A Survey of Software Frameworks for Cluster-Based Large High-Resolution Displays

Haeyong Chung, Christopher Andrews, and Chris North

Abstract—Large high-resolution displays (LHRD) enable visualization of extremely large-scale data sets with high resolution, large physical size, scalable rendering performance, advanced interaction methods, and collaboration. Despite the advantages, applications for LHRD can be developed only by a select group of researchers and programmers, since its software implementation requires design and development paradigms different from typical desktop environments. It is critical for developers to understand and take advantage of appropriate software tools and methods for developing their LHRD applications. In this paper, we present a survey of the state-of-the-art software frameworks and applications for cluster-based LHRD, highlighting a three-aspect taxonomy. This survey can aid LHRD application and framework developers in choosing more suitable development techniques and software environments for new LHRD applications, and guide LHRD researchers to open needs in LHRD software frameworks.

Index Terms—Large high-resolution display, tiled displays, distributed rendering, parallel rendering, distributed applications, graphics api, large scale visualization, programming models, input devices and strategies

1 Introduction

Advances in information and communication technologies, storage density, and increasingly sophisticated data acquisition technologies including high-fidelity laser scanners, satellite imagery, electron microscopes, etc., has led to an explosion of data. One approach to trying to make sense of the mountains of data being collected is visualization. Visualization leverages innate human abilities to illuminate patterns, trends and outliers in the data and provides easily accessible context to individual data points. However, the amount of information that can be simultaneously visualized is limited by the physical constraints of the display medium [1]. While aggregation and interaction can alleviate this problem, alleviation comes at the cost of reduction in the directness of the mapping between the visually perceived representations and the underlying data.

Large high-resolution displays (LHRD) address this issue by greatly expanding the physical size and the number of available pixels, enabling the visualization of large amounts of detailed data. A number of studies have demonstrated that the use of an LHRD for visualization yields a number of benefits, including improved user performance, increased levels of immersion, productivity, memory, and peripheral awareness in various large scale analysis tasks [2], [3], [4], [5], [6].

- *H. Chung and C. North are with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24060. E-mail: {chungh, north}@vt.edu.*
- *C. Andrews is with the Department of Computer Science, Middlebury College, Middlebury, VT 05753. E-mail: candrews@middlebury.edu.*

Manuscript received 13 Dec. 2012; revised 10 Nov. 2013; accepted 20 Nov. 2013. Date of publication 22 Dec. 2013; date of current version 27 June 2014. Recommended for acceptance by J.-D. Fekete.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TVCG.2013.272

While there are a number of techniques for constructing an LHRD system, cluster-based multi-tiled displays provide the greatest scalability as the number of pixels is not limited by the performance capabilities of a single machine [7]. However, developing software applications for this type of system is not easy, given the distributed nature of the environment. Special consideration must be given to synchronizing and rendering images seamlessly across the display tiles, sharing data, maintaining performance given the large quantity of information being displayed, facilitating user interaction, etc.

To help developers address some of these issues, numerous software frameworks have been developed to support cluster-driven LHRD systems since the early 1990s [8]. It is critical for developers to use and select appropriate development tools and techniques for their specific large display projects, since small developmental choices in LHRD software can affect the success of the application directly.

The main purpose of this paper is to survey the available approaches and frameworks and to aid developers and researchers involved in the development of LHRD applications and new toolkits. We explore the state-of-the-art of software frameworks for LHRD applications, and analyze various characteristics and developmental features of the software frameworks.

This paper is organized as follows: cluster-based hardware and software systems for LHRD are briefly discussed, focusing on different LHRD form factors. We survey different types of LHRD applications and identify several requirements in view of their developmental, performance, and interaction aspects. Next, we analyze and taxonomize the architectural design space of LHRD software frameworks. We then review software frameworks based on their target applications and unique features, utilizing a three-aspect taxonomy we constructed. This work concludes with a discussion and comparison

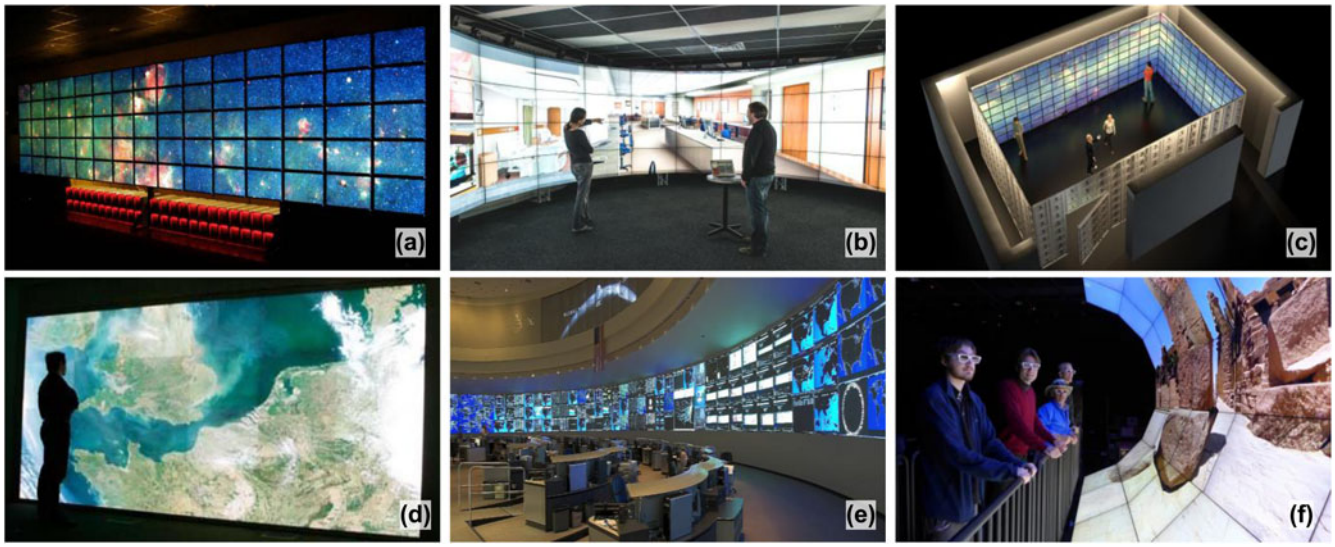


Fig. 1. Various configurations of cluster based LHRDs: (a) TACC Stallion composed of 75 30-inch LCD displays, (b) EVL/UIC CAVE2 composed of 72 near-seamless LCD panels driven by a 36-node cluster, (c) Stony Brook University Reality Deck composed of 416 LCD displays driven by an 18-node cluster (© 2013 Arie Kaufman, Stony Brook University), (d) Fraunhofer IGD Heyewall using 48 projectors and 48 nodes, (e) AT&T's 12 foot high by 250 foot wide rear-projection wall (© 2013 AT&T), (f) Calit2/UCSD Wave Display shaped like an ocean wave with 35 3D LED-backlit LCD panels.

of the most common frameworks and open opportunities for future LHRD frameworks.

2 Cluster-Based Large High Resolution Displays

Due to the difficulty of constructing large displays, tiled displays are a common approach to pushing the bounds of physical size and resolution beyond what is commercially available. To support this perspective, the individual tiles are chosen to be as uniform as possible, they are packed together as tightly as possible to minimize the space between tiles, and the displayed content is designed to show contiguous views, creating the illusion of one single, continuous display space (e.g., Fig. 1).

2.1 Characteristics

There are two main technologies used to provide the individual tiles: projectors (e.g., Figs. 1d, and1e) and LCD panels (e.g., Figs. 1a, 1b, 1c, and 1f). Rear-projection displays have the advantage that the individual tiles can directly abut or even overlap, creating a truly seamless display. In addition, projectors make it easy to create very large physical displays, and with proper configuration, projected displays can conform to a variety of surfaces, such as the smooth curve in Fig. 1e. LCD panels, on the other hand, offer a higher pixel density, creating the opportunity to present more detailed information to users working in close proximity to the display and provide greater clarity. The panels also tend to be brighter, and to have greater consistency with respect to brightness and color than projectors [9] which are subject to changes in the color temperature of the lamp as it ages. LCDs also occupy a smaller footprint in the physical space.

The simplest approach to driving a multi-tiled display is to use a single powerful computer integrated with multiple graphics cards. Recent advances, such as the AMD Eyefinity and FirePro graphics cards [10], which allow up to six monitors to be connected to a single PCIe graphics card, have made this approach even more practical. However, considerations such as the number of available expansion slots, the bus capacity, and the number of outputs and their associated maximum resolutions place hard limits on the size of the display. To solve these problems, large, multi-tiled displays are generally driven by a PC cluster. The cluster-based approach for LHRD enables the following benefits:

Performance and display size scalability. Performance bottlenecks may place limits on the number of tiles and/or the size of the display. Cluster-based large displays provide performance scalability and, nevertheless, support the creation of much larger displays than the single machine approach, by distributing the workload across different nodes in the cluster.

Scalable memory capacity. Cluster-aware applications can enable the user to take advantage of an extremely large amount of memory since nearly all of the memory space in the cluster can be used as a cache [11].

Upgradability and extensibility. The capabilities of the cluster-based display are not fixed by its initial configuration. Additional rendering and display capacity can be added later with the addition of new displays and machines. The system can take advantage of more current technologies, since the commodity industry regularly releases new and more powerful devices with decreasing costs. As compared with special purpose hardware, its compliance with standards favors software and hardware interoperability [12].

Flexible modularity and adaptability. This type of system supports flexible modularity, which enables users to customize hardware components, display sizes, and input devices that are better suited to the user's tasks and environment where the display will be installed [7].

2.2 Components and Software Frameworks

An LHRD cluster consists of two types of nodes based on their roles. The first type is the *head node*, which controls and

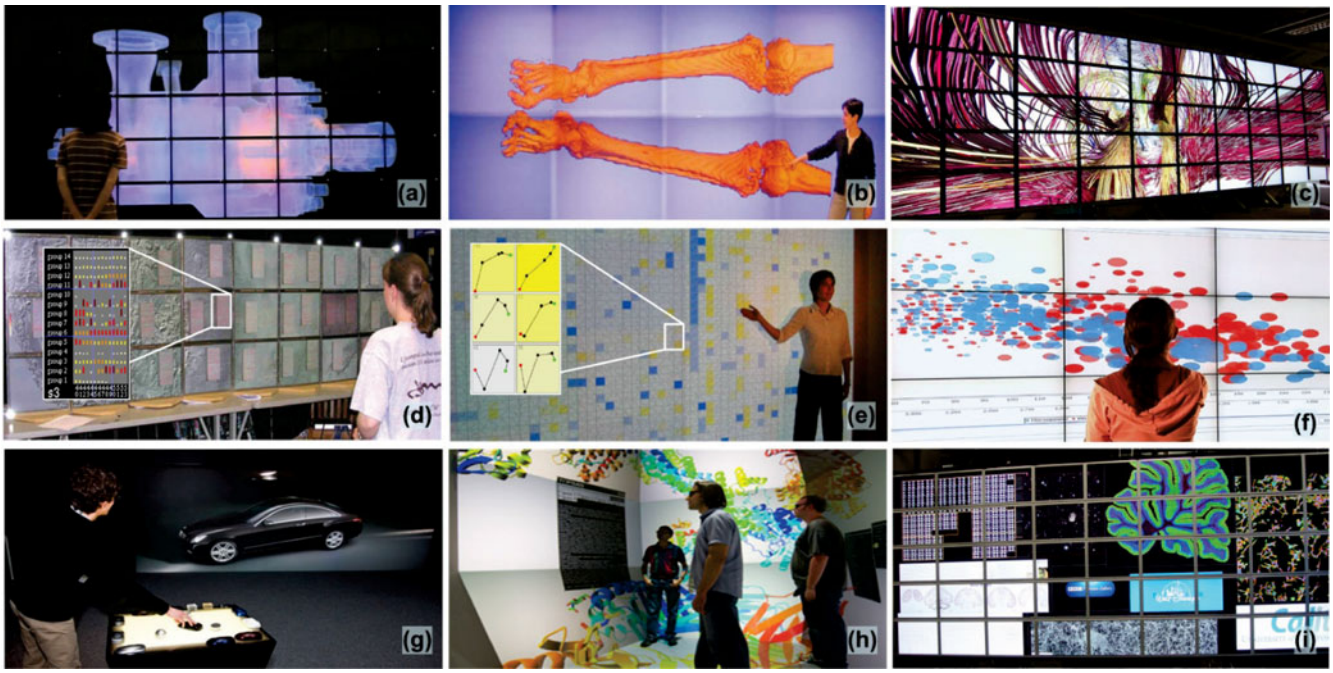


Fig. 2. LHRD applications. scientific visualization: (a) Large-scale pump data by using particle-based volume rendering (PBVR) [13], (b) Isosurface visualization of the visible woman data set [14], (c) Whole-brain DTI tractography visualization [15]; information visualization: (d) Space-centric visualization [2], (e) Effective visual correlation analysis in large trajectory-databases [16], (f) Articulate, a system supporting natural language interaction [17]; immersive visualization: (g) Mercedes-Benz Stuttgart Design Studio Powerwall which allows for 1:1 scale car modeling with tangible interfaces [18], (h) Exploration of proteins from the protein data bank in 3D in StarCave [19]; imagery and multimedia viewing: (i) Tileviewer showing seven different big images, 200-600M pixel resolution each, and two videos [20].

manages all other nodes by distributing commands and synchronizing events, data, and display outputs, according to specific LHRD configurations. The second type, *display node*, is responsible for driving a single or multiple display tiles and for rendering frames on the tiles. These nodes must coordinate their outputs, data, and tasks through the network, creating a large coherent image across multiple display tiles.

The main function of a cluster-based LHRD framework is to handle the details of synchronizing and distributing the rendering tasks across these nodes so that the developer (or the user) can focus on the higher-level functionality of the application. The framework enables the sections of the large coherent image to be rendered and displayed on each display tile in parallel.

When the image requires animation, the frame changes to each tile are synchronized by the framework across different nodes and display tiles of an LHRD. However, the amount of time to render each tile might not be the same in every node because the computing and data processing times in each node can be different due to several performance factors including latency, memory bandwidth, CPU/GPU speed, etc. In order to achieve synchronization, the head node blocks the swaps of the front and back frame buffers of the display nodes until "end of frame" messages have been received from all display nodes (Figs. 3a and 3b). Once all of the nodes have sent the message, the head node unlocks the barrier to swap the frame buffers at the same time by multicasting unlock messages to all display nodes.

Most of the software frameworks guarantee the swap buffer synchronization across multiple display tiles without any hardware support, since such a software-based approach is considered to be a more cost-effective solution and provides flexible and adaptive solutions for specific LHRD applications and display configurations [21]. However, for more accurate and rapid frame synchronization, some of the frameworks (e.g., [22]) support hardware-based synchronization that provides both swap barrier and video refresh through direct connections to the graphics cards on each display node (e.g., Quadro G-Sync graphics cards).

These processes distinguish the LHRD frameworks from other tools designed to perform general-purpose computing on clusters.

3 Applications and Requirements for LHRD

There are a number of different application domains for LHRDs. The application plays an important role in picking (or developing) a software framework, as each domain comes with its own set of requirements and bottlenecks. In contrast to the previous LHRD application surveys [9], [23], we focus on the attendant requirements and characteristics of the primary LHRD application domains in this section.

3.1 Immersive Virtual Environments and Modeling

The 3D virtual environments and modeling applications allow users to see high-fidelity models or immersive virtual environments at amplified scales and multiple degrees of detail through physical navigation. LHRDs offer a compelling alternative to more dedicated systems such as CAVEs (Figs. 1b and 2h) or head mounted displays, since they typically offer a greater data density. Some typical applications include geospatial exploration [24], architecture walkthroughs [11], [19], and design exploration [25], [18], [26] such as the 1:1 automotive model (Fig. 2g). The rendering

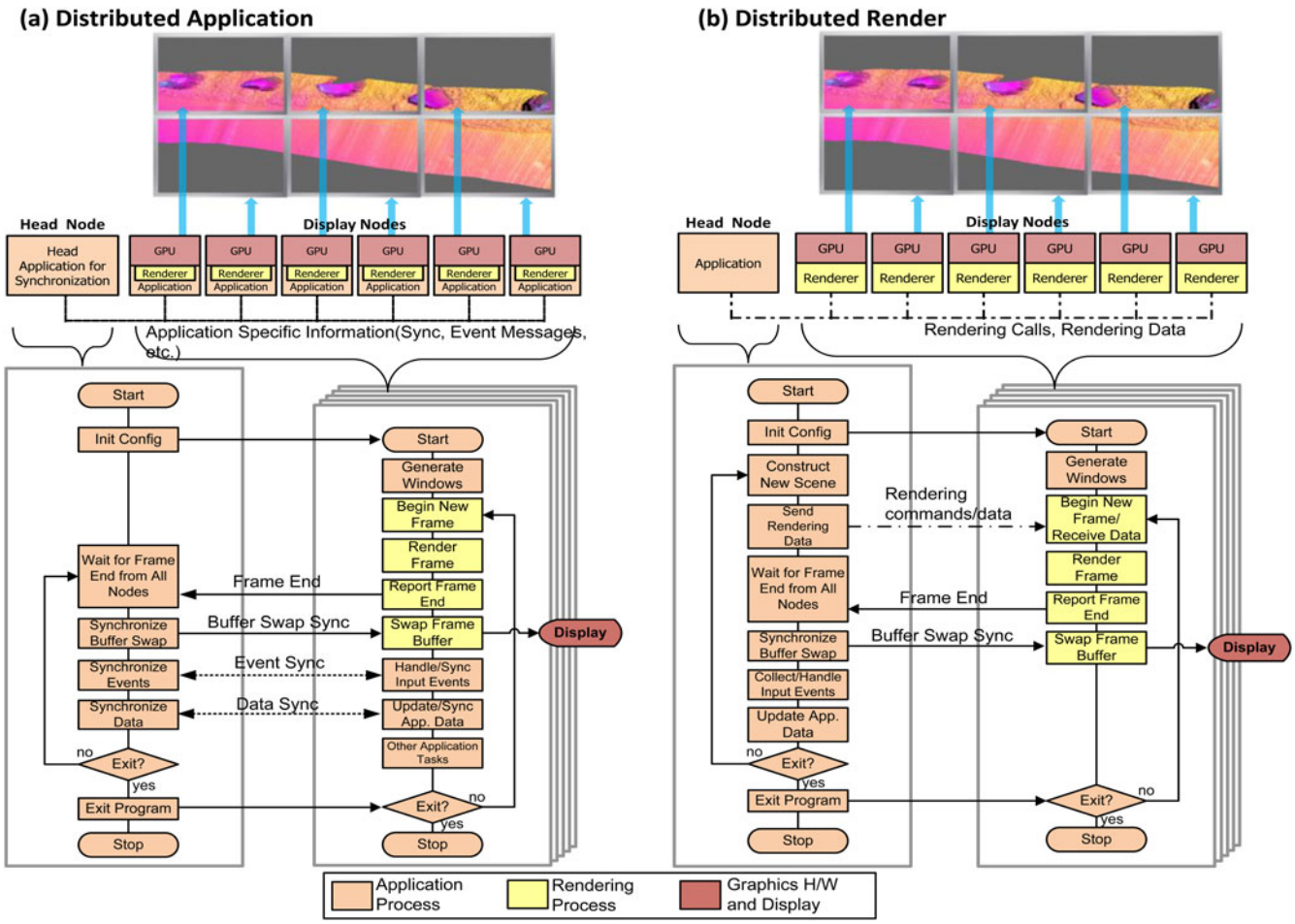


Fig. 3. Examples of two task distribution models for cluster-based LHRD.

and interaction characteristics of these applications tend to focus on pointing and object manipulation, and rely more on the exploration of static models than on highly dynamic views [7]. Because the scene changes between frames are largely achieved through transformations of the view, the best performance for these applications is typically achieved with frameworks that allow the underlying model to remain resident on the display nodes, since such frameworks can avoid transmitting a large amount of 3D model and texture data to display nodes (see Sections 5.2.2 and 8.2).

3.2 Scientific Visualization (SciVis)

The use of LHRDs for scientific visualization is driven by the need to model complex phenomena. Applications span a wide variety of scientific and engineering disciplines including biomedical science [27], [28], genomics [9], molecular dynamics [29], geosciences [30], [31], climate and atmosphere [32], [33], space science [34], [35], and building structures and architectures [36], etc. While some applications use the display space to layout multiple views [37], [38], the most common use for the display is to view a single large highly-detailed visualization. The high pixel count allows the user to observe detailed connections and interactions without losing the context of the overall structure of the data.

SciVis on LHRD typically focuses on 3D volume rendering [39], large point clouds [13] (Fig. 2a), Isosurface [14], [9], [40] (Fig. 2b), and related techniques for working with three dimensional structures. The quantity of data from which these models are derived can also be enormous, requiring multiple petabytes of information. The data set may easily exceed the available memory and computing capability of a single machine. To render these models, it is important to be able to leverage the full rendering capabilities of the cluster. The software framework underlying the application should support efficient data and rendering distribution and out-of-core methods [11]. Also of concern is load balancing [41], [42] moving rendering tasks to underutilized nodes in the cluster to achieve maximum performance.

The primary focus of research involving SciVis on LHRD has been these rendering issues. As a result, interaction has been typically reduced to a very basic set of navigation tools for exploring the displayed model. This is what most of the software frameworks support, but there is clearly more research that could be done in this area to support more complex tasks.

3.3 Information Visualization (InfoVis) and Visual Analytics

Information visualization is characterized by the use of representations for more abstract data. While there is some use of 3D representations, InfoVis applications are more commonly two-dimensional. The additional pixels of LHRD are especially useful for InfoVis where scalability has typically

been limited by the number of pixels available on a conventional display. LHRD applications in InfoVis include geospatial visualization [3], [2] (Fig. 2d), analysis in large trajectory-databases [16] (Fig. 2e), natural language queries [17] (Fig. 2f), and sensemaking to large textual data [6]. The representations are built up using simple 2D or 3D glyphs such as points, lines, images, and text, [43], [16]. The potential for graphics primitives, such as long lines and large polygons, spanning multiple tile boundaries is greatly increased, making it more difficult to distribute the rendering tasks.

Unlike the complex and largely static models used for virtual environments and scientific visualization, InfoVis applications tend to be more dynamic. While navigation is still a fundamental task, other tasks such as selection, filtering, and annotation are equally important [44]. There are some implications for InfoVis on LHRD. Each individual glyph in the visualization potentially can change based on the interaction. The changes wrought by interactions can create significant alterations in the displayed representations at multiple levels of scale. The dynamic nature of the displayed objects (or glyphs) leads to significant geometry updates spanning multiple (if not all) display tiles and nodes on the LHRD [45], [46].

3.4 Command and Control

The key objective of command and control applications is to support real-time situational awareness and collaborative decision making tasks for co-located teams of users. LHRD command and control systems are used to support a broad range of fields including military [47], aerospace, telecommunications [48], [49], large facility management [50] and energy deployment and distribution [51]. The information presented in command and control situations typically consists of multiple windows (Fig. 1e), sometimes exported from individuals' computers. Aggregating exported windows for a large display can add an additional challenge for synchronization, as the disparate applications may be running at different frame rates [21]. Since users typically sit at their personal workspaces within the command center, interaction is achieved by a conventional tethered interface such as a keyboard and mouse through the user's own personal computer rather than with the large display directly.

3.5 Imagery and Multimedia Viewing

Basic image and multimedia viewers are fairly straightforward uses of large displays. There are two basic use cases for these kinds of applications. The first is to enable the user to view large, high-resolution imagery or media at a resolution that supports the analysis of critical details without losing the overall context of the source material. This approach is useful for analyzing the imagery produced by satellites [52], radio telescopes [53], electron microscopes [54], etc. The second use case is to use the display space to view multiple images for organization or juxtaposition purposes (Fig. 2i) [20].

These types of applications need a relatively small number of polygons and fewer geometry changes to display images, as the images are treated as atomic units (typically in the form of textures). However, it is challenging to manage, load, and display a large number of high-resolution images across the display nodes. Due to the limits on texture size in the main memory and GPU's texture memory, huge images generally need to be segmented into smaller pieces in a preprocess stage, and a collection of the image pieces can also be pre-generated in different resolutions to accelerate rendering speed [20], [55], [56]. An efficient solution should provide ways of distributing texture data, minimizing the amount of data that must be handled by any particular node, and caching it (i.e., out-of-core approaches such as [57]) to reduce the network traffic.

Video content is more difficult to deal with due to its dynamic nature. On the other hand, truly high-resolution video content is fairly rare, so the problem is generally one of streaming conventionally sized content to the appropriate tiles or intelligently scaling it across the tiled display [58]. There are a number of video streaming approaches and applications for cluster-based LHRDs [59], [60], as well as some frameworks which support streaming media [61], [62], [63] (see Sections 8.1.2, 8.3.4, and 8.4.2).

4 Three Facets of The Taxonomy

There are many ways for the frameworks to address the requirements and characteristics of the LHRD applications, in addition to other functionalities to consider. To attempt to classify the various approaches, we propose a faceted taxonomy. A single taxonomy is insufficient to fully describe LHRD software frameworks, since the frameworks are typically composed of a set of different functionalities. Each facet represents a different aspect of the frameworks. On the basis of our application survey, we identified three main facets for the taxonomy:

Task Distribution Models. The rendering performance in the LHRD cluster is influenced by certain architectural factors and network utilization significantly. These include parallelizing, load-balancing, and distributing application and rendering tasks across nodes in the cluster. This facet examines how each framework distributes responsibility, and has the greatest impact on the overall performance of the application (Section 5).

Input Handling Models. The Windows, Icons, Menus, and a Pointer (WIMP) interaction model may not be well supported by the LHRD. Important concerns are, which interaction techniques are supported, where events are being generated and collected (e.g., can individual nodes generate interaction events or is one machine handling all input?), and what information is propagated to the nodes (Section 6).

Programming Models. While technical aspects play a large role in determining the performance of the application, it is also important to consider the development process as well. Since cluster-based LHRDs are not numerous, there are very few developers dedicated to developing LHRD applications. Thus, we must consider the overhead imposed on the developer by the framework. For example, some frameworks require the developer to follow a particular pattern or structure, which may, or may not, be used in other domains (Section 7).

Table 1 summarizes the advantages/disadvantages of each model within all three facets of the taxonomy.

TABLE 1
Comparison within Three Facets of the Taxonomy

Three Facet	Taxonomy	Pros	Cons
Task Distribution Models	Distributed Application	- Reduce network traffic among the nodes. - Utilize data from local memory or storages avoiding sending a large amount of information. - Can add and remove display nodes dynamically during runtime.	- Relatively poor utilization of the distributed resources of a cluster for performance scalability and data management. - Require handling synchronizations of nondeterministic applications that use random number generators.
	Distributed Renderer	- Can support rendering scalability by dividing the rendering task and data into small parts and executing them in parallel on nodes - Facilitate the management of extremely large rendering data. - Minimal computational requirements for the display nodes.	- High network traffic between the nodes. - Single head node performs the bulk of the computation.
Input Handling Models	Centralized Event Handling	- A head node handles all events for user input so that it can reduce the workload for display nodes. - Distributing events is simple and uni-directional from the event server to the other nodes.	- If multiple input devices are tightly coupled to each display node (e.g., touch interfaces, cameras, etc.), this model may requires additional programming for the devices.
	Distributed Event Handling	- Spreading input devices across the cluster increases the number of resources available to manage them. - Spread computational tasks such as raw input data into higher-level information across the cluster.	- More sophisticated ways to synchronize and share input events require complexity in the framework.
Programming Models	Non-invasive	- Transparently support existing graphics APIs of single workstations. - Developers can focus on the application logic, without significant concern about the nature of the cluster system and tiled display.	- No control over the parallel portion of the application. - Do not support all features of the original graphics APIs. - The potential of a disconnect between the expected behavior of a function and the behavior implemented in the framework.
	Minor Code Modification	- Support minimal control over the distributed graphics system - Minimize the degree to which developers need to learn about low-level cluster systems and parallel rendering.	- Limited control over the parallel portion of the application.
	Structurally Invasive	- Programmers have finer control and utilization of distributed graphics hardware and input devices.	- Typically require programmers to restructure their graphics code with multiple callback functions. - Require understanding the physical and logical components of the cluster system. - Porting existing code to the cluster-based LHRD may be difficult.

5 Task Distribution Models

Broadly speaking, all graphics intensive programs follow the same basic pattern. The application logic transforms the model into graphics primitives, usually some form of point or vertex data. These primitives are assembled into basic geometry, which is scaled and clipped against the display viewport. This is followed by the rendering process, which performs depth tests and assigns colors to pixels in the framebuffer. When the framebuffer is ready, a "buffer swap" is performed, and the contents are displayed on the screen. Then, inputs are processed, potentially changing the model and the cycle repeats.

With a conventional display, most of this process is performed on the local graphics card. However, on a cluster-based LHRD, at some point in this process information needs to be distributed to the networked nodes. One of the most important issues is how rendering tasks are distributed across the nodes and how each node retains the data.

We can classify Task distribution models based on the point within the rendering process at which data is shared among the nodes in the cluster. Thus, the question is about which tasks are performed on each LHRD node and which information is communicated. There have been several other proposed classifications based on the distribution of responsibility, most notably Chen et al. [64] and Staadt et al. [45]. Other classifications were also used [7], [62], [29], [65], [66], and [67]. However, most of these use terminology that is removed from the actual task being performed, leading to less descriptive, and in some cases contradictory labels for another classification.

We propose two categories: *distributed application*, and *distributed renderer*. In the first, network communication is primarily concerned with duplicating the application state between nodes, while in the later, actual rendering directives (graphics primitives, geometry, scene graph or even raw pixels) are communicated across the network.

5.1 Distributed Application

In this model, identical application instances are run on each node of the cluster using different configuration parameters to determine which tile(s) it is responsible for (Fig. 3a) (see Sections 8.3.1, 8.3.2, 8.3.3, 8.3.4, 8.3.7, 8.4.2, and 8.4.5). While a fully decentralized architecture is possible, this approach usually makes use of a head node which serves as the controller of the entire display (e.g., Fig. 3a). The main duty of the head node is to broadcast any information necessary to keep the system state identical across all of the nodes (i.e., synchronization). This includes any system input, including user interactions, as well as lower-level sources of the application state, such as timer information and random number generation.

The primary advantage of this model is that it has fairly small network bandwidth requirements, since only the user's input events and important application state information need to be propagated across all nodes to synchronize the distributed applications.

The intelligence of both the application and the software framework can significantly affect the performance characteristics of this approach. Since every node runs the same application, the naïve approach would be for each node to attempt to render the entire scene, relying purely on the viewport clipping to reduce the amount of geometric information actually rendered and displayed. Thus, this model can be improved through support for object culling in the framework [68], or by intelligent processing at the application level to avoid considering objects that will not be displayed by this node. Of course, this later approach puts a greater burden on the application developer, and there is the potential that

determining which objects are displayed by this node is more computationally expensive than simply rendering everything.

This model has some downsides in terms of utilization of the computational resources, performance scalability, and data management. First, since the full application and data may be maintained and executed in every node, this repeats the same, potentially expensive, computations on every node. So CPU performance has the potential to become a bottleneck. Similarly, if each node is working independently, memory contents may be duplicated in every node, removing any benefit of the large amount of memory provided by the cluster as a whole. In addition, data management is a key issue. Each node may maintain a copy of the entire data set, or there may be a centralized shared database accessible to all of the nodes, which would cause additional network traffic.

5.2 Distributed Renderer

The distributed renderer approach separates the rendering tasks from the core application (Fig. 3b [22]). Here, the head node performs all of the application level logic. The display nodes are responsible for handling graphics operations. This puts the bulk of the computation on the head node, which must take care of executing the application, handling input events, sending and splitting rendering tasks, and synchronizing frames. There is a spectrum of approaches that fall into this category, determined by where the division between the head node and the display node is made. Communication between nodes can be in the form of graphics directives, graphics primitives, geometry, or even actual pixels.

To better separate the application logic from the distribution of the rendering tasks, many frameworks also provide an intermediary process that handles the configuration and resource management for the cluster [69], [22] (see Sections 8.1.3 and 8.4.1). This shields the application from details of the cluster structure. The application sends rendering commands to the intermediary process, which performs the distribution and synchronization.

In contrast to the distributed application model, the actual application can be completely unaware of the cluster, potentially making development easier. In addition, the processes running on the rendering nodes (i.e., display nodes) can be completely removed from the application logic, and can be generalized and thus provided entirely by the framework (e.g., Fig. 3b). A key advantage here is that the computational requirements for the display nodes in the cluster are minimal. Provided they have good graphics capabilities, the nodes can otherwise have mediocre specifications with respect to CPU speed, memory, and disk capacity. However, the main disadvantage of this approach is the high network bandwidth requirements caused by the large amount of graphics information that must be communicated to the display nodes.

Based on how calls from the head node cause rendering on each display node, we can further classify this model into two rendering styles: *Immediate rendering* and *Retained rendering*.

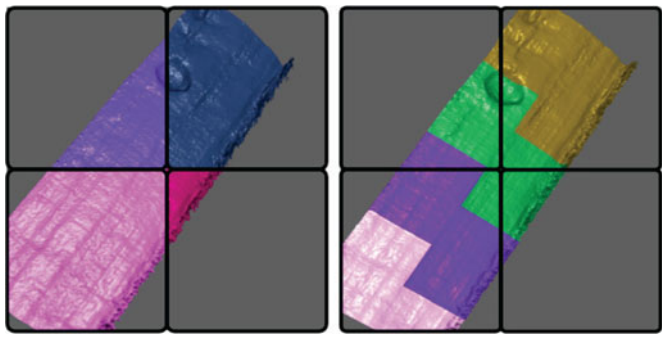


Fig. 4. Sort-first (left) and sort-last (right) on a four-tile display. The color indicates how the rendering was divided between the four nodes.

5.2.1 Immediate and Parallel Rendering

The distributed renderer supports immediate rendering, where the head node issues graphics commands that directly affect the current scene to be displayed (see Sections 8.1.1, 8.1.2, 8.1.3, 8.1.4, 8.3.6, 8.4.1, and 8.4.3). An important attribute of frameworks that follow this approach is the type of rendering information sent to the rendering nodes. The most straightforward method for immediate rendering is to send all of the graphics information (whether commands, primitives, or pixels) from the head node to all of the display nodes, and let each display node determine the portion of the view for which it is responsible. This approach leads to very high network usage. Frameworks that implement this approach will frequently make use of broadcast or multicast communication in an attempt to cut down on the network traffic [70], but the low-level information sent out from the head node still results in problems with network latency. Another issue is that there is a great deal of duplication of effort among the rendering nodes as each node will ultimately render the same objects in an attempt to determine if they fall within the clipping region [45].

This model allows for the use of parallel rendering algorithms instead which first divides the rendering problem into smaller pieces, so that each display node deals with a separate smaller piece of the content [42], [71] (see Sections 8.1.3, 8.2.1, 8.4.1, 8.4.3, and 8.4.4). The most common approach is an algorithm called *sort-first* (or *screen-based decomposition*) [72]. This algorithm sorts graphics primitives based on each display tile's viewport. It divides the entire scene into disjointed tiles (based on the display tiles of an LHRD) and each display node then renders only the graphics primitives that lie within its view space (Fig. 4left). While this approach requires the head node to perform more work, it reduces the network bandwidth requirements, and simplifies the work performed by each of the rendering nodes.

Another approach to parallel rendering is the *sort-last* (or *model-based decomposition*) algorithm [72]. This algorithm divides the model up into similarly sized pieces and distributes them to arbitrarily selected display nodes (Fig. 4right). Since rendering tasks can be evenly distributed across the display nodes, sort-last can provide better processor load balancing than sort-first among the display nodes and thus result in more consistent performance. However, since each node renders an arbitrary fragment of the scene or model, the rendering results

must be collected and composited at the display nodes before they can be displayed. The I/O overheads of this composition processing in the sort-last approach are limiting factors for dynamic visualization on the cluster-based LHRD [12], [73]. When it is used for tiled display setups, sort-last is typically used for static visualization of highly complex models on LHRD rather than for general dynamic visualizations [74]. Samanta et al. [75] have proposed combining sort-last and sort-first to increase rendering performance but this approach is not yet directly implemented by any general LHRD frameworks.

5.2.2 Retained Rendering

To further reduce network traffic during distributed rendering, the rendering nodes can reuse information from previous frames rather than completely refresh at every update [45], [76] (see Sections 8.2.1, 8.2.2, 8.2.3, 8.2.4, 8.2.5, 8.3.4, and 8.4.4). Since the primary bottleneck is the communication channel between the head node and display nodes, the issued high-level graphics commands and data can be stored on the display nodes for later use, without resending unchanged data from the head node.

Several LHRD frameworks support retained rendering at this level. On one hand, *Scene graphs* enable retained rendering at a higher level. Scene graphs are most useful for models that remain coherent (i.e., they are static, or move in well-defined ways). As such, changes across updates will be relatively small (e.g., color changes, view changes, or transformation). Every display node maintains a copy of the entire scene graph and associated graphics primitives. The head node tracks changes of the scene graph to ensure consistency in distributed nodes and broadcasts these changes to every display node. Then, each display node traverses and updates its local scene graph accordingly and performs the rendering tasks on the tiled display. Once the display nodes retain the scene graph, the head node needs to communicate only change sets to the display nodes. Of course, if the interaction involves complex geometry changes, the overall visualization performance will be significantly degraded due to the overhead of traversing and updating the scene graphs in each display node (see Section 8.2).

On the other hand, Allard and Raffin proposed a *shader*-based protocol for retained rendering [40] as another approach to transmit changes between each frame on LHRD. Graphics shaders replace the fixed functionality of the traditional graphics pipeline with programmable units. In the context of cluster-based LHRDs, instead of continually sending attribute data (such as color and geometry), the head node can send simple updated parameters between frames and let the shader on the display nodes handle the rendering procedurally (see Section 8.4.4).

6 User Interaction and Input Handling

Due to the physical size of LHRDs, conventional tethered user interfaces, such as keyboards and mice, may limit physical navigation [3] even if they are placed on a mobile platform. In addition, LHRDs are frequently used as collaborative multi-user environments. As such, there is much interest in alternative input devices and multimodal interfaces (e.g., [77], [78], [79]). Some common devices include:

- Tracking systems (e.g., infrared/magnetic marker tracking system) [77], [79], [80]
- Analog devices (e.g., analog joystick and sensors such as accelerometers and gyroscopes) [77], [81]
- Digital devices (e.g., gamepad, touch overlay, pen, etc.) [82], [83], [84]
- Camera/Computer Vision based approaches (e.g., gesture based interaction) [85]
- Multiple display surfaces (e.g., personal displays, smartphones, and tablets) [78], [86]

These input devices are often combined to create new interaction techniques or virtual input devices (see Section 8.3.4) and integrated interactive workspaces [87]. A complete survey of interaction techniques is beyond the scope of the software framework (for more detailed surveys of LHRD interaction techniques, see [88], [89]). Instead, we will limit our focus to the capabilities of the software frameworks for supporting interaction. We divide the process into two parts: the process of acquiring the raw inputs from multiple potentially-distributed input devices, and the process of handling the input events to change application state.

6.1 Input Acquisition

Most of the frameworks we discuss in this survey leave the problem of input completely in the hands of the developer, although some frameworks provide facilities for acquiring input events through specialized input servers.

Standard event acquisition. The simplest model makes use of a conventional event handling loop that accepts input directly from the OS (see Sections 8.1.1, 8.1.3, 8.2.1, 8.3.7, 8.4.1, and 8.4.5). This allows the application to make use of standard input devices that are physically connected to the head node through I/O ports such as serial or USB ports. For example, developers using Chromium [69] and OpenSG [90], write conventional desktop applications with standard event handling, and they distribute the rendering data after the application state has already been changed by input events.

Ad-hoc acquisition. Novel LHRD interactions are increasingly based on wireless and mobile input devices with various types of input information. One of the primary challenges of developing with novel input devices is that there is limited support at the OS level for transporting inputs across systems and transforming inputs into meaningful interaction events. A typical approach is to build on top of toolkits, such as TUIO [91], ICON [92], Squidy [93], VRPN [94], Opentracker [95], and kivy [96], which provide an abstraction layer between the raw input and the desired interaction events. These toolkits enable communication between input devices and the head node application through various communication protocols for interactions, such as TUIO messages [91] and OpenSound Control (OSC) [97] over the wireless network.

Input server. Some frameworks include their own input server to support a wide range of input devices and their functionalities [98], [99], [63], [86], [87] (see Sections 8.3.4, 8.3.6, and 8.4.2). In general, these input servers are responsible for simultaneously connecting with multiple input devices, gathering input from the devices, and delivering it to the application. The input server creates a form of input

device transparency, where the application is shielded from details about where the device is located and, in some cases, the actual nature of the device itself. These can be configured to abstract the inputs from these devices directly into higher-level tasks such as navigation, transformation, and selection. For example, the jBricks Input Server (jBIS) handles input distribution and manages user interaction through OSC [100]. Syzygy's input server facilitates the combination of input data from multiple devices to form an integrated input device (see Section 8.3.4) [62]. CaveLib [101] supports a more specialized input server for trackers only, which broadcasts tracker position information to the application. In the case of the distributed application frameworks, such as VR Juggler [98] and CGLX [63], the input server can also gather input from applications running on the display nodes, which can be useful if, for example, the tiles are also touch surfaces (see Section 8.4.2). In distributed renderer frameworks, Equalizer [22] is a special case of this, in that it provides facilities to gather input events from the distributed renderers (i.e., display nodes).

6.2 Input Handling Models

Once the input events are acquired, the next question is where the actual handling of the events takes place. There are two ways to handle the input events in the LHRD cluster. These approaches are closely related to the task distribution models (Section 5).

Centralized event handling. The frameworks based on this approach provide a centralized event loop (see Sections 8.1.1, 8.1.3, 8.1.4, 8.2.1, and 8.3.6), and the head node is solely responsible for receiving and handling input events for update. The head node receives all events from input devices or input servers, updates the application state accordingly, and then distributes the updated state or rendering information to the display nodes.

Distributed event handling. The frameworks based on this approach forward the input events out to the display nodes where the events are processed (see Sections 8.3.1, 8.3.2, 8.4.2, and 8.4.5). Depending on the amount of event transformation in the input servers, the information distributed consists of either raw input data or application relevant events, such as navigation commands. Each display node is then responsible for updating its own internal system and application states.

7 Programming Models

Our third comparative dimension, the programming model, is indicative of how a framework is used to create a visualization application on a cluster-based LHRD. Frameworks typically build on existing standard graphics APIs, such as OpenGL, but modify the APIs and/or add functions to support cluster-based LHRD capabilities.

The primary issue is the degree of invasiveness of the framework into the graphics API; that is, how much does the framework API affect the application code from an otherwise standard graphics application. From an application developer's point of view, this is an issue of usability of the API and generality of the resulting applications, versus specificity and power. At a high level, we distinguish three basic LHRD programming models based on how these frameworks support the existing graphics APIs, ranging from transparently non-invasive to structurally invasive.

7.1 Non-Invasive

The non-invasive programming model seeks to support the existing graphics APIs transparently to extend them without modification [102], [103] (see Sections 8.1.3, 8.1.4, 8.2.2, 8.2.3, 8.2.4, and 8.3.3). Programmers can develop an LHRD application using the original graphics libraries designed for conventional desktop hardware, such as OpenGL, OSG, or Open Inventor, without modification or additional parallelization functions. Thus, new LHRD applications can be created without learning a new API for the cluster-based LHRD. Developers can focus on the application logic, without significant concern about the nature of the cluster system and displays. This lowers the barrier to the development of LHRD applications. Also, existing applications can be immediately ported to LHRD systems, and in some cases can be run without recompiling.

Programmers typically need only to create a configuration file for their specific LHRD system, which describes every hardware element associated with the given cluster-based LHRD system. Once the configuration is specified, the framework automatically handles the deployment and parallelization of the application according to the configuration.

7.2 Minor Code Modification

Software frameworks supporting this type of programming model still allow the developer to work primarily with conventional toolkits, such as OpenGL or one of the scene graph implementations, but require part of the code to be purpose written to the LHRD environment (see Sections 8.1.2, 8.3.5, 8.3.6, 8.3.7, 8.4.2, 8.4.3, 8.4.4, and 8.4.5). Therefore, this provides the developer with a minimal control over the synchronization or view-related portions of the LHRD application while minimizing the degree to which the developer needs to be acquainted with knowledge about low-level cluster systems and parallel rendering.

There are two common types of minor code modifications required by LHRD frameworks: (1) adding setup code [61] (see Sections 8.1.2 and 8.3.7), and (2) adding or replacing some extended functions with additional functionality [104], [105], [63] (see Sections 8.4.2 and 8.4.5). First, programmers may add a few lines of application initialization code related to display configuration parameters for the display cluster (e.g., viewing and windowing parameters). Second, programmers may add or replace specific view-related functions for tiled displays, which are used in conjunction with the existing API. In an LHRD application, a single view needs to be subdivided into several independent tiles. Hence, view related functions in the original code might need to be replaced with new functions in the framework that are re-implemented for the cluster-based LHRD. This modification is very simple, and typically requires programmers to add new prefixes to some of the function names, accepting the same function parameters as the originals (see Section 8.4.2).

7.3 Structurally Invasive

The final model involves frameworks that are novel or more specific to an application itself. While frameworks in this category may not be written specifically for LHRDs and are still designed to use existing graphics APIs, they provide their own library that is not found in any other tool or development environment [98], [106], [99], [107] (see Sections 8.2.1, 8.2.5, 8.3.1, 8.3.2, 8.3.4, and 8.4.1). This gives the application developers the freedom to implement features not supported by conventional graphics APIs, but does so at the cost of generality.

This programming model provides programmers with abstractions of various physical and logical entities and functionalities of the cluster-based multi-display systems, such as cluster nodes, displays, input devices, graphics cards, windows, synchronization, etc. [22]. Thus, programmers have finer control over and utilization of distributed graphics hardware and input devices. However, in contrast to the previous two models, this requires programmers to be more aware of physical and logical components of cluster-based rendering systems.

The frameworks with this programming model typically require programmers to restructure their graphics code based on multiple callback functions that are invoked by the frameworks. To develop an LHRD application, programmers override or "fill in the blanks" of the predefined callback methods similar to the callback functions in GLUT. For example, like *glutDisplayFunc()*, the rendering routines are passed as display callback functions that are called by the framework according to the display loop in the rendering process. The contents of these rendering routines in the application code can be written with conventional graphics APIs.

8 LHRD Software Frameworks

In this section, we review a number of currently available software frameworks and toolkits for LHRDs. These frameworks have originated from a variety of communities with a number of different target applications, and thus demonstrate a fairly diverse set of characteristics. All of the frameworks we discuss, however, have either been adapted or developed to support cluster-based, multi-tiled displays.

We organize these LHRD frameworks primarily based on the target applications, and discuss each framework in the context of our faceted taxonomy. Our survey places more weight on the frameworks which have been published and have been actively used for developing various applications on cluster-based LHRDs. We checked the presence and usage of mailing lists and related internet forums for each framework as well as applications written with the framework. For an overall comparison of all of the frameworks we discuss, see the comparison matrix (Table 2).

8.1 Transparent Frameworks for Legacy Applications

The primary goal of these frameworks is essentially to hide most, if not all, evidence that the application is running on a cluster-based display from the developer. These solutions play the role of middleware, application window managers, or OpenGL drivers, allowing the tool to intercept the rendering process at various stages and to stream the results to specialized renderers across the cluster. The framework is employed to run existing applications on the cluster-based LHRD without modification or recompiling. For application development, these frameworks focus on the non-invasive programming model.

These solutions may not always lead to the best performance. Because the details of the task and data distribution model are handled entirely by the framework, the developer has limited opportunity to optimize the application for the cluster, for example by finding opportunities to parallelize the application logic, or finding ways to leverage the distributed memory available in the cluster. Another potential problem is that these frameworks typically work by replacing and re-implementing existing functionality. While this feature provides transparency, it opens the possibility for a potential disconnect between the expected behavior of a function and the behavior implemented in the framework (if it has been implemented at all).

8.1.1 DMX

Distributed Multihead X (DMX) leverages the client/server architecture of X11 to distribute X window information across the cluster [108]. The user selects a node to act as the frontend (i.e., the head node), and runs DMX's replacement X server (Xdmx) on it. DMX accepts X directives and forwards them on to the other machines in the cluster. When coupled with Xinerama [109], it unifies the remote displays into a single virtual desktop, providing complete transparency for any X11 application.

While DMX offers great flexibility in the range of applications it can support, it has significant performance limitations. The X11 protocol consists of very low-level directives for drawing and placing windows and passing along rendering information. DMX adds significantly to this overhead and compounds it with increasing network traffic as the number of back-end servers (i.e., the display nodes) increases. The system mainly supports 2D-based rendering and does not provide the opportunity to parallelize either the application or the rendering tasks. Chromium, which we will discuss shortly, does provide a DMX extension, which improves 3D rendering performance by distributing OpenGL commands to the remote nodes, but the network bandwidth requirements of DMX still restrict it to relatively small clusters.

DMX is an attractive solution if the goal is a windowing environment in which existing X11 applications can be run without modification. However, as the size of the cluster and the complexity of the content to be rendered increases, performance becomes a critical problem.

8.1.2 SAGE

SAGE was developed to integrate multiple visualization applications into a single LHRD [110], [61]. 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 SAGE application interface library (SAIL). To show information on the large display, the application generates

TABLE 2
Summary of Cluster-Based LHRD Frameworks

Name & Reference	Task Distribution Model	Event Handling Model	Programming Model	Graphics APIs	Target Applications
Transparent Frameworks					
DMX [108]	Distributed renderer, Immediate rendering, Distributing X11 calls	Centralized event handling	Non-invasive	X11/Xlib, 2D graphics and GUI API	Legacy X11 applications; Large desktop; Command centers
SAGE [110]	Distributed renderer, Immediate rendering, Distributing 2D pixels but run applications on muliplelnodes	Distributed event handling, Free Space Manager	Non-invasive or minor code modification, add setup routines in OpenGL code	No dedicated API, but supports OpenGL Wrapper (resolution limited)	Heterogeneous collaborative visualization; remote desktop and video streaming; Command centers
Chromium [69]	Distributed renderer, Immediate rendering, Distributing OpenGL calls	Centralized event handling (Input server via CRUT API)	Non-invasive	OpenGL, 3D graphics API	Legacy OpenGL applications
ClusterGL [113]	Distributed renderer, Immediate rendering, Compressed OpenGL calls through multicast	Centralized event handling	Non-invasive	OpenGL, 3D graphics API	OpenGL applications
Distributed Scene Graph Frameworks					
OpenSG [90]	Distributed renderer, Retained rendering, Scene graph	Centralized event handling	Structurally invasive, create separate client and server codes	OpenSG, GLUT, 3D graphics API	Virtual reality applications and visualization
Garuda [116]	Distributed renderer, Retained rendering, Scene graph	Centralized event handling	Non-invasive	OpenSceneGraph, 3D graphics API	Virtual reality applications and visualization
Blue-c Distributed Scene Graph [117]	Distributed renderer, Retained rendering, Scene graph	unknown	Non-invasive	OpenGL Performer, 3D graphics API	Collaborative immersive VR, telepresence, multimedia
Open Inventor Cluster Edition [65]	Distributed renderer, Retained rendering, Scene graph	Centralized event handling	Non-invasive	Open Inventor, 3D graphics API	Large scale scientific visualization
AVANGO NG [118]	Distributed renderer, Retained rendering, Scene graph	Distributed event handling	Structurally invasive	OpenSceneGraph with Python bindings, 3D graphics API	Virtual reality applications
Interactive Application Frameworks					
VR Juggler [98]	Distributed application	Distributed event handling, Input server	Structurally invasive	OpenGL, OpenSG, OpenGL Performer, OpenSceneGraph, VTK, 3D graphics API	Virtual reality applications for HMD, CAVE, Powerwall
CaveLib [101]	Distributed application	Distributed event handling, Input server (for Trackers)	Structurally invasive	OpenGL, OpenGL Performer	CAVEs, Virtual reality applications
JINX [120]	Distributed application	Distributed event handling	Non-invasive to X3D, but need to create X3D loader in C++	X3D, OpenGL, 3D graphics API	Virtual reality applications
Syzygy [62]	Distributed application and Distributed renderer, Scene graph	Distributed event handling	Structurally invasive	Scene graph, OpenGL, 3D graphics API	Virtual reality applications , multimedia
AmiraVR Cluster Ver. [123]	Distributed application	unknown	Minor code modification to AmiraVR	AmiraVR, OpenGL, Open Inventor, 3D graphics API	Virtual reality application, and visualization
jBricks [100]	Distributed renderer, Distributing ZVTM commands	Centralized event handling, Input server - jBIS	Minor code modification to ZVTM, replace camera and view functions	ZVTM, Java2D, Swing, 2D graphics API	Information visualization, user interfaces, interactive applications
MostPixelEverCE [125]	Distributed application, Processing renderer	Distributed event handling	Minor code modification to Processing codes, add some functions for configurations	Processing, 2D graphics API (OpenGL is also partially supported)	Information visualization, visual art
Scalable Rendering Frameworks					
Equalizer [22]	Distributed renderer, Immediate rendering	Distributed event handling	Structurally invasive	OpenGL, OpenSceneGraph, 3D graphics API	OpenGL high-performance visualization
CGLX [63]	Distributed application	Distributed event handling, Input server	Minor code modification, replace view functions	OpenGL, GLUT, 3D graphics API	OpenGL high-performance visualization, multimedia
Parallel iWalk [11]	Distributed renderer, Immediate rendering	unknown	Minor code modification, transmitting the viewing parameters and culling	iWalk, 3D graphics API	High-performance large-scale 3D visualization
FlowVR Render [40]	Distributed renderer, Retained rendering, Shader-based protocol	unknown	Minor code modification, new functions for Shader framework	OpenGL, FlowVR, 3D graphics API	Virtual reality and scientific visualization
MPIglut [105]	Distributed Application	Distributed event handling	Non-invasive, requires recompiling	OpenGL, 3D graphics API	OpenGL applications

pixel data, which are streamed to the SAGE receivers. The FSManager coordinates the requests and receivers, handling placement and updates. A key feature is that, unlike DMX, SAGE supports applications running on multiple remote hosts. So, for example, three users could all be working on individual machines, exporting their results to the same shared LHRD.

While direct use of SAIL is far from 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. However, the OpenGL wrapper is limited to the size of the frame buffer on the local machine, rather than to the size of the LHRD.

As with DMX, SAGE is appropriate for displaying multiple applications or windows of heterogeneous information, rather than a single large, complex visualization. While the pixel distribution approach is very generalizable, it is not a

high-performance approach, and offers less opportunity to leverage the capabilities of the cluster.

8.1.3 Chromium

Chromium [69], based on the earlier WireGL [111], is a widely used LHRD software framework due to its transparent OpenGL support. Chromium works by replacing the OpenGL shared library, so that it can be used by an OpenGL application without modification. This provides a great deal of flexibility as any application developed in any language or graphics API that links against the OpenGL library can make use of Chromium [112].

In order to execute an OpenGL application on the cluster-based LHRD, Chromium makes use of four main components. The first of these is a configuration server, called the mothership, which manages information about the configuration of the LHRD. The second is a custom application loader (crappfaker), which launches the application and links it to the Chromium library on the head node. The third component is stream processing unit (SPU), which intercepts and processes the stream of OpenGL calls of an application for multiple display tiles and nodes. The most important SPU for LHRD is *tilesort* which performs a sort-first partition of the geometry and passes the OpenGL command stream to the display nodes. The last is the crServer, which runs on the display nodes and handles the rendering with the OpenGL library.

Chromium remains a useful tool for executing OpenGL applications on LHRDs, and for writing mixed environment software. However, performance limitations may make this option insufficient for dynamic, large data visualization on LHRDs. Chromium's performance is significantly affected by the overhead of intercepting rendering commands in the head node and the related network bandwidth requirements due to the large amount of graphics commands and primitives that must be transmitted for every new frame. Another problem with Chromium is that it requires an internal implementation or wrapper for every OpenGL command that it supports. As such, 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).

8.1.4 ClusterGL

ClusterGL [113] is based on Chromium's approach, but includes optimization features to reduce the amount of network traffic, including multicast, frame differencing and data compression of the OpenGL command stream. Their benchmarks show that, for most applications, ClusterGL outperforms Chromium which supports unmodified OpenGL applications. The performance difference increases with more complex scene geometries and more display nodes.

8.2 Distributed Scene Graph Frameworks for 3D Graphics Applications

Scene graphs are used to model and simplify the management of 3D models and scenes [114]. Since scene graphs allow developers to construct complex 3D scenes in logically easy-to-understand ways, they are thus commonly used in applications for 3D virtual environments. Scene graph frameworks lend themselves to the distributed renderer model based on retained rendering. Distributed Scene graphs can improve performance of LHRD applications in two important ways. First, network traffic between the head node and display nodes are significantly reduced because only change lists need to be communicated after the initial scene graph and associated graphics data have been distributed. Second, the display nodes can make better use of the retained rendering features of their local graphics hardware and memory.

8.2.1 OpenSG

OpenSG is a distributed scene graph framework for LHRD clusters and single desktop computers. It manipulates scene graphs with multiple asynchronous threads [115], [90]. Multithreading and clustering support distinguishes OpenSG from a different scene graph framework for single workstations, OpenSceneGraph (OSG). Each display node maintains not only the same copy of a scene graph in its own data format called *FieldContainers* but also the binary content of the *Fields*. 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. OpenSG allows the head node to send user-created classes derived from FieldContainers across the cluster. Also, it enables programmers to filter specific changes on display nodes.

Extremely dynamic visualization may not be efficient for OpenSG due to the overhead in synchronization and update of the distributed scene graph [45]. 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 in developing its applications.

8.2.2 Garuda

Garuda builds on top of the OpenSceneGraph toolkit, enabling users to run legacy OSG applications on cluster-based LHRD without modification [116]. Garuda relies on a server-push and multicast approach to handle the distributed scene graphs and dynamic 3D models across multiple display nodes. Each display node performs view frustum culling using a novel adaptive algorithm [68]. The framework also supports a non-invasive programming interface which automatically replaces OSG's cull, draw, and swap functions.

8.2.3 Blue-c Distributed Scene Graph

Blue-c provides a scene graph interface based on OpenGL Performer for tiled display environments [117]. The framework employs split scene-graph architecture to minimize scene graph synchronization overhead. For example, it divides the scene graph maintained in local display nodes into the shared and local partitions. The synchronization service tracks and synchronizes updates of the scene graph on the shared partition to ensure consistency across nodes.

On the other hand, the local partition is managed and updated only by the local application at each display node. Another distinguishing characteristic of this framework is its ability to manage multimedia directly in the scene graph.

8.2.4 Open Inventor Cluster Edition

This framework emphasizes transparent scalability of existing Open Inventor applications for LHRD through the distributed scene graph [65]. To manage large volume data on each display, the framework supports a middleware called VolumeViz LDM (large data management). The middleware enables each display node to load only the necessary part of large data without duplicating the same data across the nodes. The framework also supports an extension API for rendering extremely large 3D data.

8.2.5 AVANGO NG

AVANGO NG is a distributed scene graph framework [118] and flexible display configurations for different types of LHRDs. AVANGO NG uses a scene-graph data format and distribution approach similar to OpenSG, but is based on both OpenSceneGraph and Python. Programmers can develop an entire application in Python without the need for other programming languages.

8.3 Interactive Application Frameworks

Interactive application frameworks are designed to facilitate developing interactive applications based on novel interaction techniques and modalities as well as specialized display systems such as CAVEs and HMDs. They provide a layer of abstraction over such different types of large display hardware and versatile input devices. Most of these LHRD frameworks provide an integrated environment for advanced input management and configuration, and also provide custom event handlers that collect event data from multiple input devices.

8.3.1 VR Juggler

VR Juggler is based on a virtual platform framework [98], [106]. The virtual platform consists of two main components: the *kernel* and the *manager*, which provide the application with interfaces to the hardware devices and other graphics APIs. Every application acts as an application object in the form of a C++ object. The kernel executes the application objects and controls the run-time system by brokering communications among the managers. The manager provides abstractions of multiple input devices, displays, network, and windowing systems. Programmers access new devices by simply creating a new manager in the API. Developers can extend the system during run time, and can make use of various existing graphics APIs, including OpenGL, OpenSG, OSG, VTK [119], etc. The framework also allows developers to write applications with a set of predefined kernel interface functions and existing graphics APIs.

VR Juggler achieves a consistent and stable frame rate [45], 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 code. This tool gathers and provides important performance data such as rendering times, wait time for buffer swaps, etc. This information can be used to optimize VR Juggler programs and better understand specific LHRD systems.

8.3.2 CaveLib

CaveLib was originally developed as a dedicated API for the specialized CAVE system in the early 1990's [101], [99], and has since been extended to tiled displays as well. The application instance runs as a multithreaded application on each node in which the application tasks are split into several separate threads. This allows adding or removing display nodes at run-time. It provides a parallel graphics API for creating immersive and interactive 3D environment applications on LHRDs and allows users to choose a rendering system such as OpenGL and OpenGL Performer.

8.3.3 Jinx

Jinx [120] is designed for developing and executing distributed VR applications based on X3D [121] on LHRDs. The main goal of JINX is to provide an X3D browser for cluster-based LHRDs, and a programming interface that hides the details of the cluster and user interface. Developers can reuse existing X3D scripts without modification for LHRDs, and can also use some OpenGL functions.

Jinx adaptively determines the optimized configuration to execute the X3D script. For example, each node decides the best way to communicate autonomously with other nodes based on the system configuration. Also, users can choose between MPI and sockets for communications among nodes.

8.3.4 Syzygy

Syzygy [62] is a multi-platform framework for creating 3D VR applications and other graphical applications such as tele-collaboration and multimedia. The framework provides a relatively complex programming interface that requires users to consider low-level issues of the cluster when developing an application.

This framework supports both task distribution models and allows programmers to choose one of the two models. Syzygy's processes and configuration information are managed by a dedicated distributed OS, *Phleet*. The application centralizes and maintains consistent configuration across separate networked nodes in the cluster by managing the configuration information in a single networked database. The framework enables the building of a 'virtual device' which is integrated with multiple physical input devices through an input server supporting input data filtering and gathering.

8.3.5 AmiraVR Cluster Version

The amiraVR cluster version is implemented for cluster-based LHRDs as an extension of amira and amiraVR, which focuses on 3D graphics applications with Open Inventor [122], [123]. Each display node runs an instance of amira with different viewports of the same scene. The framework

focuses on running amiraVR applications with its unique features such as 3D GUIs on the cluster based LHRD.

8.3.6 jBricks

jBricks is the only Java framework for cluster-based LHRDs in this survey [100], [87]. In contrast to many of the frameworks that focus on techniques for parallel rendering of complex 3D graphics models, jBricks focuses on Java-based 2D graphics rendering supported by ZVTM [124]. The framework is useful for developing InfoVis applications supporting a variety of user input devices. By modifying a few lines of the original ZVTM code written for single desktop computers, programmers can develop cluster-based LHRD applications. The framework directly supports a variety of 2D graphics objects such as Java2D, Swing widgets, bitmap images and text, with support for advanced stroke and fill patterns. In addition, developers can use various popular Java libraries which are extensions of ZVTM including, for example, the layout of large networks with JUNG or GraphViz, OpenStreetMap, etc.

jBricks uses a simple distributed renderer model with JGroups multicast communication. For input management, the jBricks Input Server is developed on top of multiple Java-based libraries to support various common and advanced input devices, including tablets, Wii remotes, VICON motion-trackers, interactive pens, TUIO [91], etc. This framework is designed to support rapid testing and prototyping of interaction techniques, 2D interactive visualization and post-WIMP applications for cluster-based LHRDs.

8.3.7 MostPixelsEver Cluster Edition (MPECE)

MostPixelsEver Cluster Edition is a library designed for the *Processing* environment [125]. The framework is designed to run Processing applications on cluster based LHRDs. The use of the library is somewhat invasive, since it requires the developer to include the library in the Processing code and to augment the application with some specialized commands for reading the configuration file and handling communication between the nodes (The actual communication is fairly transparent). However, this does not alter the structure of the code, so we would still label this "minimally invasive."

8.4 Scalable Rendering Frameworks

A repeating theme in our discussions is how performance rapidly degrades as the size of the cluster grows or as the complexity of the scene increases. The frameworks in this section have been designed to specifically address these performance issues for high-performance visualization applications. They emphasize scalable rendering performance with minimal abstraction of the cluster, rather than attempting to make existing graphics APIs work transparently on the LHRD. The frameworks support different features for more precise control over the parallel rendering algorithms and distributed hardware resources for the LHRD.

8.4.1 Equalizer

Equalizer is an OpenGL parallel rendering system [22]. It provides an API to create OpenGL multi-pipe applications for cluster-based LHRDs, superseding OpenGL MultiPipe SDK [107].

It is based on multi-thread processing across cluster nodes. Two different types of threads are used in the head and display nodes respectively. In contrast to Chromium which runs a full application including rendering tasks at the head node, the rendering component of an Equalizer application is executed only in the display nodes and all rendering tasks are performed locally to the OpenGL context on each display node, rather than being driven by the head node. As a result, it is able to reduce computation steps and workload in the head node, allowing it to send higher-level graphics calls thereby reducing network traffic rather than transmitting low-level graphics commands and primitives.

A dedicated configuration server manages the utilization and load balancing of distributed hardware resources in the LHRD cluster. It uses compound trees to configure multiple graphics resources and improves efficiency in decomposing rendering tasks across the cluster nodes.

To implement scalable applications with Equalizer's API, developers define callback functions similar to GLUT and employ sub-classes that present abstractions of physical and logical entities for rendering, such as the display node, GPU, window and view. In addition to the parallel programming interfaces [126], the framework provides users with useful libraries including a network library [127] and a library for multi-threaded programming [128]. Equalizer also supports up-to-date OpenGL advanced features such as VBO, GLSL, and CUDA.

8.4.2 Cross-Platform Cluster Graphics Library (CGLX)

The main purpose of Cross-Platform Cluster Graphics Library is to support high-performance rendering on the LHRD cluster with a nearly non-invasive OpenGL programming model [63]. Each display node maintains independent OpenGL contexts, and the nodes communicate through CGLX's own communication layer.

CGLX intercepts and re-implements some view-related OpenGL functions. Its API is nearly identical to those of OpenGL and GLUT, replacing only prefixes of some of the GL/GLU function names (e.g., *gluPerspective()* function is replaced with *cglXPerspective()*).

The framework is able to support multiple CPUs and multiple displays on a single node through multi-threads. Since it allows users to configure how a different thread performs rendering on each display/window in parallel or in a serial way, users can optimize the visualization performance based on different display and cluster configurations.

CGLX provides a unique GUI tool, *csconfig*, which is connected to each display node. With this tool, users can configure large displays and see a preview of an application, with different configurations, by virtually adjusting various display parameters including the size of bezels, the resolution of displays, and the arrangement of the display array.

The framework can maintain multiple input servers that are divided into two types (Passive and Active) in order to support various types of input devices and multi-touch displays such as tabletops and hand-held displays [129]. This approach facilitates streaming visualization data to different

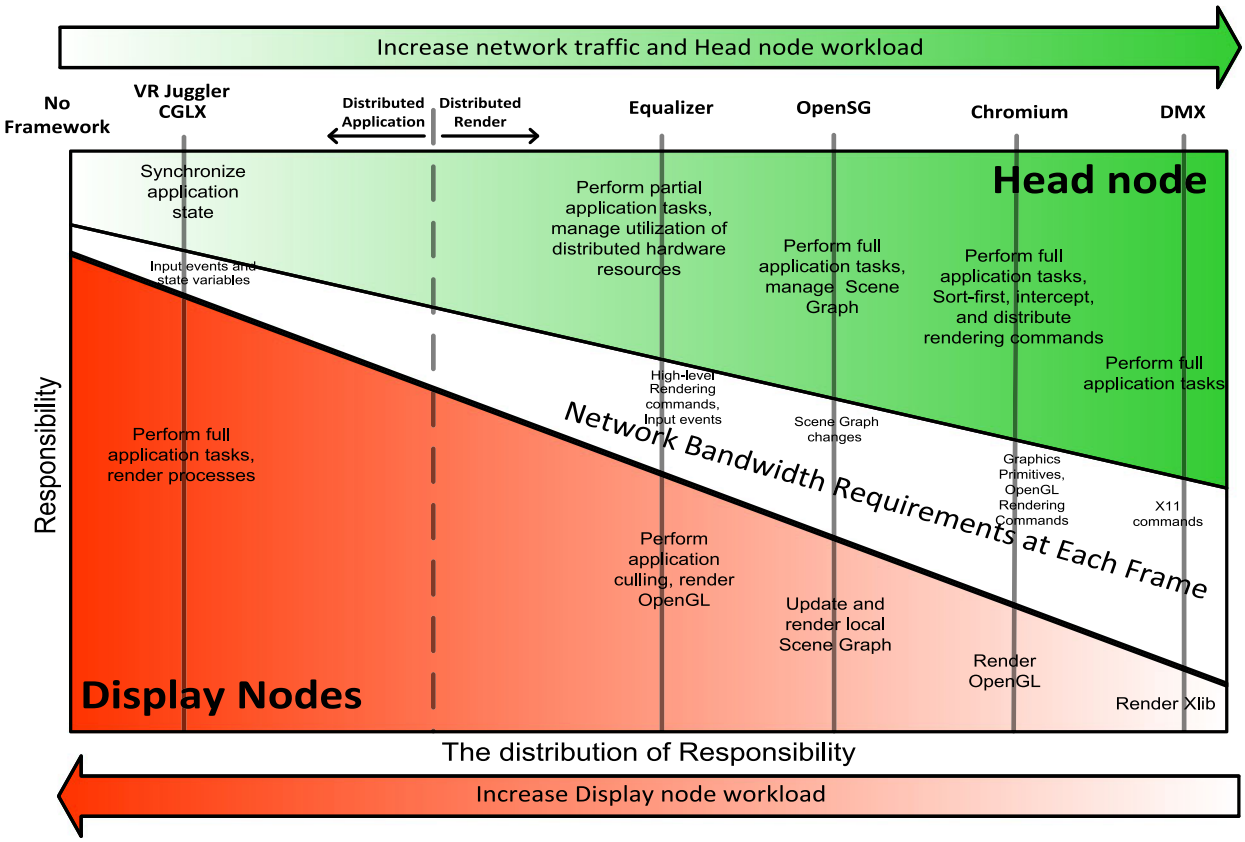


Fig. 5. Spectrum of the distribution of responsibility, between the head node and the display nodes, of a cluster-based LHRD application.

display surfaces and connecting or removing input devices during runtime of an LHRD application [86].

8.4.3 Parallel iWalk

Parallel iWalk [11] is a sort-first based scalable rendering system for tiled displays. It builds on iWalk, a framework for multi-threaded out-of-core rendering. It is designed for visualizing extremely large models on cluster-based LHRDs by combining both sort-first and out-of-core algorithms. The cluster nodes use MPI for startup and synchronization and they use sockets for other communications. The framework can run the same code as iWalk, with a few additional functions such as transmitting the viewing parameters from the head node to the display nodes and culling on each tile.

8.4.4 FlowVR Render

FlowVR is a framework dedicated to large interactive applications [40]. FlowVR Render is built on top of FlowVR and supports a sort-first parallel rendering algorithm for LHRDs. The framework distributes shader information that specifies graphics objects' visual representations, instead of relying solely on distributing OpenGL commands or primitives over a network. Thus, it can reduce cluster network bandwidth and exploits the GPU's shader capabilities.

Depending on different types of LHRD applications, the programmer can choose among different distributed rendering strategies according to how the *Renderer* and *Viewer* modules are distributed among cluster nodes. The Renderer modules accomplish both rendering and displaying on each display tile, while the Viewer modules are responsible for creating geometric objects and distributing the shaders to the Render modules.

8.4.5 MPIglut

MPIglut is designed to run OpenGL/GLUT code on cluster based LHRD using MPI [105]. MPIglut is a dedicated GLUT library specifically for the LHRD cluster, and users need to recompile OpenGL code with the library. Similar to CGLX, MPIglut replaces and re-implements some of the OpenGL functions, including *glutInit()*, and *glViewport()*, and nodes in the LHRD cluster internally communicate through MPI. The head node maintains a frontend which collects the system and user events from an OpenGL application, and the framework broadcasts them through MPI to the display nodes where synchronized OpenGL applications run in parallel.

9 Discussion

In this paper, we surveyed a number of different solutions for distributing graphics across a large tiled display. While we discussed the most popular approaches, there are many more frameworks available. Our goal was to characterize the techniques in a way that could be applied to other frameworks. Rather than providing performance measures for some small number of narrowly defined tasks and select frameworks, we instead examined the frameworks analytically, identifying characteristics that can be used to guide

selection and predict performance. Our experience has shown that the most reliable predictor of performance is the distribution of responsibility, as summarized in Fig. 5.

All of the frameworks that we examined divide work between the head node and the display nodes responsible for driving the displays. The figure helps to capture the general continuum of where the different frameworks do their work. The upper part (green region) of the diagonal is work done at the head node, while below the line (red region) is work done in the display nodes. The thickness of the white line in the diagonal represents the network bandwidth requirements.

This division plays an important role in system performance for two reasons. First, the amount of work done by the display nodes determines the degree of parallelism supported by the framework. Second, where the division occurs affects the network bandwidth requirements. As the head node assumes more responsibility, the quantity of information that must be broadcast increases. This is because the rendering process is one that transforms high-level information into increasingly more detailed structures and commands for displaying representations of that data. The more of this process that is performed by the head node, the more detailed is the information that must be communicated to the display nodes.

Our figure depicts this as a spectrum of responsibility. On the right side, we have solutions such as DMX and Chromium that perform practically all operations on the head node, such as running applications, splitting, packaging, and distributing low-level rendering information.

In the distributed scene graph frameworks like OpenSG, the head node tracks changes to the scene graph and distributes these changes to the display nodes. So, the network usage is reduced in this framework as compared to other distributed renderer models that transmit the rendering calls or primitives. The display nodes then update the local scene graph and generate rendering commands by traversing scene graphs for their associated display tiles, and the computation load on each display node is increased.

In contrast to Chromium which still performs rendering commands (which are intercepted) on the head node, Equalizer clearly splits rendering tasks from the application and runs the rendering tasks only on each display node in parallel with appropriate frustum culling. So it reduces workload to be done at the head node.

Moving to the left of the diagram, the responsibilities of the display nodes increases, until we reach solutions like CGLX and VR Juggler that only communicate events and synchronization information across the network. This requires minimal network usage, but applications on each node need to be deterministic since all nodes must run a full instance of an application.

While there is no framework that occupies the far left edge of the spectrum in our diagram, we can imagine custom solutions that require no head node (and no framework). For example, it would be fairly straightforward to simply run multiple instances of the same application on all nodes, with perhaps a command line option to determine the viewport. If all user interaction performed only local operations (i.e., highlighting or revealing additional information), network communication between the nodes could be eliminated entirely.

It is tempting to read this figure as an absolute guide to performance, with performance increasing to the left. However, the reality is more complicated than that. Different applications have different needs. The complexity of the rendering, the size of the data, the amount of computation required, the rendering scalability, and the degree of dynamics will all be factors determining which solution is the most appropriate.

We are also of the opinion that the programming model required by the framework should not be overlooked. While scene graphs tend to lead to performance benefits for many rendering tasks, they are certainly not appropriate for all applications. For example, a scene graph would be appropriate for the exploration of a highly complex model where most changes to the view are mere transformations of the model, but considerably less appropriate for a force-directed graph layout in which almost no relationship between objects remains fixed. It is also important to consider factors like development time and familiarity. For instance, on a smaller cluster DMX is a perfectly reasonable solution that allows existing applications to be run without modification. Similarly, Chromium's performance may lag many of the other solutions, but it is one of the fastest ways to port existing OpenGL code or maintain applications intended for both large and conventionally sized displays. Since most LHRDs reside in research settings in which development time is very large in relation to running time, the cost of development time cannot be overlooked.

For these reasons, single node architectures are becoming more popular. As graphics hardware capability continues to improve, a single computer with multiple graphics cards can drive many high-resolution display tiles. The advantage is that standard operating systems, windowing systems, and applications can be executed without special LHRD programming APIs. This enables rapid prototyping of new LHRD applications for researching LHRD user interface design and usability [130]. Distribution of graphics across the graphics cards is handled automatically in some cases. The presence of multiple GPUs can enable fast graphics performance, without the need for multiple CPUs. This also opens opportunities for new types of software frameworks designed for single-node multi-GPU systems that support efficient data distribution across the internal bus, load balancing of graphics rendering across GPUs, and efficient management of GPGPU computing for application data processing [131], [132].

10 Conclusion

There is no single approach to tiled rendering that can be considered the best solution for all applications. Our goal with this paper is to survey the available options and to highlight the important dimensions of the development space. It is our hope that this work will guide application developers as they select frameworks to support their large display applications, as well as inform researchers in the development of future large tiled display frameworks.

ACKNOWLEDGMENTS

This research was partially supported by grants from US National Science Foundation (NSF) CNS-1059398, NSF IIS-1218346, and L-3 STRATIS.

REFERENCES

[1] P.J. Huber, "Huge Data Sets," *Proc. Computational Statistics (COMPSTAT)*, pp. 3-13, 1994.

[2] B. Yost, Y. Haciahmetoglu, and C. North, "Beyond Visual Acuity: The Perceptual Scalability of Information Visualizations for Large Displays," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI '07)*, pp. 101-110, 2007.

[3] R. Ball, C. North, and D.A. Bowman, "Move to Improve: Promoting Physical Navigation to Increase User Performance with Large Displays," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI '07)*, pp. 191-200, 2007.

[4] X. Bi and R. Balakrishnan, "Comparing Usage of a Large High-Resolution Display to Single or Dual Desktop Displays for Daily Work," *Proc. 27th Int'l Conf. Human Factors in Computing Systems*, pp. 1005-1014, 2009.

[5] L. Shupp, C. Andrews, M. Dickey-Kurdziolek, B. Yost, and C. North, "Shaping the Display of the Future: The Effects of Display Size and Curvature on User Performance and Insights," *Human-Computer Interaction*, vol. 24, no. 1/2, pp. 230-272, 2009.

[6] C. Andrews, A. Endert, and C. North, "Space to Think: Large High-Resolution Displays for Sensemaking," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI '10)*, pp. 191-200, 2010.

[7] B. Schaeffer, "A Software System for Inexpensive VR via Graphics Clusters," http://isl.uiuc.edu/Publications/dgdpaper.pdf, 2000.

[8] G.W. Pieper, T.A. DeFanti, Q. Liu, M. Katz, and P. Papadopoulos, "Visualizing Science: The OptIPuter Project," *SciDAC Rev.*, vol. spring, pp. 32-41, 2009.

[9] G. Wallace et al., "Tools and Applications for Large-Scale Display Walls," *IEEE Computer Graphics and Applications*, vol. 25, no. 4, pp. 24-33, July/Aug. 2005.

[10] "AMD FirePro™ Professional Graphics," http://www.amd.com/firepro3d, AMD, 2012.

[11] W.T. Corrêa, J.T. Klosowski, and C.T. Silva, "Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays," *Proc. Fourth Eurographics Workshop Parallel Graphics and Visualization*, pp. 89-96, 2002.

[12] B. Raffin and L. Soares, "PC Clusters for Virtual Reality," *Proc. IEEE Virtual Reality Conf. (VR '06)*, pp. 215-222, 2006.

[13] H. Miyachi, H. Shigeta, K. Kiyokawa, H. Kuwano, N. Sakamoto, and K. Koyamada, "Parallelization of Particle Based Volume Rendering on Tiled Display Wall," *Proc. Network-Based Information Systems (NBiS)*, pp. 435-438, 2010.

[14] K. Li et al., "Building and Using a Scalable Display Wall System," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 29-37, July/Aug. 2000.

[15] V. Petrovic, J. Fallon, and F. Kuester, "Visualizing Whole-Brain DTI Tractography with GPU-Based Tuboids and LoD Management," *IEEE Computer Graphics and Applications*, vol. 13, no. 6, pp. 1488-1495, Nov./Dec. 2007.

[16] T. Schreck, T. Tekušová, J. Kohlhammer, and D. Fellner, "Trajectory-Based Visual Analysis of Large Financial Time Series Data," *ACM SIGKDD Explorations*, vol. 9, no. 2, pp. 30-37, 2007.

[17] Y. Sun, J. Leigh, A. Johnson, and S. Lee, "Articulate: A Semi-Automated Model for Translating Natural Language Queries into Meaningful Visualizations," *Proc. 10th Int'l Symp. Smart Graphics*, pp. 184-195, 2010.

[18] Notcot,"Mercedes-Benz Design Studio Powerwall," http://www.notcot.com/archives/2010/04/mercedes-benz-design-studio-po.php, 2010.

[19] T.A. DeFanti et al., "The StarCAVE, a Third-Generation CAVE and Virtual Reality OptIPortal," *Future Generation Computer Systems*, vol. 25, pp. 169-178, 2009.

[20] S.-J. Kim, "The DIVA Architecture and a Global Timestamp-Based Approach for High-Performance Visualization on Large Display Walls and Realization of High Quality-of-Service Collaboration Environments," PhD dissertationElectrical and Computer Engineering, UC Irvine, 2006.

[21] S. Nam, S. Deshpande, V. Vishwanath, B. Jeong, L. Renambot, and J. Leigh, "Multi-Application Inter-Tile Synchronization on Ultra-High-Resolution Display Walls," *Proc. ACM SIGMM Conf. Multimedia Systems*, pp. 145-156, 2010.

[22] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A Scalable Parallel Rendering Framework," *IEEE Trans. Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436-452, May/June 2008.

[23] T. Ni, 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," *Proc. IEEE Virtual Reality Conf. (VR '06)*, pp. 223-236, 2006.

[24] A. Forsberg, J. Head, N. Petro, and G. Morgan, "A 3D Geoscience Data Visualization System for Mars Applied to Undergraduate Laboratories," *Proc. 38th Lunar and Planetary Inst. Science Conf.*, p. 1297, 2007.

[25] W. Buxton, G. Fitzmaurice, R. Balakrishnan, and G. Kurtenbach, "Large Displays in Automotive Design," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 68-75, July/Aug. 2000.

[26] Renault, "Renault Technocentre Celebrates 10th Anniversary," http://www.carbodydesign.com/archive/2008/06/17-renault-technocentre-celebrates-10th-anniversary, 2008.

[27] N. Schwarz, "Distributed Volume Rendering of Very Large Data on High-Resolution Scalable Displays," PhD Dissertation, Univ. of Illinois at Chicago, 2007.

[28] A.L. Spitzer, "HIPerWall Expands 3D Capabilities with New Software," http://www.calit2.net/newsroom/article.php?id = 1406, 2011.

[29] T.v.d. Schaaf, M. Koutek, and H. Bal, "Parallel Particle Rendering: a Performance Comparison between Chromium and Aura," *Proc. Eurographics Symp. Parallel Graphics and Visualization*, pp. 137-144, 2006.

[30] A. Forsberg, Prabhat, G. Haley, A. Bragdon, J. Levy, C. Fassett, D. Shean, J. Head III, S. Milkovich, and M Duchaineau, "Adviser: Immersive Field Work for Planetary Geoscientists," *IEEE Computer Graphics and Applications*, vol. 26, no. 4, pp. 46-54, July/Aug. 2006.

[31] A. Johnson, J. Leigh, P. Morin, and P. Van Keken, "GeoWall: Stereoscopic Visualization for Geoscience Research and Education," *IEEE Computer Graphics and Applications*, vol. 26, no. 6, pp. 10-14, Nov./Dec. 2006.

[32] A. Kolb, M. Lambers, S. Todt, N. Cuntz, and C. Rezk-Salama, "Immersive Rear Projection on Curved Screens," *Proc. IEEE Virtual Reality Conf. (VR '09)*, pp. 285-286, 2009.

[33] R. Wilhelmson, P. Baker, R. Stein, and R. Heiland, "Large Tiled Display Walls and Applications in Meteorology, Oceanography, and Hydrology," *Proc. 18th Int'l Conf. Interactive Information and Processing Systems (IIPS '02)*, pp. 29-30, 2002.

[34] I. James et al., "ADVISER: Immersive Scientific Visualization Applied to Mars Research and Exploration," *Photogrammetric Eng. & Remote Sensing*, vol. 17, pp. 1219-1225, 2005.

[35] ORNL, "VISUALIZATION: New Ways to Understand the Data," *ORNL REV.*, vol. 37, pp. 28-29, 2004.

[36] N. Schwarz and J. Leigh, "Distributed Volume Rendering for Scalable High-Resolution Display Arrays," *Proc. Fifth Int'l Conf. Computer Graphics Theory and Applications (GRAPP '10)*, pp. 211-218, 2010.

[37] T.A. Sandstrom, C. Henze, and C. Levit, "The Hyperwall," *Proc. IEEE Coordinated and Multiple Views in Exploratory Visualization*, pp. 124-133, 2003.

[38] J. Leigh et al., "An Experimental OptIPuter Architecture for Data-Intensive Collaborative Visualization," *Proc. Third Ann. Workshop Advanced Collaborative Environments*, 2003.

[39] N. Schwarz et al., "Vol-a-Tile - A Tool for Interactive Exploration of Large Volumetric Data on Scalable Tiled Displays," *Proc. IEEE Visualization (VIS '04)*, pp. 598-19, 2004.

[40] J. Allard and B. Raffin, "A Shader-Based Parallel Rendering Framework," *Proc. IEEE Visualization (VIS '05)*, pp. 127-134, 2005.

[41] F. Erol, S. Eilemann, and R. Pajarola, "Cross-Segment Load Balancing in Parallel Rendering," *Proc. Eurographics Symp. Parallel Graphics and Visualization*, pp. 41-50, 2011.

[42] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J.P. Singh, "Load Balancing for Multi-Projector Rendering Systems," *Proc. SIGGRAPH /Eurographics Workshop Graphics Hardware*, pp. 107-116, 1999.

[43] C.c. Zhang, J. Leigh, T.A. Defanti, M. Mazzucco, and R. Grossman, "TeraScope: Distributed Visual Data Mining of Terascale Data Sets over Photonic Networks," *Future Generation Computer Systems*, vol. 19, pp. 935-943, 2003.

[44] C. Andrews, A. Endert, B. Yost, and C. North, "Information Visualization on Large, High-Resolution Displays: Issues, Challenges, and Opportunities," *Information Visualization*, vol. 10, pp. 341-355, 2011.

[45] O. Staadt, J. Walker, C. Nuber, and B. Hamann, "A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering," *Proc. Eurographics Workshop Virtual Environments*, pp. 261-270, 2003.

[46] E. Shaffer, D.A. Reed, S. Whitmore, and B. Schaeffer, "Virtue: Performance Visualization of Parallel and Distributed Applications," *Computer*, vol. 32, no. 12, pp. 44-51, Dec. 1999.

[47] R.T. Stevens, "Testing the NORAD Command and Control System," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 1, pp. 47-51, Mar. 1968.

[48] B. Wei, C. Silva, E. Koutsofios, S. Krishnan, and S. North, "Visualization Research with Large Displays," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 50-54, July/Aug. 2000.

[49] AT&T, "Global Network Operations Center," http://www.att.com/, 2013.

[50] K. Carter, "MCC Undergoes a Makeover - Jefferson Lab's Accelerator Control Room Gets a New Face," http://www.sura.org/news/2004/docs/jlab_ot.pdf, 2004.

[51] R.C. Johnson, "Visualization Analytics Writ Large," http://www.smartertechnology.com/c/a/Business%20Analytics/Visualization-Analytics-Writ-Large, 2011.

[52] N.K. Krishnaprasad et al., "JuxtaView - A Tool for Interactive Visualization of Large Imagery on Scalable Tiled Displays," *Proc. IEEE Cluster Computing*, pp. 411-420, 2004.

[53] P. Adams, "Hubble Images on TACC Tiled Display," http://www.vizworld.com/2010/01/tacc-tiled-display, 2010.

[54] R. Singh et al., "Real-Time Multi-Scale Brain Data Acquisition, Assembly, and Analysis Using an End-to-End OptIPuter," *Future Generation Computer Systems*, vol. 22, pp. 1032-1039, 2006.

[55] S. Yamaoka, K.-U. Doerr, and F. Kuester, "Visualization of High-Resolution Image Collections on Large Tiled Display Walls27,"*Future Generation Computer Systems*, vol. 279, pp. 498-505, 2011.

[56] D. Svistula, J. Leigh, A. Johnson, and P. Morin, "MagicCarpet: A High-Resolution Image Viewer for Tiled Displays," http://www.evl.uic.edu/cavern/sage/applications.php, 2008.

[57] K. Ponto, K. Doerr, and F. Kuester, "Giga-stack: A Method for Visualizing Giga-Pixel Layered Imagery on Massively Tiled Displays," *Future Generation Computer Systems*, vol. 26, pp. 693-700, 2010.

[58] H. Chen, G. Wallace, A. Gupta, K. Li, T. Funkhouser, and P. Cook, "Experiences with Scalability of Display Walls," *Proc. Immersive Projection Technology (IPT) Workshop*, 2002.

[59] Calit2,"Streamer," http://hiperwall.calit2.uci.edu/?q=node/6, 2007.

[60] D. Kosovic, "MacOSX-Based Video Streamer," http://www.evl.uic.edu/cavern/sage/download.php, 2009.

[61] B. Jeong et al., "High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environment," *Proc. ACM/IEEE SuperComputing Conf.*, pp. 24-24, 2006.

[62] B. Schaeffer and C. Goudeseune, "Syzygy: Native PC Cluster VR," *Proc. IEEE Virtual Reality (VR '03)*, pp. 15-22, 2003.

[63] K. Doerr and F. Kuester, "CGLX: A Scalable, High-Performance Visualization Framework for Networked Display Environments," *IEEE Trans. Visualization and Computer Graphics*, vol. 99, no. 3, Mar. 2011.

[64] H. Chen, D.W. Clark, Z. Liu, G. Wallace, K. Li, and Y. Chen, "Software Environments For Cluster-Based Display Systems," *Proc. First Int'l Symp. Cluster Computing and the Grid*, pp. 202-210, 2001.

[65] D. Lichau, M. Heck, and T. Dufour, "Open Inventor and VolumeViz LDM Cluster Edition," http://www.vsg3d.com/open-inventor/scale-viz, 2005.

[66] G. Humphreys and P. Hanrahan, "A Distributed Graphics System for Large Tiled Displays," *Proc. IEEE Visualization (VIS '99)*, pp. 215-223, 1999.

[67] R. Samanta, T. Funkhouser, and K. Li, "Parallel Rendering with k-Way Replication," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pp. 75-84, 2001.

[68] Nirnimesh, P Harish, and P.J., "Narayanan, Culling an Object Hierarchy to a Frustum Hierarchy," *Proc. Fifth Indian Conf. Computer Vision, Graphics, and Image Processing*, pp. 252-263, 2006.

[69] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J.T. Klosowski, "Chromium: A Stream Processing Framework for Interactive Rendering on Clusters," *Proc. ACM SIGGRAPH*, pp. 693-712, 2002.

[70] S. Thibault, X. Cavin, O. Festor, and E. Fleury, "Unreliable Transport Protocol for Commodity-Based OpenGL Distributed Visualization," *Proc. Workshop Commodity-Based Visualization Clusters*, 2002.

[71] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland, "Scalable Rendering on PC Clusters," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, pp. 62-70, July/Aug. 2001.

[72] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 23-32, 1994.

[73] X. Cavin, C. Mion, and A. Filbois, "Cots Cluster-Based Sort-Last Rendering: Performance Evaluation and Pipelined Implementation," *Proc. IEEE Visualization (VIS '05)*, pp. 111-118, 2005.

[74] K. Moreland, B. Wylie, and C. Pavlakos, "Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pp. 85-92, 2001.

[75] R. Samanta, T. Funkhouser, K. Li, and J.P. Singh, "Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs," *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware*, pp. 97-108, 2000.

[76] T.v.d. Schaaf, L. Renambot, D. Germans, H. Spoelder, and H. Bal, "Retained Mode Parallel Rendering for Scalable Tiled Displays," *Proc. Immersive Projection Technology Workshop*, 2002.

[77] M. Nancel, J. Wagner, E. Pietriga, O. Chapuis, and W. Mackay, "Mid-Air Pan-and-Zoom on Wall-Sized Displays," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI '11)*, pp. 177-186, 2011.

[78] Y. Jansen, P. Dragicevic, and J.-D. Fekete, "Tangible Remote Controllers for Wall-Size Displays," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI '12)*, pp. 2865-2874, 2012.

[79] G. Shoemaker, T. Tsukitani, Y. Kitamura, and K.S. Booth, "Body-Centric Interaction Techniques for Very Large Wall Displays," *Proc. Sixth Nordic Conf. Human-Computer Interaction: Extending Boundaries (CHI '10)*, pp. 463-472, 2010.

[80] C. Rooney and R.A. Ruddle, "A New Method for Interacting with Multi-Window Applications on Large, High Resolution Displays," *Proc. Theory and Practice of Computer Graphics*, pp. 75-82, 2008.

[81] A. Endert, P. Fiaux, H. Chung, M. Stewart, C. Andrews, and C. North, "ChairMouse: Leveraging Natural Chair Rotation for Cursor Navigation on Large, High-Resolution Displays," *Proc. Ann. Conf. Extended Abstracts on Human Factors in Computing Systems*, pp. 571-580, 2011.

[82] A. Khan, G. Fitzmaurice, D. Almeida, N. Burtnyk, and G. Kurtenbach, "A Remote Control Interface for Large Displays," *Proc. ACM 17th Ann. Symp. User Interface Software and Technology (UIST '04)*, pp. 127-136, 2004.

[83] F. Guimbretière, M. Stone, and T. Winograd, "Fluid Interaction with High-Resolution Wall-Size Displays," *Proc. ACM 14th Ann. Symp. User Interface Software and Technology (UIST '01)*, pp. 21-30, 2001.

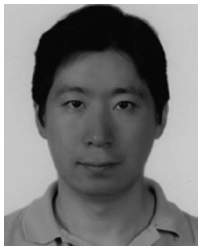
[84] D. Machaj, C. Andrews, and C. North, "Co-located Many-Player Gaming on Large High-Resolution Displays," *Proc. Int'l Conf. Computational. Science and Eng.*, vol. 4, pp. 697-704, 2009.

[85] D. Stødle, T.-M.S. Hagen, J.M. Bjørndalen, and O.J. Anshus, "Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays," *J. Virtual Reality and Broadcasting*, vol. 5, pp. 1860-2037, 2008.

[86] K. Ponto, K. Doerr, T. Wypych, J. Kooker, and F. Kuester, "CGLXTouch: A Multi-User Multi-Touch Approach for Ultra-High-Resolution Collaborative Workspaces," *Future Generation Computer Systems*, vol. 27, pp. 649-656, 2011.

[87] M. Beaudouin-Lafon, S. Huot, M. Nancel, W. Mackay, E. Pietriga, R. Primet, J. Wagner, O. Chapuis, C. Pillias, J. Eagan, T. Gjerlufsen, and C. Klokmose, "Multisurface Interaction in the WILD Room," *Computer*, vol. 45, no. 4, pp. 48-56, Apr. 2012.

[88] T.K. Khan, A. Middel, I. Scheler, and H. Hagen, "A Survey of Interaction Techniques and Devices for Large High Resolution Displays," *Open Access Series in Informatics*, vol. 19, pp. 27-35, 2011.
[89] T. Bierz, "Interaction Technologies for Large Displays-An Overview,"*GI-Edition Lecture Notes in Informatics*, pp. 195-204, 2006.
[90] D. Reiners, G. Voß, and J. Behr, "OpenSG: Basic Concepts," *Proc. OpenSG Symp.*, 2002.
[91] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza, "TUIO: A Protocol for Table-Top Tangible User Interfaces," *Proc. Sixth Int'l Workshop Gesture in Human-Computer Interaction and Simulation*, 2005.
[92] P. Dragicevic and J.-D. Fekete, "Support for Input Adaptability in the ICON Toolkit," *Proc. Sixth Int'l Conf. Multimodal Interfaces*, pp. 212-219, 2004.
[93] W.A. König, R. Rädle, and H. Reiterer, "Squidy: A Zoomable Design Environment for Natural User Interfaces," *Proc. Extended Abstracts Human Factors in Computing Systems (CHI '09)*, pp. 4561-4566, 2009.
[94] I. Russell, C. Hudson, A. Seeger, H. Weber, J. Juliano, and A.T. Helser, "VRPN: A Device-Independent, Network-Transparent VR Peripheral System," *Proc. ACM Symp. Virtual Reality Software and Technology (VRST '01)*, pp. 55-61, 2001.
[95] G. Reitmayr and D. Schmalstieg, "Opentracker-An Open Software Architecture for Reconfigurable Tracking Based on XML," *Proc. IEEE Virtual Reality Conf. (VR '01)*, pp. 285-286, 2001.
[96] M. Virbel, T. Hansen, and O. Lobunets, "Kivy-A Framework for Rapid Creation of Innovative User Interfaces," *Proc. Mensch & Computer Workshop*, pp. 69-73, 2011.
[97] M. Wright, A. Freed, and A. Momeni, "OpenSound Control: State of the Art 2003," *Proc. Conf. New Interfaces for Musical Expression*, pp. 153-160, 2003.
[98] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A Virtual Platform for Virtual Reality Application Development," *Proc. ACM SIGGRAPH ASIA*, pp. 89-96, 2008.
[99] C. Cruz-Neira, D.J. Sandin, and T.A. DeFanti, "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE," *Proc. Conf. Computer Graphics and Interactive Techniques*, pp. 135-142, 1993.
[100] E. Pietriga, S. Huot, M. Nancel, and R. Primet, "Rapid Development of User Interfaces on Cluster-Driven Wall Displays with jBricks," *Proc. ACM Third SIGCHI Symp. Eng. Interactive Computing Systems (EICS '11)*, pp. 185-190, 2011.
[101] M. Szymanski, "CAVELib Support for PC Visualization Clusters," *Advanced Imaging*, vol. 19, pp. 39-44, 2004.
[102] A. Mohr and M. Gleicher, "Non-Invasive, Interactive, Stylized Rendering," *Proc. Symp. Interactive 3D Graphics*, pp. 175-178, 2001.
[103] C. Niederauer, M. Houston, M. Agrawala, and G. Humphreys, "Non-Invasive Interactive Visualization of Dynamic Architectural Environments," *Proc. Symp. Interactive 3D Graphics*, pp. 55-58, 2003.
[104] H. Igehy, G. Stoll, and P. Hanrahan, "The Design of a Parallel Graphics Interface," *Proc. Conf. Computer Graphics and Interactive Techniques*, pp. 141-150, 1998.
[105] O.S. Lawlor, M. Page, and J. Genetti, "MPIglut: Powerwall Programming Made Easier," *J. WSCG*, vol. 16, pp. 130-137, 2008.
[106] J. Allard, V. Gouranton, E. Melin, and B. Raffin, "Parallelizing Pre-Rendering Computations on a Net Juggler PC Cluster," *Proc. Symp. Immersive Projection Technology*, 2002.
[107] P. Bhaniramka, P.C.D. Robert, and S. Eliemann, "OpenGL Multipipe SDK: A Toolkit for Scalable Parallel Rendering," *Proc. IEEE Visualization (VIS '05)*, pp. 119-126, 2005.
[108] K.E. Martin, D.H. Dawes, and R.E. Faith, "Distributed Multihead X Project," http://dmx.sourceforge.net, 2003.
[109] M.T. Asmus, "Xinerama," http://sourceforge.net/projects/xinerama, 2001.
[110] L. Renambot et al., "Sage: The Scalable Adaptive Graphics Environment," *Proc. Fourth Workshop Advanced Collaborative Environments (WACE)*, vol. 9, no. 23, pp. 2004-2009, 2004.
[111] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, "WireGL: a Scalable Graphics System for Clusters," *Proc. Int'l Conf. Computer Graphics and Interactive Techniques*, pp. 129-140, 2001.
[112] K. Moreland and D. Thompson, "From Cluster to Wall with VTK," *Proc. IEEE Symp. (PVG '03)*, pp. 25-31, 2003.
[113] B. Neal, P. Hunkin, and A. McGregor, "Distributed OpenGL Rendering in Network bandwidth Constrained Environments," *Proc. Eurographics Symp. Parallel Graphics and Visualization*, 2011.
[114] H. Sowizral, "Scene Graphs in the New Millennium," *IEEE Computer Graphics and Applications*, vol. 20, no. 1, pp. 56-57, Jan./Feb. 2000.
[115] M. Roth, G. Voss, and D. Reiners, "Multi-Threading and Clustering for Scene Graph Systems," *Computer & Graphics*, vol. 28, pp. 63-66, 2003.
[116] Nirnimesh, P Harish, and P.J. Narayanan, "Garuda: A Scalable Tiled Display Wall Using Commodity PCs," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 5, pp. 864-877, Sept./Oct. 2007.
[117] M. Naef, E. Lamboray, O. Staadt, and M. Gross, "The Blue-c Distributed Scene Graph," *Proc. IEEE Virtual Reality Conf. (VR '03)*, pp. 275-276, 2003.
[118] R. Kuck, J. Wind, K. Riege, and M. Bogen, "Improving the AVANGO VR/AR framework: Lessons learned," *Proc. Workshop Virtuelle und Erweiterte Realität*, pp. 209-220, 2008.
[119] W.J. Schroeder, L.S. Avila, and W. Hoffman, "Visualizing with VTK: A Tutorial," *IEEE Computer Graphics & Applications*, vol. 20, no. 5, pp. 20-27, Sept./Oct. 2000.
[120] L.P. Soares and M.K. Zuffo, "JINX: An X3D browser for VR Immersive Simulation Based on Clusters of Commodity Computers," *Proc. 3D Web Technology*, pp. 79-86, 2004.
[121] W.D. Consortium, "X3D," http://www.web3d.org/x3d, 2005.
[122] TGS and VRCO, "The amiraVR Cluster Version," ftp://ftp.tuebingen.mpg.de/pub/kyb/bweber/zib/share/doc/hxtracking/AmiraVR-cluster.html, 2003.
[123] C.D. Hansen and C.R. Johnson, "Visualization Handbook," . Academic Press, 2004.
[124] E. Pietriga, "A Toolkit for Addressing HCI Issues in Visual Language Environments," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 145-152, 2005.
[125] B. Westing, H. Nieto, R. Turknett, and K. Gaither, "MostPixelsEverCE: A Tool for Rapid Development with Distributed Displays," *Proc. SIGCHI Conf. Human Factors in Computing Systems Extended Abstracts (CHI '13)*, 2013.
[126] S. Eilemann, "Sequel: Parallel Programming Interface," http://www.equalizergraphics.com/api.html, 2012.
[127] S. Eilemann, "Collage: A Network Library for Building Heterogenous, Distributed Applications," http://www.libcollage.net, 2012.
[128] S. Eilemann, "Lunchbox: A Library for Multi-Threaded Programming," https://github.com/Eyescale/Lunchbox, 2012.
[129] K. Doerr, "Standard Server Types," http://www.hiperworks.com/pirdoc/cglx-doc/pirNet/serv_std_types_p.html, 2010.
[130] C. Andrews and C. North, "Analyst's Workspace: An Embodied Sensemaking Environment for Large, High Resolution Displays," *Proc. IEEE Conf. Visual Analytics Science and Technology (VAST '12)*, pp. 123-131, 2012.
[131] C. Peng, P. Mi, and Y. Cao, "Load Balanced Parallel GPU Out-of-Core for Continuous LOD Model Visualization," *Proc. Ultrascale Visualization Workshop*, 2012.
[132] R. Hagan and Y. Cao, "Load Balanced Parallel GPU Out-of-Core for Continuous LOD Model Visualization," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, 2011.



Haeyong Chung received the MS degree in computer and systems engineering at Rensselaer Polytechnic Institute and is working toward the PhD degree in computer science at Virginia Tech. His current research interests focus on visual analytics and information visualization on large high-resolution displays and display ecologies.

Christopher Andrews is a visiting assistant professor of computer science at Middlebury College. His research focuses primarily on improving communication between human and computer by leveraging human cognitive and perceptual abilities. His particular interests include visual analytics, digital art, and CS education.

Chris North is an associate professor of computer science at Virginia Tech. His recent focus is in visual analytics, information visualization, HCI, large displays, and evaluation methods.