks identified as having both a renderer and application distribution model typically implement a distributed application approach (an instance of the application runs on each cluster node), but their API is flexible enough to decouple application logic from rendering when desired. Frameworks with a multi-level API expose optional higher level interfaces that can be skipped when the programmer needs more control over the implementation.

unloaded at runtime. CalVR supports rendering stereo views for multiple users [53], but does not support fully separate views displaying independent immersive applications.

3.6.5 *AVANGO*

AVANGO is a distributed scene-graph framework [54] and flexible display configurations for different types of cluster-based large displays. AVANGO uses a data distribution architecture similar to OpenSG, but is based on OpenSceneGraph and the Python scripting language. Programmers can develop an entire application in Python without the need for other programming languages. AVANGO integrates modules that implement utility functions for VR applications like 3D menu support, display setup and input device configuration.

3.7 Summary

A summary of the reviewed software frameworks for large displays and immersive environments can be found in Table 2. The last row of the table enumerates the characteristics of the proposed HRE operating system, which can be detailed as follows:

1. **Distribution Model:** ideally, the HRE OS should be flexible enough to support both a renderer-based and application-based application model. This can be achieved through application level replicated execution, if the operating system API is powerful enough to support asymmetric code execution and message passing between display and application nodes. In other words, user code should be 'aware' of its execution environment and have access to a low level API that supports implementing a distributed rendered model, if so desired.
2. **Input Handling:** the HRE OS should handle both 2D and 3D input devices, events and interaction techniques. Depending on the execution environment, an application should be able to switch input devices without significant decrease in functionality or interaction efficiency. Since HRE systems exist

at the intersection between 2D display walls and 3D immersive environments, HRE applications could make use of both 2D devices (mouse pointers, touch) and 3D devices (wands, markerless tracking systems, etc.). Choosing 2D interaction over 3D or vice-versa should only depend on application requirements and not on inherent limitations of the operating system. Gives the variety of input devices that need to be supported, the input handing model should also be generic and extensible, allowing developers to integrate new devices as needed.

3. **API Level:** as discussed in the introduction, a multi-layer API is a way to support developers with a broad range of requirements for their applications. A high level API provides a simple and quick access path to the HRE system. Lower level APIs provide flexibility at the cost of increased complexity.
4. **Multiview Support:** in the introduction, through examples and references to previous work, I also argued about the importance of letting users layout information as they see fit. Supporting this capability in an HRE operating system depends on two requirements. First, an interaction model should be provided to let users re-arrange content. Second, the operating system should use the layout information to optimize the use of rendering and computational resources in the HRE cluster.
5. **Immersion Support:** together with multiview support, this is one of the core objectives of this proposal. The HRE operating system should support all the advances immersion features present in state-of-the-art immersive application frameworks such as multiple stereo rendering modes, user-centered stereo, head tracking, etc. Additionally, given the desire to support multiple HRE users simultaneously, the operating system should offer some form of multi-user immersion.

The summary table illustrates how no existing software framework supports all the requirements for the HRE operating system described above. In particular, frameworks that support some form of multiview do not support immersion, and vice versa.

The next section of this proposal will describe the work done so far towards an operating system for hybrid reality environments. I will present the core of the current implementation of the HRE operating system, and illustrate how the current implementation satisfies the aforementioned requirements, with the exclusion of full multiview support.

In section 5 I present some results from the use of the current implementation: I illustrate its use to develop an application used by a co-located research group. Moreover, I outline the results of a system performance study to validate the current implementation and demonstrate the feasibility of the fully dynamic immersive multiview approach I propose to implement.

Section 6 represents the core of my proposal: In it I describe how I plan to further advance the current implementation with a focus on dynamic multiview. I detail the challenges in supporting true multiview support for immersive applications running on a display cluster and how I plan to tackle them.

In section 7 and 8 I draft an evaluation plan for my proposed research and lay out a timeline for this work, including implementation, evaluation and planned publications.

4 Prior Research

4.1 OmegaDesk: a Small-Scale Hybrid Immersive Workstation

In 2010, The Electronic Visualization Lab designed and implemented the OmegaDesk [55]. The OmegaDesk prototype is a hybrid, 2D/3D work desk that enables users to work seamlessly with 2D content (such as text documents and web browsers), as well as 3D content (such as 3D geometry and volume visualizations), and integrates multi-touch-sensitive surfaces. OmegaDesk consists of two stereo displays, one positioned horizontally in a 45-degree angle and another positioned vertically in front of the user. The bottom display is enhanced with a touch sensing overlay; additionally, the position of the user head and hands is tracked using either a marker-based or markerless tracking system. This combination of heterogeneous display and interaction devices made it possible to create hybrid applications that used immersive and non-immersive device features, as shown in Figure 3.

Figure 3. Two examples of OmegaDesk applications. On the left, a flow visualization, showing a 3D map view on the top screen, and multiple configurable 2D plots on the bottom screen. On the right, an educational application displaying blood components.

4.2 Omegalib: a Middleware for Hybrid Immersive Platforms

As part of the OmegaDesk research team, I developed Omegalib, a middleware designed to support application development on that platform. As Figure 4 shows, Omegalib acts as an abstraction layer between the hybrid reality hardware and user applications. At the back end, Omegalib uses Equalizer to drive the display system. In section 2.1, we observed how Equalizer uses an advanced and verbose syntax to support its flexibility, making it complex and error prone to write or maintain configuration files. Omegalib addresses this issue by describing display system geometry, cluster configuration and input capabilities through a compact system description file. When applications launch, the Omegalib runtime system transparently generates the required extended configuration files and forwards them to Equalizer, which then proceeds to initialize and setup the display system.

At the front end, Omegalib integrates several APIs and visualization tools through the use of an abstract scene graph and pluggable render passes. The abstract scene graph makes it possible to decouple object interaction techniques from concrete object representations. A concrete object can for instance be an OpenSceneGraph node [37], a Visualization Toolkit (VTK) Actor [51], or some other entity provided by one of the Omegalib front-ends. Concrete objects can be attached to nodes of the scene graph, which then represent the interface between user code and the underlying objects. Users can therefore control object transformations, visibility, hierarchy, etc. through a unified API that is independent from the library used to draw the objects on screen. Each front-end provides a render pass that can be attached to the Omegalib rendering system, and that takes care of drawing concrete objects of a specific class. Users needing low level drawing control can implement their own render pass and obtain a GLUT-like callback interface that can be used to execute custom OpenGL code. Render passes can be prioritized and can perform 2D or 3D drawing. For instance, a high priority 2D rendering pass can be used to overlay 2D content on top of an interleaved-stereo 3D scene.

4.2.1 *Application Model*

The default execution and rendering mode for Omegalib is replicated execution: each node in a display cluster runs an instance of the target application, while a master instance synchronizes the updates and buffer swaps across the nodes. Omegalib also makes it possible to control execution on a node-by-node basis (the API offers functions to check on what node the application is executing), and it also supports the synchronization of custom data objects between master and slave instances. Omegalib extensions and applications are implemented as three component types that communicate with the Omegalib runtime and with each other, as represented in Figure 1: Event Services, Application Modules and Render Passes.

Event Services implement input event processing. A service may implement support for a specific category of physical input devices (like a keyboard or mouse controller), or it can aggregate and reprocess events generated by other services. Event services can also receive input data from a remote input server, making it possible to dedicate a separate machine as the hub for all input devices in an HRE installation. Event services run exclusively on the master

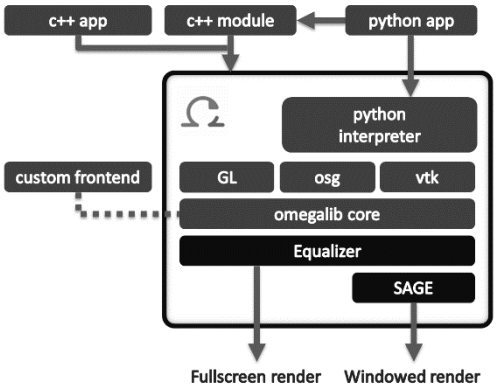

Figure 4. The Omegalib software stack

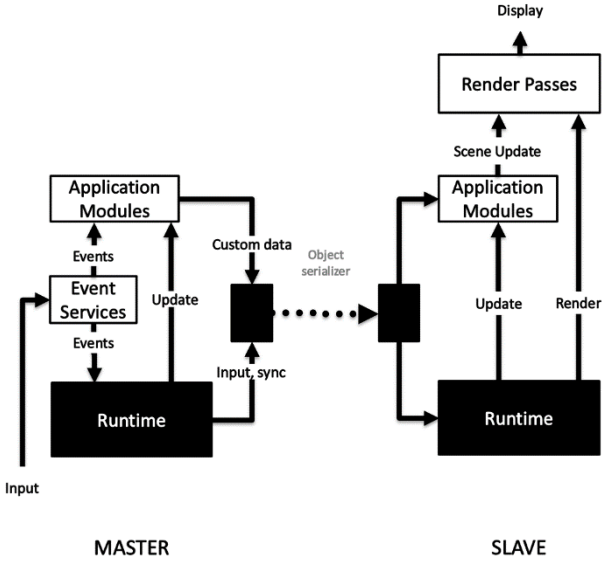

Figure 5. An overview of the communication flow of a distributed Omegalib application. In this example, the Master node runs in headless mode, and only takes care of application and input event processing.

instance of a distributed application. The Omegalib runtime takes care of serializing and distributing events to slave instances.

Application Modules contain the core of an Omegalib application logic. Modules can receive input events, update the abstract scene graph (or other custom scene representations), and communicate with other modules. Modules run on both master and slave instances of Omegalib regardless of whether they display content or not.

Render Passes have been presented in the previous section: they implement all the functionality needed to render scenes to cluster tiles or to secondary image streams. Application developers typically don't need to implement render passes, unless they need to perform custom low-level drawing operations. Render passes are implemented by integration libraries, to forward draw context and issue draw commands in the correct format used by the third party library. Render Passes are executed only on application nodes that need to perform rendering operations. For instance a headless master configuration will not run render passes on the master application instance.

4.2.2 *Dynamic Configuration*

As mentioned in the introduction, one of the objectives of Omegalib is the creation of user-defined and reconfigurable workspaces: areas of the display that are dedicated to a 2D or 3D application, that can be re-defined while the application is running and can be independently controlled.

To support this feature, we let users specify an optional 2D display region as a startup argument to an Omegalib application. The application runtime uses this information to find the subset of nodes in the rendering cluster that drive displays in the selected region: it then adjusts the system configuration to launch the application only on the identified target nodes. Multiple applications can be launched on the system in this fashion. The runtime also manages the networking setup, making sure that each application instance uses an independent set of ports to establish communication between the application master and the slaves.

To increase the flexibility of workspaces, we also let users expand or shrink the visible area of each workspace within the bounds specified at run-time. When a user shrinks a workspace, the runtime disables rendering on the nodes whose tiles are not covered by the workspace active area (Figure 6). While GPU resources are de-allocated, the

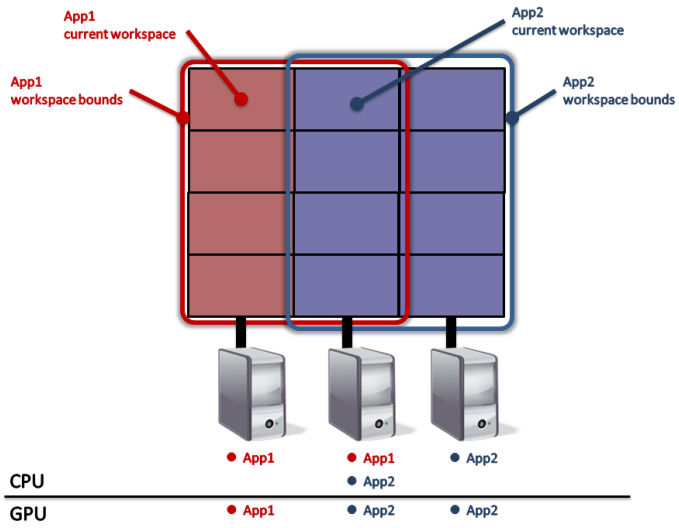

Figure 6. An example of dynamic workspace configuration for two running applications. Each machine controls one display column consisting of four tiles. Application 1 is launched on the left and center column. Application 2 is launched on the center and right columns. Applications workspaces overlap on the central column. In this example, the runtime configuration allocates the central column to Application 2. As shown in the bottom rows, both applications are running slave instances on the center column machine, but only the active application is assigned to the GPU.

application remains available on the machine: if the user later decides to expand the workspace again, rendering on the inactive tiles can be reset almost instantaneously.

When launching multiple applications, it is therefore possible to run applications on overlapping regions of the display system: the display space shared by multiple regions can be associated to any of the overlapping applications at runtime. This double level of dynamic configuration (launch-time and runtime) provides a good level of control over cluster resource usage versus workspace flexibility. On one end, applications can be launched on non-overlapping regions, optimizing cluster resource allocation (each node is dedicated to a single application) but giving up dynamic display allocation. On the other end, applications can be launched on fully overlapping regions covering the entire display space: in this case the workspaces have the full runtime flexibility, at the cost of sub-optimal cluster resource allocation (each node needs to keep one active instance of each application)

4.2.3 *Input Filtering*

When the Hybrid Display Environment is running multiple applications, it is necessary to let a user quickly interact with any of them without requiring switching to a different physical device or other complex interactions. It is also desirable to let different users control different workspaces and let them switch between them in a seamless way. In Omegalib, this feature is implemented through ray-based event filtering. We assume that the main interaction device in the environment offers 6-DOF tracking (as is the case for most large scale immersive environments). We can therefore use the tracking information to generate a 3D ray originating from the device, and compute whether the ray intersects one of the display tiles. Only the application that is controlling the tile (based on its runtime configuration) will process the event stream generated by the devices. This naturally scales to multiple devices controlling independent applications. It is also possible to choose another 6-DOF tracking source as the input to the event filter: for instance, a pair of tracked glasses can be used to filter input events based on gaze direction. Each application has control over its own event filtering policy. An application can also decide to disable filtering if it needs to let users interact with it regardless of its display state.

4.2.4 *Panoptic Stereo*

Since Omegalib is aimed at supporting co-located collaborative groups, it is fundamental to provide stereo-viewing capabilities to multiple users in the system. Because traditional CAVEs use viewer-centered perspective, one effect that typically occurs is that when the tracked user looks 180 degrees away from a given stereo screen, stereo reverses. While this is acceptable for a single user, given they are no longer looking at the screen, it poses a problem for other team members, especially in a larger cylindrical environment such as CAVE2. To solve this issue, Omegalib supports *panoptic stereo*. Techniques similar to Panoptic stereo have been implemented in the past to support multi-user stereo without generating multiple views[56]. When Panoptic stereo is enabled, the system tracks the main

viewer's position, but generates head orientation information based on each display plane normal. This way, the stereo frustum is projected outward to each display panel, preserving stereo separation for each user in the system. An additional benefit of Panoptic stereo is its independence from frame latency: When users move their head around, stereo will look correct regardless of the application frame rate, leading to a much more comfortable viewing experience. This comes at the price of a slightly incorrect overall projection, particularly when users rotate their head sideways. Panoptic stereo can be turned on and off at runtime.

4.2.5 *Scripting*

Although the main Omegalib API is in C++, most of it functionality is exposed to a scripting interface: the Omegalib runtime embeds a python interpreter that can be used to launch standalone script applications, or can be used to control running applications through a runtime console. Other than facilitating development access to non-technical users, scripting support has the added advantage of simplifying application portability. A script application can run on a personal computer or on a HRE without requiring recompilation or reconfiguration. This makes it possible for scientists to work on a visualization script on their own computer, save it to a flash drive, plug the drive into the HRE system and re-launch the visualization during a collaborative research session.

4.2.6 *Application Control and Message Passing*

We have so far discussed two of the software components of the HRE operating system model presented in Figure 1: the application runtime and input manager. The third and final component is the controller and Inter-process communication (IPC) manager. The primary purpose of the controller is to manage the execution of applications, providing users with an interface to start and stop application instances and manage their workspace areas.

The controller (whose Omegalib implementation is called MissionControl) runs as a server to which applications connect once started. Applications connected to a MissionControl server can exchange messages with each other (typically script commands), or receive messages from third party software through an external interface. MissionControl therefore allows multiple views in the HRE to communicate with each other, for instance in order to coordinate the information they display. Since views run as separate processes in the cluster, their frame rates are independent: this is a desirable feature when one of the views is computationally or graphically intensive, while others require real-time interactivity.

4.3 Porthole: view streaming and interaction with personal devices

Another approach to multiview support is based on streaming additional 2D or 3D views to a *secondary device* like a laptop, smartphone or tablet. These devices can act both as secondary display surfaces for visualization output and as input sources to the application. Exploratory work on this approach led to the development of Porthole: Porthole is a dynamic, HTML5-based decoupled interface to immersive environments. The goal of Porthole is to ease the interaction between VE systems and smartphones, tablets, laptops or desktop computers, without the need for ad-hoc client applications. Clients can connect to CAVE2 using an HTML5 enabled browser, receive an application-specific user interface, and interact with additional views of the application data generated by the immersive application and sent to the device as pixel streams.

Personal secondary views (PSVs) are appealing for a number of reasons. In a multi-user setting, PSVs create a private visualization space that each user can control to visualize information relevant to his own analysis task. PSVs can also act as a sort of augmented-reality tool: A smartphone-based PSV can be moved around by a user within a virtual environment and provide additional data about objects that fall within the view frustum [57]. From a technology standpoint, it makes sense to use laptops, smartphones or tablets as PSV channels. PSVs implemented on these devices can be used as a view and a source of input to the application. Touch screens offer a suitable surface for 2D interaction tasks like system control and brushing that are difficult to perform precisely in a VE [17]. Moreover, since PSVs are decoupled from the main visualization environment, they can be used to facilitate remote collaboration: users outside of the VE can connect to it, receive a view of the data and interact with it in real time as users present in the VE see the changes in the main and local secondary views. This is a particularly interesting aspect, since the size and cost of state-of-the-art virtual reality environments limits their access for everyday scientific workflows.

Figure 7. Two photographs taken during a co-located collaborative meeting in CAVE2. On the left, an Omegalib immersive visualization is running on the full display. On the right, CAVE2 is split into two workspaces to display additional 2D views. Switching between the two modes can be done at runtime, without resetting running applications.

5 Prior Results

To evaluate the effectiveness of Omegalib in supporting co-located collaborative work, we used it as the development platform for a geo-science application for the visualization and analysis of sonar data. This application was first used during a two-day meeting of the multidisciplinary team working on the NASA ENDURANCE project.

The Environmentally Non-Disturbing Under-ice Robotic Antarctic Explorer (ENDURANCE) is an autonomous underwater vehicle (AUV) designed to explore the extreme environment of the perennially ice-covered lakes of the McMurdo dry Valleys, Antarctica. ENDURANCE operated during two Antarctic summer seasons (2008 and 2009). The AUV operated depending on 3 distinct science objectives: Water chemistry profiling, Bathymetry scanning, and glacier exploration. For the purpose of bathymetry reconstruction, the source data consisted of about 200 million distinct sonar range returns, plus navigation data and AUV attitude information at 0.2 second intervals [58].

Over the course of the full two-day ENDURANCE meeting, the research group had to complete multiple tasks: discuss new vehicle designs for a future mission, analyze mission logs and cross-reference them to water chemistry readings, and generate a new 3D map of the lake based on the collected sonar data.

As shown in Figure 7**,** the team used the display in different configurations during the meeting. During the initial evaluation of sonar data, the entire display was dedicated to a 1-to-1 scale, immersive visualization of the sonar point cloud. This visualization allowed the team to identify issues in the data, compare depth measurements from different data sources and iterate through data collected for each mission. Later in the meeting, the 3D workspace was shrunk to make space for additional 2D views representing satellite imagery from Lake Bonney and different versions of the lake bathymetry represented as a contour map. One of the 2D views was controlled by an Omegalib script running a VTK pipeline and was linked to the 3D view. As researchers picked points in the immersive environment (effectively making 'virtual measurements' of the lake depth at points they deemed relevant), the 2D view would update, regenerating the contour information to take the new points into account. A researcher could use the hand-held interaction device (a tracked game controller) to navigate the 3D view, pick depth points or rearrange and resize the 2D views by simply pointing in the desired direction on the screen. Other users could control the view arrangement and content from their laptops. The workspace allocation could also be controlled by the hand-held device (using an on-screen menu) or from one of the laptops.

Another advantage for multiple workspace support is specifically targeted at application developers. As many other large scale display environments, HREs like CAVE2 are expensive and offer limited availability: they are often a highly contended resource, with multiple application developers scheduling access to the system to make sure their work does not conflict with others'. Emulators and smaller system replicas help, but are not a perfect substitute. For instance, estimating the performance of an application in an emulated environment is complex, due to the difference in hardware and display geometry between the two environments. Thanks to runtime workspace configuration, multiple developers can use an HRE system concurrently. We observed this usage pattern multiple times in the CAVE2 system. Developers join a work session and negotiate display allocation with others, so that each developer

has a section of the display exclusively available to him/her. Developers then use a command line switch to start their application on their workspace. Developers occasionally ask others to control the full display for a brief time, usually to test a specific feature.

5.1 Performance Evaluation

As mentioned in section 1.2, Omegalib supports static and dynamic workspace configurations. In fully dynamic setups, application viewports can be customized at runtime, to occupy any non-overlapping portion of the display. The tradeoff of dynamic workspaces is the need to replicate and synchronize multiple running applications across all nodes running a tiled display.

My working hypothesis is the following: for typical immersive application workloads, a dynamic workspace setup (i.e. replicating applications on all nodes) consumes additional system resources but has a small effect on application performance. I make this hypothesis based on the following assumptions: 1) like most modern graphic frameworks, Omegalib typically operates a few high priority non-graphics threads. High priority non-graphics threads include scene database update, logic/physics and network processing. 2) mid-tier machines used in current generation cluster displays have 8-32 CPU cores, enough to allocate multiple aforementioned workloads on disjoint cores. 3) Baseline network usage for replicated immersive applications mostly consists of input, synchronization and low frequency geometry transfers. Average per-frame usage is typically 1-10Mbps, orders of magnitude less than consumer-grade wired network connectivity available today.

Based on these assumptions, we identify the graphical processing unit (GPU) as the most likely bottleneck for Omegalib applications running on a modern cluster-based HRE. Omegalib allocates the GPU (and graphics-processing CPU threads) to a single application on each node, depending on the workspace associated to that node's tiles: I therefore expect applications to achieve similar frame rates, regardless of workspace configuration (static/dynamic).

To determine the performance effect of a dynamic workspace setup, I evaluated system and application behavior of an experimental setup running three workspaces with distinct applications, in static and dynamic workspace mode. The experiment was carried out on the computer cluster running the CAVE2 system. This cluster consists of 36 machines with 16 2.9GHz Xeon E5 cores, 64GB memory, 2TB local storage and 20Gb/s connectivity between nodes and storage server, and NVIDIA GTX 680 graphics with 2GB dedicated memory. Each machine (and graphics card) drives two display tiles with 1366x736 pixels of resolution each. The choice of a three workspace setup was motivated by empirical observations of the usage pattern of the CAVE2 cylindrical display area. In typical research sessions, observed the display being typically split into one, two (left, right) or three (left, center, right) work areas, each one dedicated to a specific task. In the experiment, I replicated the most demanding configuration, with three applications running on a 6x4 tile section each. Around 30 Omegalib applications and examples were available at the time of writing: I chose three that would stress different system components (Table 3).

The ENDURANCE application is a point cloud data visualization tool supporting the real-time display and query of large point cloud datasets (5-200M datapoints). The application makes heavy use of vertex, geometry and pixel shaders to efficiently filter and rasterize points as shaded spheres. The application frame rate depends on the number of point batches within the view frustum.

The Video Player application plays back pre-rendered stereo videos with arbitrary frame sizes (typically using the same resolution as the tiled display, ~75MPixels in CAVE2). Each player slave process loads a dedicated frame stream as jpeg images from networked or local storage, and uses several worker threads to decode images fast enough to support 60fps playback.

The Physics Demo application demonstrated Bullet physics integration with Omegalib and displays a spinning hollow cube containing several hundred box shaped rigid

Table 3. Summary of applications used in the performance evaluation experiment. The bottlenecks column indicates system components most likely to limit application frame rate. The GPU use column qualitatively indicates how much aggregate GPU resources (processing, memory, bandwidth) are used by the application. The threads column indicates the typical amount of CPU threads used by the application.

Name	Bottlenecks	GPU use	Threads
ENDURANCE	GPU	High	3-6
Video Player	CPU, NET	Low	3-32
Physics Demo	CPU, GPU	Medium	3-7

Fig. 8. An unwrapped view of the cylindrical CAVE2 display running the workspace setup used for evaluation. The three applications are, from left to right, ENDURANCE, Video Player, and Physics Demo.

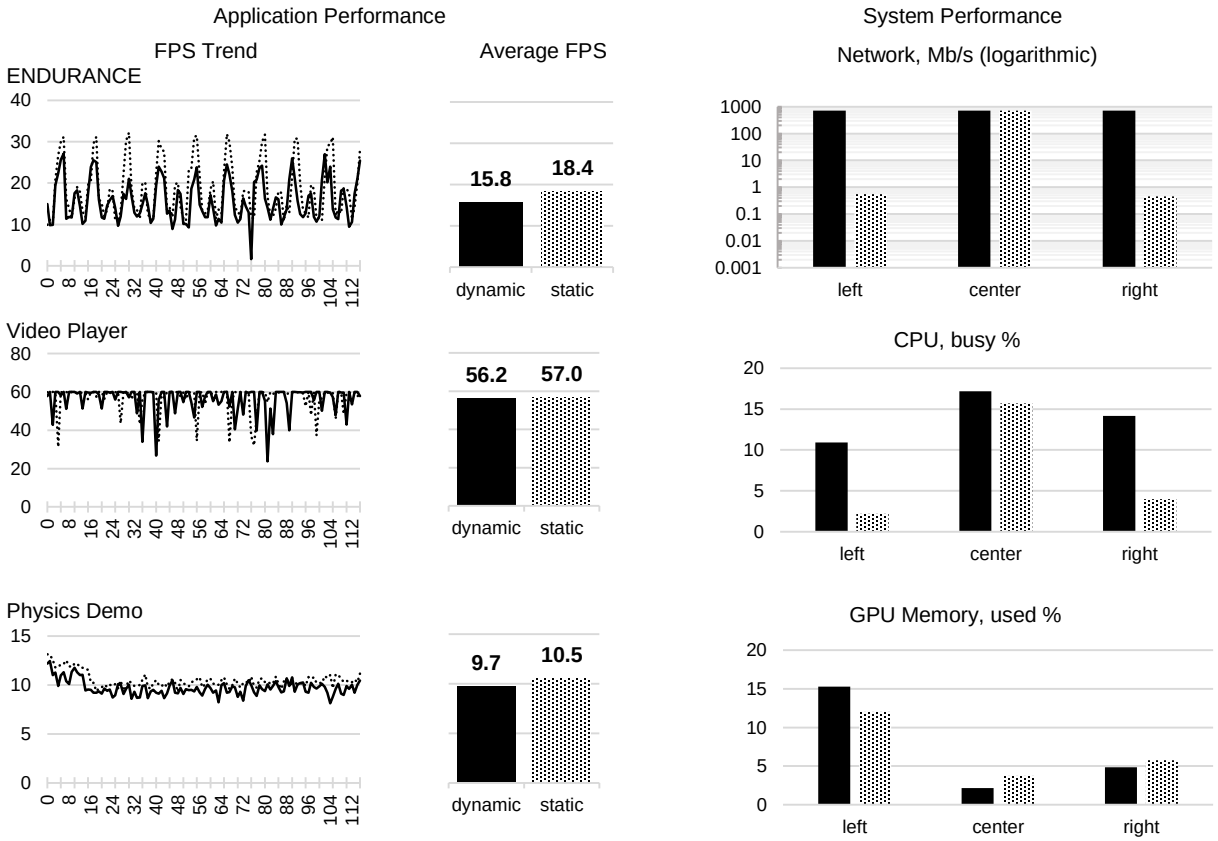

Figure 9. Application and system performance results from the multiview evaluation experiment. Charts on the left show frames-per-second (fps) trends over ~2 minutes, and fps averages for the three test applications, in the dynamic and static workspace setups. Charts on the right summarize cluster system performance for three components (Network, CPU, GPU) for distinct areas of the display (left, center, right) corresponding to the active workspace areas of the three test applications.

bodies (300 in our experiment). The spinning cube is used to ensure all rigid bodies remain active throughout the simulation, keeping a constant load on the physics engine.

Applications were scripted to execute a repeatable experiment in the dynamic and static workspace conditions: I used the MissionControl system presented in previous sections to inject script commands into the three applications at scheduled times, ensuring a consistent collection of application and system performance data.

For the static workspace condition, each application would run on a fixed portion of the CAVE2 display. In the dynamic workspace condition, applications were all started on the entire display, and after startup were configured to use the same display section as in the static workspace condition. Visually, both experiments rendered the same content

on the CAVE2 display, but in the dynamic workspace condition each display cluster node was running a replica of each application, two in update-only mode and one in update and display mode.

5.1.1 *Evaluation Results*

During experimental runs, application frame rates were logged at 1 second intervals, and system network, CPU, GPU and memory usage on all 36 CAVE2 nodes were recorded, again at 1 second intervals. System performance data was aggregated by display section, using the left-center-right tile allocation described previously.

Figure 9 summarizes the results of the aforementioned experimental runs. As predicted, the framerate trends for all three application are similar for static and dynamic workspace setups. Average frame rates are between 1 and 3 fps less for a fully dynamic workspace setup, an acceptable tradeoff for the flexibility of runtime workspace resizing and reallocation. System performance data also matches our expectations. For instance, network usage in the dynamic workspace condition is higher on all three display sections, since the biggest user of network resources (the Video Player application) is now running on the entire display. A similar trend can be observed for CPU usage. It is worth noticing that it would be trivial to optimize network usage, since each application instance is aware of its display status. A "smarter" video player implementation could stop frame buffering on inactive display nodes, bringing network usage to very similar levels to the static workspace condition. We did not introduce this optimization in our experiment, since we were interested in evaluating the behavior of applications unaware of dynamic workspace optimizations, to determine how much the Omegalib runtime could do without requiring application modification.

6 Proposed Research

In section 3.7 I observed how both classic immersive application frameworks and display wall software environments have desirable features for a hybrid environment software infrastructure. In particular, display wall environments like SAGE have the ability to freely layout (and resize) application viewports, support direct interaction from multiple users and already integrate with a variety of widely used general-purpose applications like picture viewers, document viewers, browsers, etc. Another major advantage is compositor/application frame rate decoupling: since applications run separately from the compositor software, they do not influence the compositor or each other's responsiveness.

Although Omegalib already supports pixel streaming to the SAGE environment, it does so with several limitations:

1. Streaming is only supported for 2D views. Streaming a viewer-centered stereo 3D view would require the application runtime to receive information about the physical position of the 2D viewport on the display surface, in order to recompute the off-axis projection frustum. Furthermore, if the viewport is resized without changing the resolution of the original viewport (the default mode of operation in SAGE), the runtime would also need to re-adjust the eye basis accordingly.
2. Even if multiple Omegalib applications can stream to SAGE simultaneously, a single node from each of them is in charge of rendering and streaming the view. This is the streaming mode most SAGE applications use since it is the simplest to implement and is sufficient for classic applications like document viewers. But for HRE applications, single node streaming is inefficient: the cluster rendering resources are left mostly unused, and as the view resolution scales up a single rendering node quickly becomes a bottleneck.
3. Omegalib applications running in SAGE are not able to filter interaction based on their screen position. In other words, a pointer movement or any other form of input acting upon an Omegalib SAGE window is not forwarded to the corresponding Omegalib application.

Addressing the aforementioned limitations would make it possible to use SAGE as an advanced and seamless front-end for Hybrid Reality Environments, allowing it to fully support multiple 3D immersive visualizations in addition to the 2D display and content management capabilities it already exposes. I therefore propose to extend the software framework developed so far, to fully realize the vision for an integrated operating system for Hybrid Reality

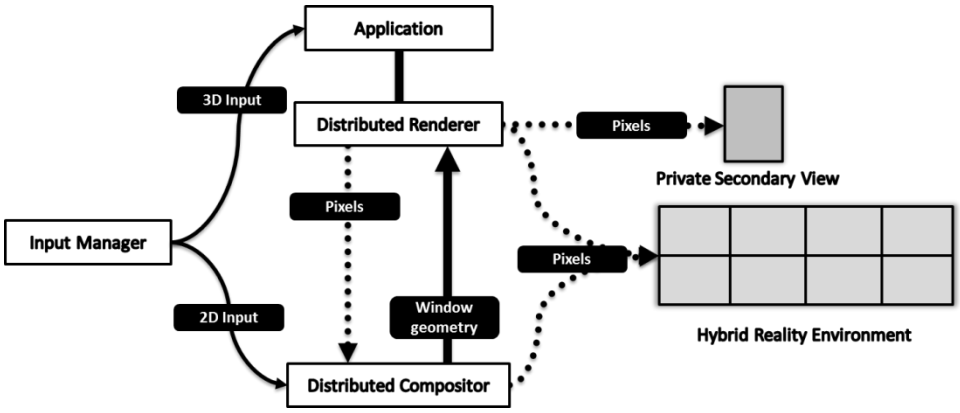

Figure 10. Distributed rendering and compositing model

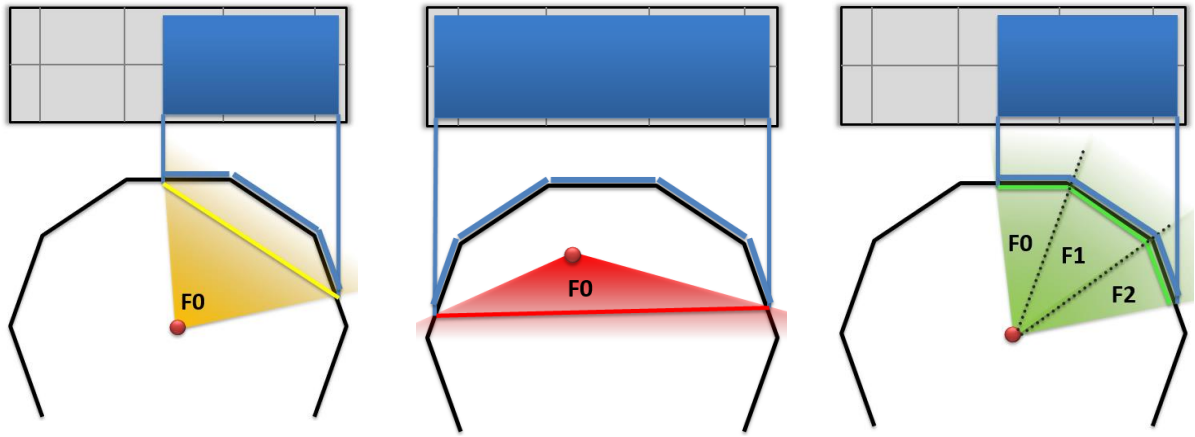

Figure 11. Examples of correct and incorrect frustum generation for movable stereo windows. On the left, a projection frustum generated by the window corners (yellow) does not match the actual display surface (blue segments). The resulting projection can go from slightly warped to completely incorrect as shown in the center example, where the combination of head position and window size leads to a frustum that is inverted w.r.t. the physical display surface. On the right, an example of correctly generated frustum set.

Environments as presented in section 1. The remainder of this section will illustrate the technical details of my proposal, followed by a possible evaluation experiment.

6.1 User-Centered Stereo Views in SAGE

The first objective of this research proposal is to add full support for user-centered stereo 3D windows within SAGE. As indicated in the previous section, this will require additional information to flow from the compositing manager to the Omegalib runtime, as shown in Figure 10. In detail, the compositing manager will communicate the position of a 3D window in normalized screen coordinates. Normalized screen coordinates are needed here since the compositing manager is not aware of the physical geometry and position of the display, as it is representing it as a simple 2D surface with undetermined world position. The application runtime has knowledge of the display geometry, and can convert the 2D window corners into real-world 3D points. These points can then be used in conjunction with the eye position to generate a new frustum for the off-axis stereo projection. Although this simple technique works well for planar displays (like classic display walls), it leads to incorrect projection results on arbitrary display surfaces, like the CAVE2 cylindrical display. Figure 11 (left and center) illustrates the issue: on non-planar displays, a single frustum generated using the window 3D corners will not match the actual display projection surface. To address this issue, we need to generate a set of per-tile frusta, each one corresponding to one of the physical display tiles, as shown in Figure 11 (right).

6.2 Distributed Rendering

As mentioned in the previous section, Omegalib already supports streaming to SAGE from a single node per-application, but this solution does not scale as the size of the output window increases. For simple applications like document or image viewers, a single rendering node is acceptable since the size of the output SAGE window does not affect the size of the source pixel stream. The frame buffer is simply scaled up or down to match the window size, and in general a screen pixel does not correspond 1-to-1 to a pixel in the source frame buffer. On the other hand, complex 2D and 3D visualizations need to make full use of the screen area available to them: as the size of their viewport increases, the size of the source frame buffer should increase as well, to constantly support a full resolution output to the hybrid environment display.

To satisfy this requirement, rendering distribution is fundamental: a single rendering node would quickly become a bottleneck as the number of rendered pixels increases. This is particularly important for user-centered stereo visualizations, as immersion depends on constant framerate and low latency between tracking and frame updates. Rendering distribution is also important to make a fair use of the resources available to computer clusters driving large-scale HREs like CAVE2. Without it, a few nodes would perform all the graphics heavy-lifting, while the rest of the system rendering resources would be left mostly unused, working only on compositing tasks.

Thanks to SAGE support for multi-stream windows and Omegalib distributed rendering and dynamic workspace capabilities, it is now possible to implement a rendering allocation algorithm that re-assigns rendering resources at runtime, depending on window positions and sizes. The proposed algorithm would work as follows:

1. New applications launch in dynamic workspace mode. A slave instance of the application runs on each cluster node. The active workspace is initially empty (no rendering is performed).
2. When the application is allocated to a SAGE window (or the window position/size changes):
    a. The master runtime uses the 2D window area to identify the set of machines whose tiles intersect the window.
    b. The set of machines become the new active workspace. Rendering is enabled on their tiles. The rendering target is set to an off-screen buffer.
    c. The slave runtimes connect to SAGE and setup rendering for a subset of the result window. Each machine determines the position and size of its assigned rendering rectangle based on the intersection between its tile and the output window.
    d. If the window partially intersects a tile, the off-screen buffer resolution on the corresponding machine is recalculated to match the intersection area.

This algorithm guarantees that rendering is always local to the machine displaying a specific window portion, and that render load (in pixels) assigned to a single machine never exceed the size of the physical tile (or tiles) connected to the machine. As the output window moves around, rendering resources are reallocated on the cluster to be as ‘local’ as possible to the window position. This reduces the need for pixel streaming across nodes and achieves a simple sort-last load balancing of rendering resources. As long as windows don’t overlap, each machine will remain in charge of rendering for a single application. More complex rendering allocation algorithms could be implemented on top of this one, for instance to take the global system load into account and redirect rendering to non-local nodes, at the price of increased network traffic due to pixel streaming.

6.3 Hybrid Interaction Support

Another important feature of the HRE operating system is the ability of multiple users to interact with the system, using a variety of input devices depending on task requirements. Furthermore, users should be able to seamlessly switch between application interaction (i.e. navigating within a view), window interaction (i.e. re-arranging or resizing windows) and system interaction (i.e. starting and stopping applications).

In a large-scale HRE system, a classic primary interaction device is a tracked wand or similar controller, offering 6-DOF tracking and several digital and analog inputs. Alternate input devices may be standard mouse pointers or secondary touch-based interfaces like tablets or smartphones.

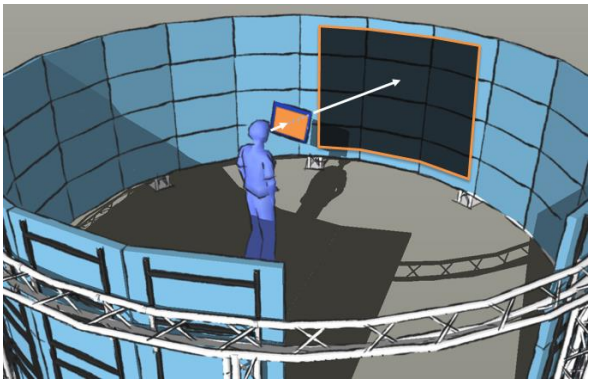

Figure 12. Tablet window association using head and tablet tracking.

Figure 13. Multiple interaction devices can be associated to application instances running in the Hybrid Reality Environment.

In the case of wands, it is desirable to let users perform 2D and 3D interaction using the same device, and have multiple users in the system control different windows. Section 4.2.3 illustrated the ray-based input filtering technique that has been implemented in the Omegalib runtime. This technique can be adapted to work in conjunction with SAGE and let the wand device seamlessly switch between interaction modes. For this technique to work, it is important to create an association between a wand and a specific window, in a sense letting a specific application 'acquire' input from a specific input device. Subsequent interaction does not depend on the user continuous pointing towards the application viewport, which may be intuitive or unpractical for tasks like navigation. This is also similar to the traditional 'active window' desktop metaphor, where the last window that has been clicked on by the user is the one with input focus, even after the mouse pointer leaves the window area. A key difference in the HRE context is that multiple interaction devices may be present, and each application will support association with any number of devices.

A similar interaction technique could be applied to a tracked handheld device, like a tablet or smartphone. I propose to use a tablet device as a secondary input/output tool. A tablet can be associated to a specific application using the a ray-based technique similar to the one used for wands: as the user holds the device in front of him or her, the runtime will identify which application can be seen through the tablet display, based on tablet tracking, tablet display size and head position/orientation. By touching a button or performing a gesture on the screen, the user could associate the tablet to the relevant application, and get a personalized application interface on the handheld display. The display could then be used as a see-through interface providing additional information on specific areas of the visualization, it could be used as an annotation tool, or it could provide fine-grained input controls to the application. Techniques similar to the aforementioned ones have been investigated in the past ([57], [59]–[62]), but never in the context of a multi-view immersive framework like the one discussed in this proposal.

7 Evaluation

Measuring the success of this proposed research involves both human and machine factors. On the machine side, it is important to assess the performance of the software infrastructure discussed so far. Section 6.2 underlined the importance of resource allocation across the cluster when multiple applications are running. In the evaluation of prior work I have already shown how a dynamic workspace setup performs similarly to statically allocated workspaces. Since the rendering allocation algorithm I propose to implement is based on the dynamic workspace technique, I expect application performance and system resource usage to be in line with the previous results. I plan to test the system with an extended number of applications to confirm this hypothesis and to better characterize possible stress points in the software infrastructure. Based on the results of this evaluation I may be able to optimize the runtime to further reduce resource usage, for instance by letting applications query the runtime for their state (active or inactive) on a node. Alternatively, I may provide optimization guidelines for typical scenarios, to drive multiview application developers.

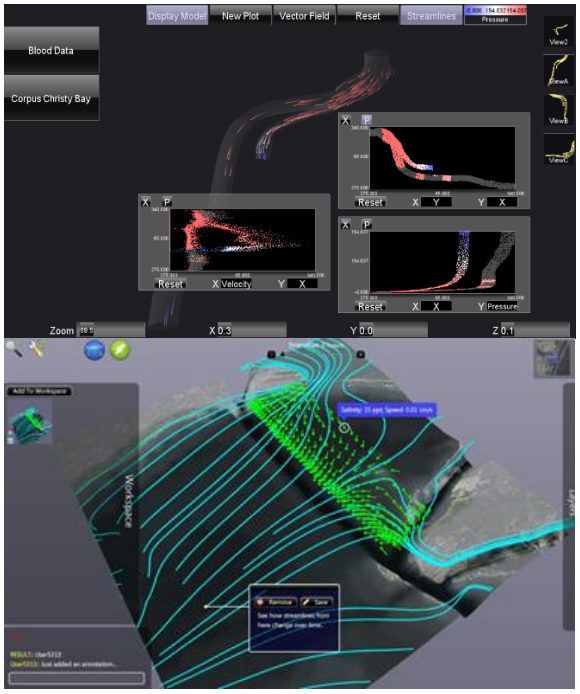

Figure 14. A screenshot of the FlowViz application displaying a portion of the corpus Christi bay dataset.

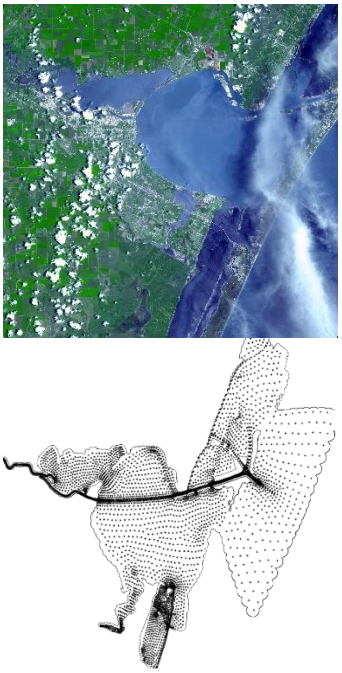

Figure 15. The web-based visualization tool created as part of my M.S. Thesis work, visualizing dynamic streamlines generated from the Corpus Christi Bay current data.

Figure 16. Top: a satellite image of Corpus Christy Bay, Texas. Bottom: A map of the 4216 data collection stations used to create the dataset.

Regarding the human side of the evaluation, I consider two alternatives. The first option is to evaluate the infrastructure in a real world collaboration setting, similar to the ENDURANCE meeting presented in section 5. Given the continuing collaboration between the team involved in ENDURANCE and EVL, it is likely that one or more similar research meetings will take place in the CAVE2 HRE in the near future. Sessions of those meetings could make use of the improved infrastructure described in this proposal. Depending on the nature of these meetings, acquiring consistent quantitative data may be difficult, and this alternative would be oriented more towards a qualitative observational study.

A second option would be to test the infrastructure in a controlled user study. Although I plan to finalize the details of this user study as a future step (see the timeline in the following section), I believe the following to be some of its requirements:

1. The study should simulate a co-located collaborative work session, since this is the principal usage mode of large scale HREs, and the proposed software infrastructure specifically supports it. Therefore, testing sessions should involve two-three users working together in a problem solving or visual analysis task.
2. The study should require the use of multiple, inter-related 2D and 3D views of data. The 3D data should be dense or spatially complex, to encourage the use of stereo cues and physical navigation around the data. The 2D data should be heterogeneous and include documents, images and websites if possible.
3. The layout of views, and the creation/destruction of views should be determined by users as part of the experiment.
4. Users should use wands, tablets, tracked glasses or remote pointers from their laptops as they see fit. The experiment should not force users to interact with the system and applications through a specific device.

Following these requirements would make it possible to create a controlled experiment that emulates with good accuracy typical usage patterns of a large scale Hybrid Reality Environment.

The test scenario could make use of real research data and visualizations (like the ENDURANCE data), assuming that users can be trained quickly enough to interpret it, and solve sole simple task based on it. Tasks could involve

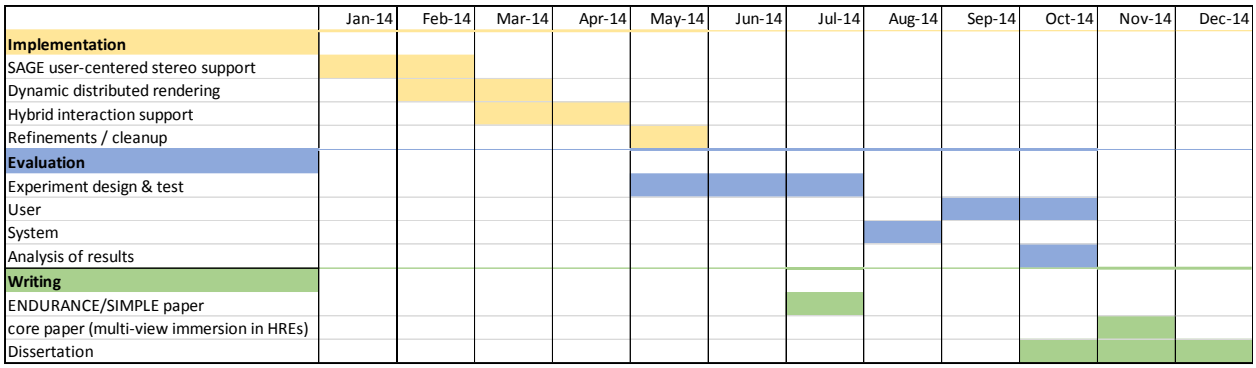

Figure 17. A draft timeline for this proposal.

identifying features in the data like geographical points, boundaries or outliers. Feature descriptions could be left intentionally vague to encourage users new to the data to consult predefined documents or websites for guidance. Other tasks could require manipulation of the data (marking bad data, adding annotations and/or waypoints, etc.)

Another dataset usable for the test scenario is the Corpus Christy Bay dataset I have previously used for my M.S. thesis work [63] (Figure 15) and for the OmegaDesk hybrid FlowViz application [64] (Figure 3, Figure 14). This dataset has been made available by the Texas Water Development Board, and provides time varying data about salinity, current speed and direction collected across the bay at various depths (Figure 16). The dataset contains about 3 million datapoints. Given the its 3D complexity and the presence of multiple time points, this dataset has good potential in demonstrating the value of using multiple 2D and 3D immersive views to make sense of the hydrological and chemical properties of the bay environment.

A control group condition for the study would be represented by a visualization application running on the HRE system without making use of the immersive multiview features of the HRE operating system. In other words, the study design may compare user performance while using a single, immersive visualization of the target dataset (simulating visualization in a classic immersive environment) versus the multiview hybrid visualization discussed in this proposal.

In the case of a strictly quantitative study, user performance could be measured as time-to-completion or error rate of tasks involving the identification of target features in the dataset (for instance, identifying the boundary of areas of high salinity and their movement over time, or marking areas with significant water turbulence and linking them to the local bathymetry, water properties or time of day). Data collection on co-located collaboration dynamic would most likely be qualitative descriptions of the collaboration patterns emerging in the two conditions (classic single view immersive vs. multiview immersive).

Each user study session should involve 2-3 users. Depending on the details of the study design it may be possible to test both conditions with each user group, although it may be difficult to fully clear learning effects through task differentiation or randomization. The user study will likely require 10-20 subjects if both conditions are tested for each group and 20-30 subjects if conditions are to be tested separately. In the case of two-user groups, this would result in 5-10 groups in the first scenario and 10-15 in the second (with 5-7 assigned to each condition).

The software infrastructure resulting from this work will also be made public as part of the already-existing Omegalib and SAGE software repositories. I plan to continuously integrate my work into the repositories, as an optional extension or branch of the current software. Other HRE users outside of the Electronic Visualization Lab may decide to use some core features of this work as they become available, and may provide additional qualitative data to support evaluation.

8 Timeline

Figure 17 illustrates the work timeline for my proposed research, including time set aside for dissertation and paper writing. One and a half month is reserved to each of the three major implementation goals discussed in section 5. Most of the summer will be dedicated to refinements and the finalization of an evaluation study. The actual evaluation should take place at the beginning of the fall 2014 semester, with one month dedicated to the analysis of results. The second half of the semester will be reserved for writing one paper or journal article summarizing this research, for finalizing the dissertation and preparing for the defense.

8.1 Publications

The following is a selection of works I already published as first author or co-author, related to hybrid reality environment hardware and software:

1. Febretti, A., Nishimoto, A., Mateevitsi, V., Renambot, L., Johnson, A., Leigh, J. "Omegalib: a Multi-View Application Framework for Hybrid Reality Environments" *to be presented at IEEE Virtual Reality (IEEE VR 2014), Minneapolis, MN, 03/29/2014 – 04/02/2014*
2. Reda, K., Aurisano, J., Febretti, A., Leigh, J., Johnson, A. "Visualization Design Patterns for Ultra-Resolution Display Environments" *In Proceedings of the Workshop on Visualization Infrastructure and Systems Technology (VISTech '13), Denver, CO, 11/22/2013 - 11/22/2013*
3. Reda, K., Febretti, A., Knoll, A., Aurisano, J., Leigh, J., Johnson, A., Papka, M., Hereld, M. "Visualizing Large, Heterogeneous Data in Hybrid-Reality Environments" *Computer Graphics and Applications, IEEE, 07/01/2013 - 08/01/2013*
4. Febretti, A., Nishimoto, A., Thigpen, T., Talandis, J., Long, L., Pirtle, JD, Peterka, T., Verlo, A., Brown, M., Plepys, D., Sandin, D., Renambot, L., Johnson, A., Leigh, J. "CAVE2: A Hybrid Reality Environment for Immersive Simulation and Information Analysis" *Proceedings of IS&T/SPIE Electronic Imaging, The Engineering Reality of Virtual Reality 2013, San Francisco, CA, 02/04/2013 - 02/05/2013*
5. Febretti, A., Mateevitsi, V.A., Chau, D., Nishimoto, A., McGinnis, B., Misterka, J., Johnson, A., Leigh, J. "The OmegaDesk: Towards A Hybrid 2D & 3D Work Desk" *7th International Symposium on Visual Computing (ISVC11), Las Vegas, Nevada, 09/26/2011 - 09/28/2011*

9 References

[1] C. Cruz-Neira, J. Leigh, M. Papka, C. Barnes, S. M. Cohen, S. Das, R. Engelmann, R. Hudson, T. Roy, L. Siegel, C. Vasilakis, T. A. DeFanti, and D. J. Sandin, "Scientists in wonderland: A report on visualization applications in the CAVE virtual reality environment," in *Proceedings of 1993 IEEE Research Properties in Virtual Reality Symposium*, 1993, pp. 59–66.

[2] C. Andrews, A. Endert, and C. North, "Space to think: large high-resolution displays for sensemaking," *Proceedings of the 28th international conference on Human factors in computing systems*, 2010.

[3] B. Yost, Y. Haciahmetoglu, and C. North, "Beyond visual acuity: the perceptual scalability of information visualizations for large displays," *Proceedings of the SIGCHI conference on Human Factors*, 2007.

[4] T. a. DeFanti, D. Acevedo, R. a. Ainsworth, M. D. Brown, S. Cutchin, G. Dawe, K.-U. Doerr, A. Johnson, C. Knox, R. Kooima, F. Kuester, J. Leigh, L. Long, P. Otto, V. Petrovic, K. Ponto, A. Prudhomme, R. Rao, L. Renambot, D. J. Sandin, J. P. Schulze, L. Smarr, M. Srinivasan, P. Weber, and G. Wickham, "The future of the CAVE," *Central European Journal of Engineering*, vol. 1, no. 1, pp. 16–37, Nov. 2010.

[5] T. A. DeFanti, J. Leigh, L. Renambot, B. Jeong, A. Verlo, L. Long, M. Brown, D. J. Sandin, V. Vishwanath, Q. Liu, M. J. Katz, P. Papadopoulos, J. P. Keefe, G. R. Hidley, G. L. Dawe, I. Kaufman, B. Glogowski, K.-U. Doerr, R. Singh, J. Girado, J. P. Schulze, F. Kuester, and L. Smarr, "The OptIPortal, a scalable visualization, storage, and computing interface device for the OptiPuter," *Future Generation Computer Systems*, vol. 25, no. 2, pp. 114–123, Feb. 2009.

[6] A. Febretti, A. Nishimoto, T. Thigpen, J. Talandis, L. Long, J. D. Pirtle, T. Peterka, A. Verlo, M. D. Brown, D. Plepys, D. Sandin, L. Renambot, A. Johnson, and J. Leigh, "CAVE2 : A Hybrid Reality Environment for Immersive Simulation and Information Analysis," 1992.

[7] K. Reda, A. Febretti, A. Knoll, J. Aurisano, J. Leigh, A. Johnson, M. E. Papka, and M. Hereld, "Visualizing Large, Heterogeneous Data in Hybrid-Reality Environments," *IEEE Computer Graphics and Applications*, vol. 33, no. 4, pp. 38–48, Jul. 2013.

[8] S. Teasley, L. Covi, M. Krishnan, and J. Olson, "How does radical collocation help a team succeed?," *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, 2000.

[9] D. Angelo, G. Wesche, M. Foursa, M. Bogen, and D. d'Angelo, "The Benefits of Co-located Collaboration and Immersion on Assembly Modeling in Virtual Environments," *Advances in Visual Computing*, pp. 478–487, 2008.

[10] R. Jagodic, "Collaborative Interaction And Display Space Organization In Large High-Resolution Environments," *Ph.D. Dissertation*, 2012.

[11] M. M. P. Nieminen, M. Tyllinen, and M. Runonen, "Digital War Room for Design," *Lecture Notes in Computer Science*, pp. 352–361, 2013.

[12] T. C. Hutchinson, F. Kuester, T.-J. Hsieh, and R. Chadwick, "A hybrid reality environment and its application to the study of earthquake engineering," *Virtual Reality*, vol. 9, no. 1, pp. 17–33, Oct. 2005.

[13] T. Allen, "Managing the flow of technology: Technology transfer and the dissemination of technological information within the R&D organization," *MIT Press Books*, 1984.

[14] E. Hutchins and L. Palen, "Constructing meaning from space, gesture, and speech," *NATO ASI Series F Computer and Systems Sciences*, 1997.

[15] R. Wilhelmson, P. Baker, R. Stein, and R. Heiland, "Large Tiled Display Walls and Applications in Metereology, Oceanography and Hydrology," *IEEE Computer Graphics And Applications*, pp. 12–14.

[16] G. P. Johnson, G. D. Abram, B. Westing, P. Navr'til, and K. Gaither, "DisplayCluster: An Interactive Visualization Environment for Tiled Displays," *2012 IEEE International Conference on Cluster Computing*, no. Figure 1, pp. 239–247, Sep. 2012.

[17] M. Beaudouin-Lafon, "Lessons learned from the wild room, a multisurface interactive environment," *23rd French Speaking Conference on Human-Computer Interaction*, 2011.

[18] G. S. Schmidt, O. G. Staadt, M. a. Livingston, R. Ball, and R. May, "A Survey of Large High-Resolution Display Technologies, Techniques, and Applications," *IEEE Virtual Reality Conference (VR 2006)*, pp. 223–236, 2006.

[19] H. Smallman and M. S. John, "Information availability in 2D and 3D displays," *... and Applications, IEEE*, 2001.

[20] R. Springmeyer, M. Blattner, and N. Max, “A characterization of the scientific data analysis process,” *Proceedings of the 3rd …*, 1992.

[21] M. Tory and A. Kirkpatrick, “Visualization task performance with 2D, 3D, and combination displays,” *Visualization and …*, 2006.

[22] N. F. Polys and D. A. Bowman, “Design and display of enhancing information in desktop information-rich virtual environments: challenges and techniques,” *Virtual Reality*, vol. 8, no. 1, pp. 41–54, Jun. 2004.

[23] N. Polys and C. North, “Snap2Diverse: coordinating information visualizations and virtual environments,” *SPIE 5295, Visualization and Data Analysis 2004*, 2004.

[24] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart, “The CAVE: audio visual experience automatic virtual environment,” *Communications of the ACM*, vol. 35, no. 6, pp. 64–72, Jun. 1992.

[25] H. Chung, C. Andrews, and C. North, “A Survey of Software Frameworks for Cluster-Based Large High-Resolution Displays,” *IEEE transactions on visualization and computer graphics*, pp. 1–20, 2013.

[26] G. Humphreys, M. Eldridge, and I. Buck, “WireGL: a scalable graphics system for clusters,” *… on Computer Graphics …*, pp. 129–140, 2001.

[27] G. Humphreys, M. Houston, and R. Ng, “Chromium: a stream-processing framework for interactive rendering on clusters,” *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2002*, 2002.

[28] B. Neal, P. Hunkin, and A. McGregor, “Distributed OpenGL rendering in network bandwidth constrained environments,” 2011.

[29] L. Renambot, A. Rao, and R. Singh, “SAGE: the scalable adaptive graphics environment,” *Proceedings of WACE*, vol. 9, no. 23, 2004.

[30] S. Eilemann, “Equalizer: A scalable parallel rendering framework,” *IEEE transactions on visualization and computer graphics*, vol. 15, 2009.

[31] K.-U. Doerr and F. Kuester, “CGLX: a scalable, high-performance visualization framework for networked display environments.,” *IEEE transactions on visualization and computer graphics*, vol. 17, no. 3, pp. 320–32, Mar. 2011.

[32] K. Ponto, K. Doerr, and T. Wypych, “CGLXTouch: A multi-user multi-touch approach for ultra-high-resolution collaborative workspaces,” *Future Generation Computer Systems*, 2011.

[33] M. Snir, S. Otto, and D. Walker, *MPI: the complete reference*. 1995.

[34] J. Allard and B. Raffin, “A Shader-Based Parallel Rendering Framework,” *IEEE Visualization 2005 - (VIS’05)*, pp. 17–17, 2005.

[35] G. V. J. B. Dirk Reiners, “OpenSG: Basic Concepts.”

[36] M. Roth, G. Voss, and D. Reiners, “Multi-threading and clustering for scene graph systems,” *Computers & Graphics*, 2004.

[37] R. Osfield and D. Burns, “Open Scene Graph,” 2004. [Online]. Available: http://www.openscenegraph.com.

[38] P. Harish and P. Narayanan, “Garuda: A scalable tiled display wall using commodity PCs,” *Visualization and Computer …*, 2007.

[39] P. Harish and P. Narayanan, “Culling an object hierarchy to a frustum hierarchy,” *Computer Vision, Graphics and Image …*, 2006.

[40] E. Pietriga, S. Huot, M. Nancel, and R. Primet, “Rapid development of user interfaces on cluster-driven wall displays with jBricks,” *Proceedings of the 3rd ACM …*, no. June, 2011.

[41] E. Pietriga, “A toolkit for addressing hci issues in visual language environments,” *Visual Languages and Human-Centric Computing, …*, 2005.

[42] M. Kaltenbrunner and T. Bovermann, “TUIO: A protocol for table-top tangible user interfaces,” *… . of the The 6th Int’l …*, 2005.

[43] T. Gjerlufsen, C. Klokmose, and J. Eagan, “Shared substance: developing flexible multi-surface applications,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 3383–3392.

[44] J. Mangan, D. Srour, and F. Kuester, “MediaCommons: A Scalable Abstraction for Tiled Display Environments,” *sc13.supercomputing.org*.

[45] M. Szymanski, “CAVELIB Support For PC Visualization Clusters,” *ADVANCED IMAGING-MELVILLE NY THEN FORT …*, 2004.

[46] C. Cruz-Neira, D. Sandin, and T. DeFanti, “Surround-screen projection-based virtual reality: the design and implementation of the CAVE,” *… of the 20th annual conference on …*, 1993.

[47] W. Sherman, “FreeVR,” 2005.

[48] W. Sherman, D. Coming, and S. Su, “FreeVR: honoring the past, looking to the future,” *IS&T/SPIE …*, 2013.

[49] P. McDowell and R. Darken, “Delta3D: a complete open source game and simulation engine for building military training systems,” *The Journal of Defense …*, 2006.

[50] a. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, “VR Juggler: a virtual platform for virtual reality application development,” *Proceedings IEEE Virtual Reality 2001*, pp. 89–96, 2001.

[51] “The Visualization Toolkit.” [Online]. Available: http://www.vtk.org/.

[52] J. Schulze, A. Prudhomme, P. Weber, and T. A. Defanti, “CalVR: an advanced open source virtual reality software framework,” *SPIE 8649 The Engineering Reality of Virtual Reality*, 2013.

[53] J. Schulze and D. Acevedo, “Democratizing rendering for multiple viewers in surround vr systems,” *IEEE Symposium on 3D User Interfaces (3DUI)*, pp. 77–80, 2012.

[54] R. Kuck, J. Wind, K. Riege, and M. Bogen, “Improving the avango vr/ar framework: Lessons learned,” *5th Workshop of the GI-VR/AR …*, 2008.

[55] A. Febretti, V. Mateevitsi, D. Chau, A. Nishimoto, B. McGinnis, J. Misterka, A. Johnson, and J. Leigh, "The OmegaDesk: towards a hybrid 2D and 3D work desk," *Advances in Visual Computing*, pp. 13–23, 2011.

[56] T. Holtkämper, S. Scholz, A. Dressler, M. Bogen, and A. Manfred, "Co-located collaborative use of virtual environments," *Proceedings AAPG Annual Convention and Exhibition.*, pp. 1–6, 2007.

[57] P. George, "Nomad devices for interactions in immersive virtual environments," *IS&T/SPIE ...*, 2013.

[58] K. Richmond, A. Febretti, S. Gulati, C. Flesher, B. P. Hogan, A. Murarka, G. Kuhlman, M. Sridharan, A. Johnson, W. C. Stone, J. Priscu, P. Doran, C. Lane, D. Valle, C. Science, S. M. St, L. J. Hall, and W. T. S. Chicago, "Sub-Ice Exploration of an antarctic lake: results from the Endurance Project," in *17th International Symposium on Unmanned Untethered Submersible Technology (UUST11)*, 2011.

[59] M. Tsang and G. Fitzmaurice, "Boom chameleon: simultaneous capture of 3D viewpoint, voice and gesture annotations on a spatially-aware display," *Proceedings of the 15th ...*, vol. 4, no. 2, pp. 111–120, 2002.

[60] R. Aspin and K. H. Le, "Augmenting the CAVE: An Initial Study into Close Focused, Inward Looking, Exploration in IPT Systems," *11th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'07)*, pp. 217–224, Oct. 2007.

[61] N. Kukimoto and J. Nonaka, "Scientific visualization in collaborative virtual environment with PDA-based control and 3D annotation," *JSME International Journal ...*, vol. 48, no. 2, pp. 252–258, 2005.

[62] F. Berwein, "Superimposed Displays," 2009.

[63] A. Febretti, "Evaluating the Flash platform for web-based collaborative data visualization," University of Illinois at Chicago, 2008.

[64] A. Febretti and A. Nishimoto, "The FlowViz Hybrid Visualization Tool," 2009. [Online]. Available: http://febretpository.googlecode.com/svn/site/flowviz/index.shtml.