University of Zurich[UZH]

Department of Informatics

Modern Distributed Rendering Systems

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zürich

for the degree of
Doctor of Science (Ph. D.)

by
Stefan Eilemann

Accepted on the recommendation of

Prof. Dr. Renato Pajarola
Prof. Dr. ABC DEF

March 22, 2018

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, March 22, 2018

The head of the Ph. D. program in Informatics: Prof. Abraham Bernstein, Ph.D.

ABSTRACT

We are living in the big data age: An ever increasing amount of data is being produced by users through data acquisition and simulations. While large scale analysis and simulatons have received significant attention for cloud computing and HPC systems, software to efficiently visualize large amounts of data is struggling to keep up.

We propose to research system software to facilitate and accelerate large data visualization through parallel rendering, and to validate the research and development of this system software by the development of new applications for large data visualization.

This research and development will enable domain scientists and large data engineers to better extract meaning from their data, making it feasible to explore more data by accelerating the rendering and allowing the use of high-resolution displays to see more detail.

Due to the nature of this research, we propose an engineering-driven, iterative research process. Based on the foundations of a generic parallel rendering system, individual research questions can be addressed in isolation and optimized through data-driven benchmarking, and integrated in product quality into the parallel rendering system.

KURZFASSUNG

Wir leben im groen Datenzeitalter: Immer mehr Datenmengen werden in den letzten Jahren die von den Anwendern durch Datenerfassung und Simulationen erzeugt werden. Whrend in groem Mastab Analyse und Simulationen haben groe Aufmerksamkeit fr Cloud Computing erhalten. und HPC-Systemen, Software zur effizienten Visualisierung groer Datenmengen ist die kmpfen, um Schritt zu halten.

Wir schlagen vor, Systemsoftware zu erforschen, um groe Datenmengen zu vereinfachen und zu beschleunigen. Visualisierung durch paralleles Rendering, und zur Validierung der Forschung und Entwicklung. Entwicklung dieser Systemsoftware durch die Entwicklung von neuen Anwendungen fr groe Datenvisualisierung.

Diese Forschung und Entwicklung wird es den Wissenschaftlern ermglichen, die sich mit der Erforschung von Domnen und groen Datenmengen beschftigen. Ingenieuren, um die Bedeutung ihrer Daten besser zu extrahieren. mehr Daten zu erforschen, indem Sie das Rendering beschleunigen und die Verwendung von hochauflsende Displays, um mehr Details zu sehen.

Aufgrund des Charakters dieser Forschung schlagen wir eine ingenieurgetriebene, iterative Forschungsprozesses. Basierend auf den Grundlagen eines generischen parallelen Renderings System knnen individuelle Forschungsfragen isoliert bearbeitet werden und optimiert durch datengetriebenes Benchmarking und integriert in die Produktqualitt in das parallele Rendering-System.

ACKNOWLEDGMENTS

The research leading to this proposal was supported in part by the Blue Brain Project, the Swiss National Science Foundation under Grant 200020-129525, the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 604102 (Human Brain Project), the Hasler Stiftung grant (project number 12097), and the King Abdullah University of Science and Technology (KAUST) through the KAUST-EPFL alliance for Neuro-Inspired High Performance Computing.

I would like to take the opportunity to thank the Blue Brain Project, in particular the visualization team, and all the other contributors for their support in the research leading to this proposal. I would like to thank Prof. Renato Pajarola for his long-term commitment to my research work and Patrick Bouchaud for putting me onto the path taken by this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

BACKGROUND

1.1 Motivation

After decades of exponential growth in computational performance, storage and data acquisition, computing is now well in the big data age, where future advances are measured in our capability to extract meaningful information from the available data. Visual analysis based on interactive rendering of three-dimensional data has been proven to be a particularly efficient approach to gain intuitive insight into the spatial structure and relations of very large 3D data sets. These developments create new, unique challenges for applications and system software to enable users to fully exploit the available resources to gain insight from their data.

The quantity of computed, measured or collected data is exponentially growing, fueled by the pervasive diffusion of digitalization in modern life. Moreover, the fields of science, engineering and technology are increasingly defined by a data driven approach to conduct research and development. High-quality and large-scale data is continuously generated at a growing rate from sensor and scanning systems, as well as from data collections and numerical simulations in a number of science and technology domains.

Display technology has made significant progress in the last decade. High-resolution screens and tiled display walls are now affordabe for most organizations and are getting deployed at an increasing rate. This increased resolution and display size helps with understanding the data, but with the quadratic increase in pixels to be rendered, it increases the pressure on rendering algorithms to deliver

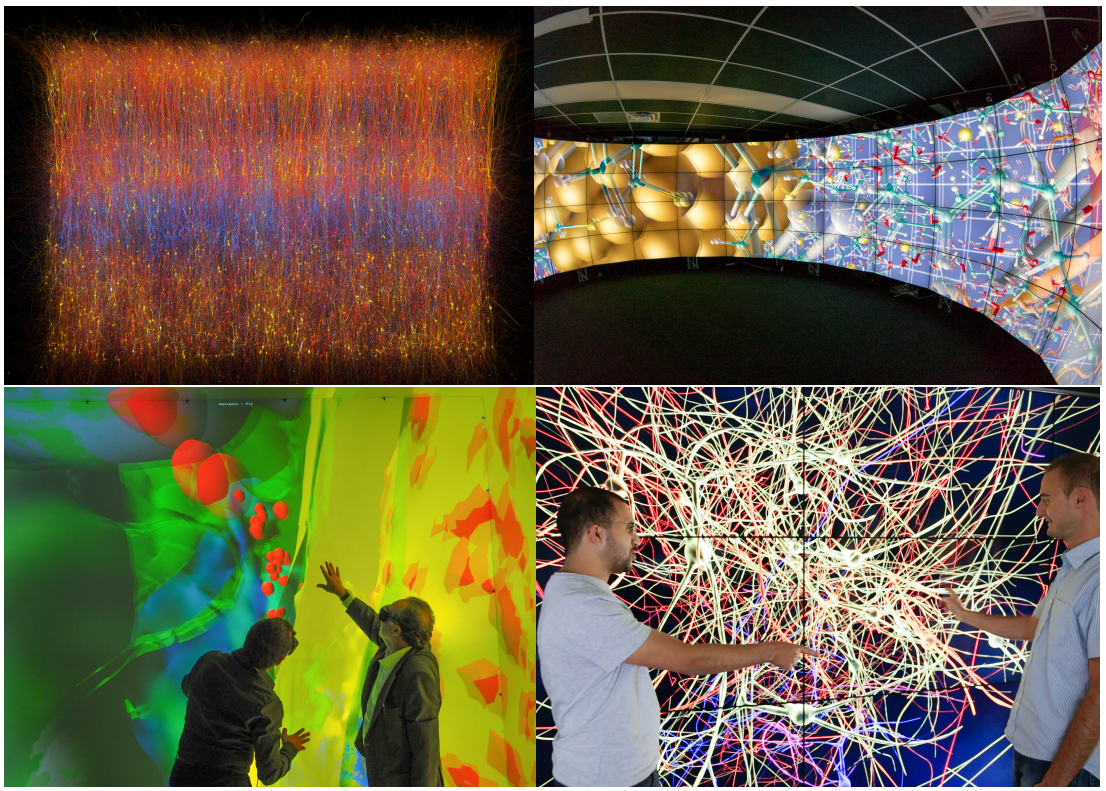

Figure 1.1: *Large Data Visualization: Large data visualization of a brain simulation, molecular visualization in the Cave*[2]*, exploration of EM stack reconstructions in a Cave, collaborative data analysis on a tiled display wall.*

interactive framerates. Furthermore, for larger system it becomes necessary to develop parallel and distributed applications.

However, not only applications are becoming more and more data-driven, but also the technology used to tackle these kinds of problems is rapidly witnessing a paradigm shift towards massively parallel on-chip and distributed parallel cluster solutions. On one hand, parallelism within a system has increased massively, with tenths of CPU cores, thousands of GPU cores and multiple CPUs and GPUs in a single system. On the other hand, massively parallel distributed systems are easily accessible from various cloud infrastructure providers, and are also affordable for on-site hosting for many organizations.

System software to exploit the available hardware parallelism capable of performing efficient interactive data exploration has not kept up with the pace in hardware developments and data gathering capabilities. On one hand, this is due to an inherent delay between hardware and software capabilities, since develop-

ment typically only starts once the hardware is available. On the other hand, existing software engineered for different design parameters has a significant inertia to change, to the extreme of the necessity to rewrite it from scratch.

In the context of emerging data-intensive knowledge discovery and data analysis, efficient interactive data exploration methodologies have become increasingly important. Visual analysis by means of interactive visualization and inspection of three-dimensional data is a particularly efficient approach to gain intuitive insight into the spatial structure and relations of very large 3D data sets. However, defining visual and interactive methods scalable with problem size and degree of parallelism, as well as generic applicability of high-performance interactive visualization methods and systems are recognized among the major current and future challenges.

1.2 Interactive Visualization

1.3 Parallel Rendering

The main performance indicator for Large Data Interactive Rendering is the performance of the rendering algorithm, that is, the framerate with which the program produces new images. This framerate can be improved by either using faster or more hardware, or by better algorithms exploiting the existing hardware and data. This proposal primarily focuses on the first approach using parallel rendering to exploit the CPU and GPU parallelism available on a single system or a distributed cluster. The early fundamental concepts have been laid down in [Molnar et al., 1994] and [Crockett, 1997]. A number of domain specific parallel rendering algorithms and special-purpose hardware solutions have been proposed in the past, however, only few generic parallel rendering frameworks have been developed (Figure 1.2). We will focus on sort-last and sort-first rendering, since sort-middle architectures are only feasible in a hardware implementation due to the large amount of fragments processed and transferred in the sorting stage.

1.3.1 Domain specific solutions

Cluster-based parallel rendering has been commercialized for off-line rendering (i.e. distributed ray-tracing) for computer generated animated movies or special effects, since the ray-tracing technique is inherently amenable to parallelization for off-line processing. Other special-purpose solutions exist for parallel rendering in specific application domains such as volume rendering [Li et al., 1997; Wittenbrink, 1998; Huang et al., 2000; Schulze and Lang, 2002; Garcia and Shen, 2002; Nie et al., 2005] or geo-visualization [Vezina and Robertson, 1991; Agranov

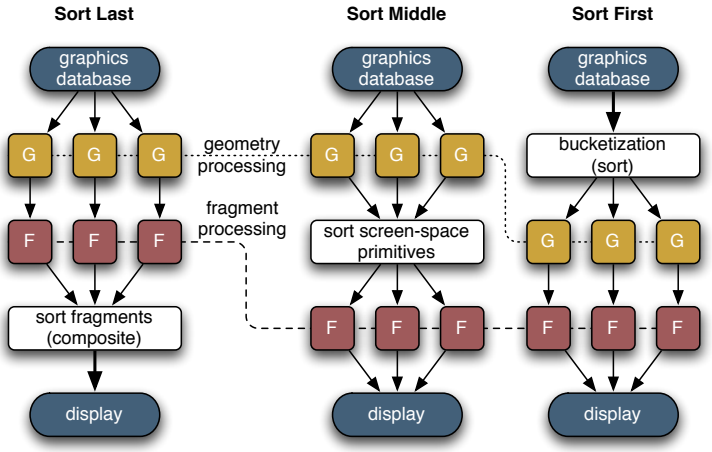

Figure 1.2: *Sort-last, sort-middle and sort-first parallel rendering*

and Gotsman, 1995; Li et al., 1996; Johnson et al., 2006]. However, such specific solutions are typically not applicable as a generic parallel rendering paradigm and do not translate to arbitrary scientific visualization and distributed graphics problems.

In [Niski and Cohen, 2007], parallel rendering of hierarchical level-of-detail (LOD) data has been addressed and a solution specific to sort-first tile-based parallel rendering has been presented. While the presented approach is not a generic parallel rendering system, basic concepts presented in [Niski and Cohen, 2007] such as load management and adaptive LOD data traversal can be carried over to other sort-first parallel rendering solutions.

1.3.2 Special-purpose architectures

Historically, high-performance real-time rendering systems have relied on an integrated proprietary system architecture, such as the early SGI graphics super computers. These special-purpose solutions have become a niche product as their graphics performance does not keep up with off-the-shelf workstation graphics hardware and scalability of clusters.

Due to its conceptual simplicity, a number of special-purpose image compositing hardware solutions for sort-last parallel rendering have been developed. The proposed hardware architectures include Sepia [Moll et al., 1999; Lever, 2004], Sepia 2 [Lombeyda et al., 2001a; Lombeyda et al., 2001b], Lightning 2 [Stoll et al., 2001], Metabuffer [Blanke et al., 2000; Zhang et al., 2001], MPC Compositor [Muraki et al., 2001] and PixelFlow [Molnar et al., 1992; **?**], of which only a few have reached the commercial product stage (i.e. Sepia 2 and MPC

Compositor). However, the inherent inflexibility and setup overhead have limited their distribution and application support. Moreover, with the recent advances in the speed of CPU-GPU interfaces, such as PCI Express, NVLink and other modern interconnects, combinations of software and GPU-based solutions offer more flexibility at comparable performance.

1.3.3 Generic approaches

A number of algorithms and systems for parallel rendering have been developed in the past. On one hand, some general concepts applicable to cluster parallel rendering have been presented in [Mueller, 1995; Mueller, 1997] (sort-first architecture), [Samanta et al., 1999; Samanta et al., 2000] (load balancing), [Samanta et al., 2001] (data replication), or [Cavin et al., 2005; Cavin and Mion, 2006] (scalability). On the other hand, specific algorithms have been developed for cluster based rendering and compositing such as [Ahrens and Painter, 1998], [Correa et al., 2002] and [Yang et al., 2001; Stompel et al., 2003]. However, these approaches do not constitute APIs and libraries that can readily be integrated into existing visualization applications, although the issue of the design of a parallel graphics interface has been addressed in [Igehy et al., 1998].

Only few generic APIs and (cluster-)parallel rendering systems exist which include VR Juggler [Bierbaum et al., 2001] (and its derivatives), Chromium [Humphreys et al., 2002] (an evolution of [Humphreys and Hanrahan, 1999; Humphreys et al., 2000; Humphreys et al., 2001]), ClusterGL [Neal et al., 2011] and OpenGL Multipipe SDK [Jones et al., 2004; Bhaniramka et al., 2005; MPK,]. These approaches can be categorized into transparent interception and distribution of the OpenGL command stream and into the parallization of the application rendering code (Figure 1.3).

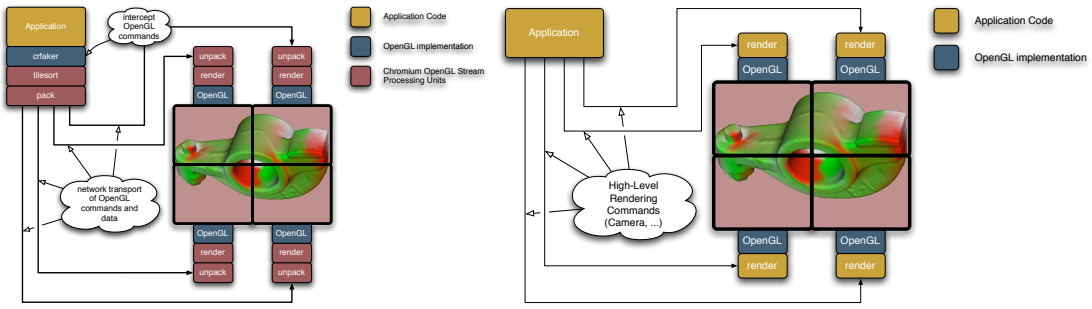

Figure 1.3: *Transparent OpenGL interception and parallelization of the rendering code*

VR Juggler [Bierbaum et al., 2001; Just et al., 1998] is a graphics framework for virtual reality applications which shields the application developer from the underlying hardware architecture, devices and operating system. Its main aim is

to make virtual reality configurations easy to set up and use without the need to know details about the devices and hardware configuration, but not specifically to provide scalable parallel rendering. Extensions of VR Juggler, such as for example ClusterJuggler [Bierbaum and Cruz-Neira, 2003] and NetJuggler [Allard et al., 2002], are typically based on the replication of application and data on each cluster node and basically take care of synchronization issues, but fail to provide a flexible and powerful configuration mechanism that efficiently supports scalable rendering as also noted in [Staadt et al., 2003]. VR Juggler does not support scalable parallel rendering such as sort-first and sort-last task decomposition and image compositing nor does it provide support for network swap barriers (synchronization), distributed objects, image compression and transmission, or multiple rendering threads per process, important for multi-GPU systems.

While Chromium [Humphreys et al., 2002] provides a powerful and transparent abstraction of the OpenGL API, that allows a flexible configuration of display resources, its main limitation with respect to scalable rendering is that it is focused on streaming OpenGL commands through a network of nodes, often initiated from a single source. This has also been observed in [Staadt et al., 2003]. The problem comes in when the OpenGL stream is large in size, due to not only containing OpenGL calls but also the rendered data such as geometry and image data. Only if the geometry and textures are mostly static and can be kept in GPU memory on the graphics card, no significant bottleneck can be expected as then the OpenGL stream is composed of a relatively small number of rendering instructions. However, as it is typical in real-world visualization applications, display and object settings are interactively manipulated, data and parameters may change dynamically, and large data sets do not fit statically in GPU memory but are often dynamically loaded from out-of-core and/or multiresolution data structures. This can lead to frequent updates not only of commands and parameters wich have to be distributed but also of the rendered data itself (geometry and texture), thus causing the OpenGL stream to expand dramatically. Furthermore, this stream of function calls and data must be packaged and broadcast in real-time over the network to multiple nodes for each rendered frame. This makes CPU performance and network bandwidth a more likely limiting factor.

The performance experiments in [Humphreys et al., 2002] indicate that Chromium is working quite well when the rendering problem is fill-rate limited. This is due to the fact that the OpenGL commands and a non-critical amount of rendering data can be distributed to multiple nodes without significant problems and since the critical fill-rate work is then performed locally on the graphics hardware.

Chromium also provides some facilities for parallel application development, namely a sort-last, binary-swap compositing SPU and an OpenGL extension providing synchronization primitives, such as a barrier and semaphore. It leaves other problems, such as configuration, task decomposition as well as process and thread

management unaddressed. Parallel Chromium applications tend to be written for one specific parallel rendering use case, such as for example the sort-first distributed memory volume renderer [Bethel et al., 2003] or the sort-last parallel volume renderer raptor [Houston, 2005]. We are not aware of a generic Chromium-based application using many-to-one sort-first or stereo decompositions.

The concept of transparent OpenGL interception popularized by WireGL and Chromium has received some further contributions. While some commercial implementations such as TechViz and MechDyne Conduit continue to exist, on the research side only ClusterGL [Neal et al., 2011] has been presented recently. ClusterGL employs the same approach as Chromium, but delivers a significantly faster implementation of transparent OpenGL interception and distribution for parallel rendering. Transparent OpenGL interception is an appealing apporach for some applications since it requires no code changes, but it has inherent limitiations due to the fact that eventually the bottleneck becomes the single-threaded application rendering code, the amount of application data the single application instance can load or process, or the the size of the OpenGL command stream send over the network.

CGLX [Doerr and Kuester, 2011] tries to bring parallel execution transparently to OpenGL applications, by emulating the GLUT API and intercepting certain OpenGL calls. In contrast to frameworks like Chromium and ClusterGL which distribute OpenGL calls, CGLX follows the distributed application approach. This works transparently for trivial applications, but quickly requires the application developer to address the complexities of a distributed application, when mutable application state needs to be synchronized across processes. For realistic applications, writing parallel applications remains the only viable approach for scalable parallel rendering, as shown by the success of Paraview, Visit and Equalizer-based applications.

OpenGL Multipipe SDK (MPK) [Bhaniramka et al., 2005] implemented an effective parallel rendering API for a shared memory multi-CPU/GPU system. It is similar to IRIS Performer [Rohlf and Helman, 1994] in that it handles multi-GPU rendering by a lean abstraction layer via a conceptual callback mechanism, and that it runs different application tasks in parallel. However, MPK is not designed nor meant for rendering nodes separated by a network. MPK focuses on providing a parallel rendering framework for a single application, parts of which are run in parallel on multiple rendering channels, such as the culling, rendering and final image compositing processes.

Software for driving and interacting with tiled display walls has received significant attention, including Sage [DeFanti et al., 2009] and Sage 2 [Marrinan et al., 2014] in particular. Sage was built entirely around the concept of a shared framebuffer where all content windows are separate applications using pixel streaming but is no longer actively supported. Sage 2 is a complete, browser-centric

reimplementation where each application is a web application distributed across browser instances. DisplayCluster [Johnson et al., 2012], and its continuation Tide [Blue Brain Project, 2016], also implement the shared framebuffer concept of Sage, but provide a few native content applications integrated into the display servers. These solutions implement a scalable display environment and are a target display platform for scalable 3D graphics applications.

1.4 Dissertation Structure

CHAPTER 2

CONTRIBUTIONS

CHAPTER 3

PARALLEL RENDERING FRAMEWORK ARCHITECTURE

3.1 Overview

A generic parallel rendering framework has to cover a wide range of use cases, target systems, and configurations. This requires on one hand a strong separation between the implementation of an application and its configuration, and on the other hand a careful design to allow the resulting program to scale up to hundreds of nodes, while providing a minimally invasive API for the developer. In this section we present the system architecture of the Equalizer parallel rendering framework, and motivate its design in contrast to related work.

The motivation to use parallel rendering is either driven by the need to drive multiple displays or projectors from multiple GPUs and potentially multiple nodes, or by the need to increase rendering performance to be able to visualize more data or use a more demanding rendering algorithm for higher visual quality. Occasionally both needs coincide, for example for the analysis for large data sets on high fidelity visualization systems.

Fundamentally, there are two approaches to enable applications to use multiple GPUs: transparent interception at the graphics API (typically OpenGL) or extending the application to support parallel rendering natively. The first approach has been extensively explored by Chromium and others, while the second is the foundation for this thesis. The architecture of Equalizer is founded on an in-depth

requirements analysis of typical visualization applications, existing frameworks, and previous work on OpenGL Multipipe SDK.

The task of parallelizing a visualization application boils down to configuring the application's rendering code differently for each resource, enabling this rendering code to access the correct data, and synchronizing execution. For scalable rendering, when multiple GPUs are used to accelerate a single output, partial results need to be collected from all contributing resources and combined on the output. Equalizer has a strong seperation of the rendering code from its runtime configuration. The configuration is layed out in a hierarchical resource description and compound trees configuring the resources for parallel and scalable rendering. The configuration is a dynamic runtime structure, with a serialized configuration file format.

In the following we will first describe the execution model and runtime configurability, followed by the how the generic configuration is used to model the desired visualization setup, and finally introduce specifics of scalable and distributed rendering.

3.2 Asynchronous Execution Model

The core execution model for parallel rendering was pioneered by CAVELib [DeFanti et al., 1998], refined by OpenGL Multipipe SDK for shared memory systems and scalable rendering, and substantially extended by Equalizer for asynchronous and distributed execution. By analysing the typical architecture of a visualization application we observe an initialization phase, a main rendering loop, and an exit phase. Equalizer decomposes these steps for parallel execution.

The main rendering loop typically consists of four phases: submitting the rendering commands to the graphics subsystem, displaying the renderered image, and retrieving events from the operating system, based on which the application state is updated and a new image is rendered. The configuration of the rendering is largely hard-coded, with a few configurables such as field of view or stereo separation. For parallel execution, we need to separate the rendering code from this main loop, and execute it in parallel with different rendering parameters, as shown in Figure 3.1. Similarly, the initialization and exit phase also needs to be decomposed to allow managing multiple, distributed resources.

Without going into the details at this point, another critical design parameter are synchronization points. Most implementations use a per-frame barrier or similar synchronization to manages parallel execution. In larger installations, this is detrimental for scalability, as even slight load imbalances limit parallel speedup. The Equalizer execution model is fully asynchronuous, and only introduces synchronization points when strictly needed. The main synchronization points are:

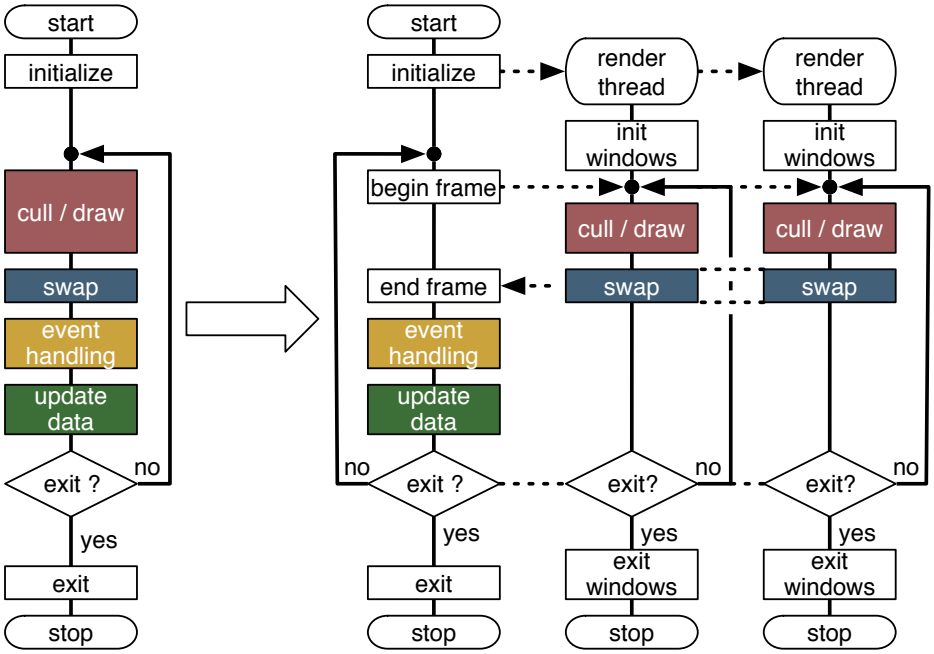

Figure 3.1: *Simplified execution flow of a classical visualization application and an Equalizer application.*

configured swap barriers between a set of output which have to display simultaneously, the availability of input frames for scalable rendering, and a task synchronization to prevent runaway of the main loop execution. By default, Equalizer keeps up to one frame of latency in execution, that is, some resource might render the next frame while others are still finishing the current. Nonetheless, resources which are ready will immediately display their result. The asynchronous execution architecture, coupled with a frame of latency, allows pipelining of many operations, such as the application event processing, task computation and load balancing, rendering, image readback, compression, network transmission, and compositing.

Figure 3.2 shows the execution of the rendering tasks of a 2-node sort-first compound without latency and with a latency of one frame. The asynchronous execution pipelines rendering operations and hides imbalances in the load distribution, resulting in an improved framerate. For example, we have observed a speedup of 15% on a five-node rendering cluster when using a latency of one frame instead of no latency.

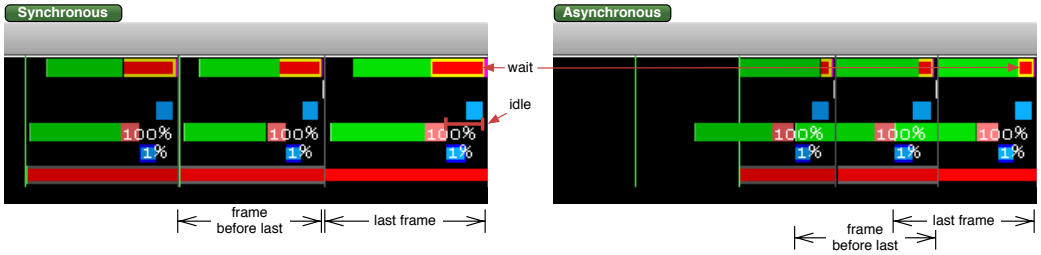

Figure 3.2: *Synchronous and Asynchronous Execution*

3.2.1 Programming Interface

Equalizer provides a framework to facilitate the development of distributed as well as non-distributed parallel rendering applications. The programming interface is based on a set of C++ classes, modeled closely to the resource hierarchy of a graphics rendering system. The application subclasses these objects and overrides C++ task methods, similar to C callbacks. These task methods will be called in parallel by the framework, depending on the current configuration. This parallel rendering interface is significantly different from Chromium [Humphreys et al., 2002] and more similar to VRJuggler [Bierbaum et al., 2001] or OpenGL Multipipe SDK [Bhaniramka et al., 2005].

To separate the responsibilities in a parallel rendering application, different entities are resonsible for different aspects of the runtime system: the application process to drive a rendering session, the server process or thread to control the parallel rendering configuration, render clients to execute the rendering tasks, and an administrative API to reconfigure the rendering session at runtime.

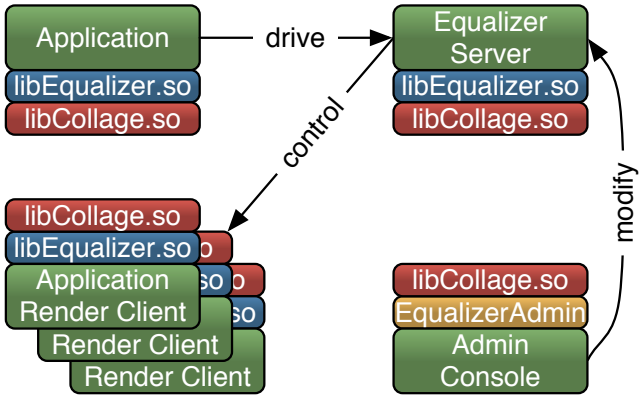

Figure 3.3: *Parallel Rendering Entities*

3.2.2 Application

The main application thread in Equalizer drives the rendering, that is, it carries out the main rendering loop, but does not actually execute any rendering. Depending on the configuration, the application process often hosts one or more render client

threads. When a configuration has no additional nodes besides the application node, all application code is executed in the same process, and no network data distribution has to be implemented. The main rendering loop is quite simple: The application requests a new frame to be rendered, synchronizes on the completion of a frame and processes events received from the render clients. Figure 3.1 shows a simplified execution model of an Equalizer application.

3.2.3 Server

The Equalizer server manages the parallel rendering session. It is an asynchronous execution thread or process which receives requests from the application and serves these requests using the current configuration, launching and stopping rendering client processes on nodes, determining the rendering tasks for a frame, and synchronizing the completion of tasks.

3.2.4 Render Client

During initialization of the server, the application provides a rendering client executable. The rendering client is often, especially for simple applications, the same executable as the application. However, in more sophisticated implementations the rendering client can be a smaller executable which only contains the application-specific rendering code. The server deploys this rendering client on all nodes specified in the configuration.

In contrast to the application process, the rendering client does not have a main loop and is completely controlled by the Equalizer framework based on application's commands. A render client consists of the following threads: the node main thread, one network receive thread, and one thread for each graphics card (GPU) to execute rendering tasks. If a configuration also uses the application node for rendering, then the application process uses one or more render threads, consistent with render client processes.

The Equalizer client library implements the main loop, which receives network events, processes them, and invokes the necessary task methods provided by the developer.

The task methods clear the frame buffer as necessary, execute the OpenGL rendering commands as well as readback, and assemble partial frame results for scalable rendering. All tasks have default implementations so that only the application specific methods have to be implemented, which at least involves the `frameDraw()` method executing a rendering task. For example, the default callbacks for frame recomposition during scalable rendering implement tile-based assembly for sort-first and stereo decompositions, and z-buffer compositing for sort-last rendering.

Render Context

The render context is the core entity abstracting the application-specific rendering algorithm from the system-specific configuration. It specifies:

Buffer OpenGL-style read and draw buffer as well as color mask. These parameters are influenced by the current eye pass, eye separation and anaglyphic stereo settings.

Viewport Two-dimensional pixel viewport restricting the rendering area. For correct operations, both glViewport and glScissor have to be used. The pixel viewport is influenced by the destination viewport definition and viewports set for sort-first/2D decompositions.

Frustum Frustum parameters as defined by glFrustum. Typically the frustum used to set up the OpenGL projection matrix. The frustum is influenced by the destination's view definition, sort-first decomposition, tracking head matrix and the current eye pass.

Head Transformation A transformation matrix positioning the frustum. For planar views this is an identity matrix and is used in immersive rendering. It is normally used to set up the 'view' part of the modelview matrix, before static light sources are defined.

Range A one-dimensional range with the interval [0..1]. This parameter is optional and should be used by the application to render only the appropriate subset of its data for sort-last rendering.

Event Handling

Event handling routes events from the source (the window in the rendering thread) gradually to the the application main thread for consumption. At each step, events can be observed, transformed or dropped. Events are received from the operating system in the rendering thread, transformed there into a generic representation, and sent over the network to the application. The application processes them in the main loop and modifies its internal state accordingly.

3.3 Configuration

A configuration consists of the declaration of the rendering (hardware) resources, the physical and logical description of the projection system, and the description on how the aforementioned resources are used for parallel and scalable rendering.

The rendering resources are represented in a hierarchical tree structure which corresponds to the physical and logical resources found in a 3D rendering environment: nodes (computers), pipes (graphics cards), windows, and channels.

Physical layouts of display systems are configured using canvases with segments, which represent 2D rendering areas composed of multiple displays or projectors. Logical layouts are applied to canvases and define views on a canvas.

Scalable resource usage is configured using a compound tree, which is a hierarchical representation of the rendering decomposition and recomposition across the resources.

3.3.1 Rendering Resources

The first part of the configuration is a hierarchical structure of nodes-pipes-windows-channels describing the rendering resources. The developer will use instances of these classes to implement application logic and manage data.

The node is the representation of a single computer in a cluster. One operating system process of the render client executable will be used for each node. Each configuration might also use an application node, in which case the application process is also used for rendering. All node-specific task methods are executed from the main thread.

The pipe is the abstraction of a graphics card (GPU), and uses an operating system thread for rendering. All pipe, window and channel task methods are executed from the pipe thread. The pipe maintains the information about the GPU to be used by the windows for rendering.

The window encapsulates a drawable and an OpenGL context. The drawable can be an on-screen window or an off-screen pbuffer or framebuffer object (FBO). Windows on the same pipe share their OpenGL rendering resources. They execute their rendering tasks sequentially on the pipe's execution thread.

The channel is the abstraction of an OpenGL viewport within its parent window. It is the entity executing the actual rendering. The channel's viewport is overwritten when it is rendering for another channel during scalable rendering. Multiple channels in application windows may be used to view the model from different viewports. Sometimes, a single window is split across multiple projectors, e.g., by using an external splitter such as the Matrox TripleHead2Go.

3.3.2 Display Resources

Display resources are the second part of a configuration. They describe the physical display setup (canvases-segments), logical display (layouts-views) and head tracking of users within the visualization installation (observers).

A canvas represents one physical projection surface, e.g., a PowerWall, a curved screen or an immersive installation. Canvases provide a convenient way to configure projection surfaces. A canvas uses layouts, which describe logical views. Typically, each desktop window uses one canvas, one segment, one layout

and one view. One configuration might drive multiple canvases, for example an immersive installation and an operator station. Planar surfaces, e.g., a display wall, configure a frustum for the respective canvas. For non-planar surfaces, the frustum will be configured on each display segment.

The frustum can be specified as a wall or projection description in the same reference system as used by the head-tracking matrix calculated by the application. A wall is completely defined by the bottom-left, bottom-right and top-left coordinates relative to the origin. A projection is defined by the position and head-pitch-roll orientation of the projector, as well as the horizontal and vertical field-of-view and distance of the projection wall. Figure 3.4 illustrates the wall and projection frustum parameters.

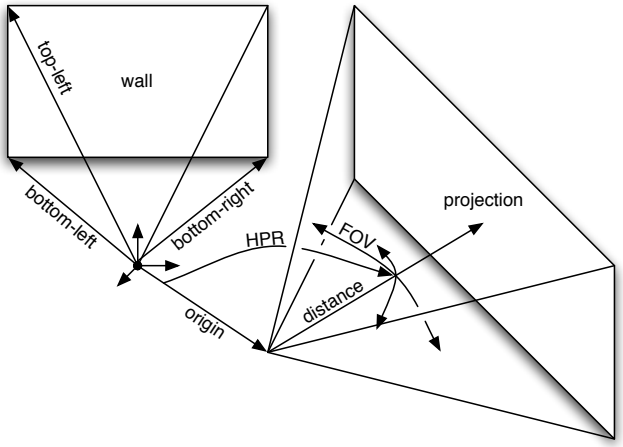

Figure 3.4: *Wall and Projection Parameters*

A canvas consists of one or more segments. A planar canvas typically has a frustum description (see Section 3.3.3), which is inherited by the segments. Non-planar frusta are configured using the segment frusta. These frusta typically describe a physically correct display setup for Virtual Reality installations.

A canvas has one or more layouts. One of the layouts is the active layout, that is, this set of views is currently used for rendering. It is possible to specify OFF as a layout, which deactivates the canvas. It is possible to use the same layout on different canvases.

A segment represents one output channel of the canvas, e.g., a projector or a display. A segment has an output channel, which references the channel to which the display device is connected. To synchronize the video output, a canvas swap barrier or a swap barrier on each segment synchronize the respective window buffer swaps.

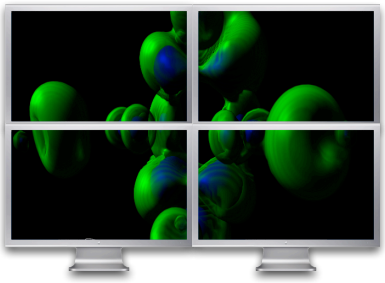

Figure 3.5: *A Canvas using four Segments*

A segment covers a part of its parent canvas, which is configured using the segment viewport. The viewport is in normalized coordinates relative to the canvas. Segments might overlap

(edge-blended projectors) or have gaps between
each other (display walls, Figure 3.5[1]). The viewport is used to configure the segment's default frustum from the canvas frustum description, and to place layout views correctly.

A layout is the grouping of logical views. It is used by one or more canvases. For all given layout/canvas combinations, Equalizer creates destination channels when the configuration file is loaded. These destination channels are later referenced by compounds to configure scalable rendering. Layouts can be switched at runtime by the application. Switching a layout will activate different destination channels for rendering.

A view is a logical view of the application data, in the sense used by the Model-View-Controller pattern. It can be a scene, viewing mode, viewing position, or any other representation of the application's data. A view has a fractional viewport relative to its layout. A layout is often fully covered by its views, but this is not a requirement. Each view can have a frustum description. The view's frustum overrides frusta specified at the canvas or segment level. This is typically used for non-physically correct rendering, e.g., to compare two models side-by-side on a canvas. If the view does not specify a frustum, it will use the sub-frustum resulting from the covered area on the canvas.

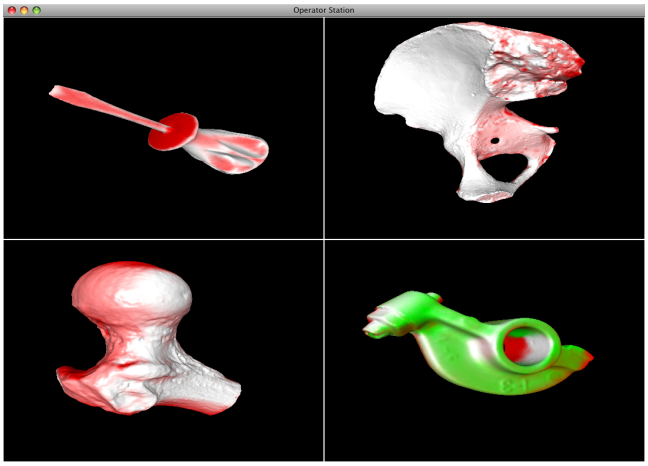

Figure 3.6: *Layout with four Views*

A view might have an observer, in which case its frustum is tracked by this observer. Figure 3.6 shows an example layout using four views on a single segment. Figure **??** shows a real-world setup of a single canvas with six segments using underlap, with a two-view layout activated. This configuration generates eight destination channels.

An observer represents an actor looking at multiple views. It has a head matrix, defining its position and orientation within the world, an eye separation and focus distance parameters. Typically, a configuration has one observer. Configurations with multiple observers are used if multiple, head-tracked users are in

[1] Dataset courtesy of VolVis distribution of SUNY Stony Brook, NY, USA.

Figure 3.7: *Display Wall using a six-Segment Canvas with a two-View Layout*

the same configuration session, e.g., a non-tracked control host with two tracked head-mounted displays.

3.3.3 Compounds

Compound trees are used to describe how multiple rendering resources are combined to produce the desired output, especially how multiple GPUs are aggregated to increase the performance. They are the core contribution enabling a flexible resource configuration.

Root compound Define an empty top-level compound when synchronizing multiple destination views. Multiple destination views are used for multi-display systems, e.g., a PowerWall or CAVE. All windows used for one display surface should be swap-locked (see below) to provide a seamless image. A single destination view is typically used for providing scalability to a single workstation window.

Destination compound(s) Define one compound for each destination channel, either as a child of the empty group, or as a top-level compound. Set the destination channel by using the canvas, segment, layout and view name or index. The compound frustum will be calculated automatically based on the segment or view frustum. Note that one segment may created multiple view/segment channels, one for each view intersection of each layout used

on the canvas. Only the compounds belonging to the active layout of a canvas are active at runtime.

Scalability If desired, define scalability for each of your destination compounds. Add one compound using a source channel for each contributor to the rendering. The destination channel may also be used as a source.

Decomposition On each child compound, limit the rendering task of that child by setting the viewport, range, period and phase, pixel, subpixel, eye or zoom as desired.

Runtime Adjustments A load_equalizer may be used on the destination compounds to set the viewport or range of all children each frame, based on the current load. A view_equalizer may be used on the root compound to assign resources to all destination compounds, which have to use load_equalizers. A framerate_equalizer should be used to smoothen the framerate of DPlex compounds. A DFR_equalizer may be used to set the zoom of a compound to achieve a constant framerate. One compound may have multiple equalizers, e.g., a load_equalizer and a DFR_equalizer for a 2D compound with a constant framerate.

Recomposition For each source compound, define an output_frame to read back the result. Use this output frame as an input_frame on the destination compound. The frames are connected with each other by their name, which has to be unique within the root compound tree. For parallel compositing, describe your algorithm by defining multiple input and output frames across all source compounds.

Compound Channels

Each compound has a channel, which is used by the compound to execute the rendering tasks. One channel might be used by multiple compounds. Compounds are only active if their corresponding destination channel is active, that is, if the parent layout of the view which created the destination channel is active on at least one canvas.

Unused channels, windows, pipes and nodes are not instantiated during initialization. Switching an active layout may cause rendering resources to be stopped and started. The rendering tasks for the channels are computed by the server and send to the appropriate render client nodes at the beginning of each frame.

Frustum

Compounds have a frustum description to define the physical layout of the display environment. The frustum specification is described in Section **??**. The frustum

description is inherited by the children, therefore the frustum is defined on the topmost compound, typically by the corresponding segment.

Compound Classification

The channels of the leaf compounds in the compound tree are designated as source channels. The topmost channel in the tree is the destination channel. One compound tree might have multiple destination channels, e.g., for a swap-synchronized immersive installation.

All channels in a compound tree work for the destination channel. The destination channel defines the 2D pixel viewport rendered by all leaf compounds. The destination channel and pixel viewport cannot be overridden by child compounds.

Tasks

Compounds execute a number of tasks: clear, draw, assemble and readback. By default, a leaf compound executes all tasks and a non-leaf compound assemble and readback. A non-leaf compound will never execute the draw task.

A compound can be configured to execute a specific set of tasks, for example to configure the multiple steps used by binary-swap compositing.

Decomposition - Attributes

Compounds have attributes which configure the decomposition of the destination channel's rendering, which is defined by the viewport, frustum and database. A viewport decomposes the destination channel and frustum in screen space. A range tells the application to render a part of its database, and an eye rendering pass can selectively render different stereo passes. A pixel parameter adjusts the frustum so that the source channel renders an even subset of the parent's pixels. A subpixel parameter tells the source channels to render different samples for one pixel to perform anti-aliasing or depth-of-field rendering. Setting one or multiple attributes causes the parent's view to be decomposed accordingly. Attributes are cumulative, that is, intermediate compound attributes affect and therefore decompose the rendering of all their children.

Recomposition - Frames

Compounds use output and input frames to configure the recomposition of the resulting pixel data from the source channels. An output frame connects to an input frame of the same name. The selected frame buffer data is transported from the output channel to the input channel. The assembly routine of the input channel will block on the availability of the output frame. This composition process is

extensively described in Section **??**. Frame names are only valid within the compound tree, that is, an output frame from one compound tree cannot be used as an input frame of another compound tree.

3.4 Compositing

3.5 Load Balancing

3.6 Distribution Layer

CHAPTER 4

PARALLEL RENDERING MODES

CHAPTER 5

COMPOSITING OPTIMISATIONS

C H A P T E R 6

LOAD BALANCING ALGORITHMS

CHAPTER 7

DATA DISTRIBUTION AND SYNCHRONIZATION

C H A P T E R 8

CONCLUSION

8.1 Future Work

BIBLIOGRAPHY

[MPK,] OpenGL Multipipe SDK.

[Agranov and Gotsman, 1995] Agranov, G. and Gotsman, C. (1995). Algorithms for rendering realistic terrain image sequences and their parallel implementation. *The Visual Computer*, 11(9):455–464.

[Ahrens and Painter, 1998] Ahrens, J. and Painter, J. (1998). Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*.

[Allard et al., 2002] Allard, J., Gouranton, V., Lecointre, L., Melin, E., and Raffin, B. (2002). NetJuggler: Running VR Juggler with multiple displays on a commodity component cluster. In *Proceeding IEEE Virtual Reality*, pages 275–276.

[Bethel et al., 2003] Bethel, W. E., Humphreys, G., Paul, B., and Brederson, J. D. (2003). Sort-first, distributed memory parallel visualization and rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 41–50.

[Bhaniramka et al., 2005] Bhaniramka, P., Robert, P. C. D., and Eilemann, S. (2005). OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization*, pages 119–126.

[Bierbaum and Cruz-Neira, 2003] Bierbaum, A. and Cruz-Neira, C. (2003). ClusterJuggler: A modular architecture for immersive clustering. In *Proceedings Workshop on Commodity Clusters for Virtual Reality, IEEE Virtual Reality Conference*.

[Bierbaum et al., 2001] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. (2001). VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality*, pages 89–96.

[Blanke et al., 2000] Blanke, W., Bajaj, C., D.Fussel, and Zhang, X. (2000). The metabuffer: A scalable multi-resolution 3-d graphics system using commodity rendering engines. Technical Report TR2000-16, University of Texas at Austin.

[Blue Brain Project, 2016] Blue Brain Project (2016). Tide: Tiled Interactive Display Environment. https://github.com/BlueBrain/Tide.

[Cavin and Mion, 2006] Cavin, X. and Mion, C. (2006). Pipelined sort-last rendering: Scalability, performance and beyond. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*.

[Cavin et al., 2005] Cavin, X., Mion, C., and Filbois, A. (2005). COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proceedings IEEE Visualization*, pages 111–118. Computer Society Press.

[Correa et al., 2002] Correa, W. T., Klosowski, J. T., and Silva, C. T. (2002). Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96.

[Crockett, 1997] Crockett, T. W. (1997). An introduction to parallel rendering. *Parallel Computing*, 23:819–843.

[DeFanti et al., 1998] DeFanti, T. A., Adamczyk, D., Cruz-Neira, C., Czernuszenko, M., Ghazisaedy, M., Hayakawa, H., Pape, D., Sandin, D. J., Nickless, B., and Sherman, B. (1998). CAVELib. http://www.evl.uic.edu/pape/CAVE/prog/.

[DeFanti et al., 2009] DeFanti, T. A., Leigh, J., Renambot, L., Jeong, B., Verlo, A., Long, L., Brown, M., Sandin, D. J., Vishwanath, V., Liu, Q., Katz, M. J., Papadopoulos, P., Keefe, J. P., Hidley, G. R., Dawe, G. L., Kaufman, I., Glogowski, B., Doerr, K.-U., Singh, R., Girado, J., Schulze, J. P., Kuester, F., and Smarr, L. (2009). The optiportal, a scalable visualization, storage, and

computing interface device for the optiputer. *Future Gener. Comput. Syst.*, 25(2):114–123.

[Doerr and Kuester, 2011] Doerr, K.-U. and Kuester, F. (2011). CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):320–332.

[Garcia and Shen, 2002] Garcia, A. and Shen, H.-W. (2002). An interleaved parallel volume renderer with PC-clusters. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 51–60.

[Houston, 2005] Houston, M. (2005). Raptor. http://graphics.stanford.edu/projects/raptor/.

[Huang et al., 2000] Huang, J., Shareef, N., Crawfis, R., Sadayappan, P., and Mueller, K. (2000). A parallel splatting algorithm with occlusion culling. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*.

[Humphreys et al., 2000] Humphreys, G., Buck, I., Eldridge, M., and Hanrahan, P. (2000). Distributed rendering for scalable displays. *IEEE Supercomputing*.

[Humphreys et al., 2001] Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. (2001). WireGL: A scalable graphics system for clusters. In *Proceedings Annual Conference on Computer Graphics and Interactive Techniques*, pages 129–140.

[Humphreys and Hanrahan, 1999] Humphreys, G. and Hanrahan, P. (1999). A distributed graphics system for large tiled displays. *IEEE Visualization 1999*, pages 215–224.

[Humphreys et al., 2002] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T. (2002). Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702.

[Igehy et al., 1998] Igehy, H., Stoll, G., and Hanrahan, P. (1998). The design of a parallel graphics interface. *Proceedings of SIGGRAPH 98*, pages 141–150.

[Johnson et al., 2006] Johnson, A., Leigh, J., Morin, P., and Van Keken, P. (2006). GeoWall: Stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 26(6):10–14.

[Johnson et al., 2012] Johnson, G. P., Abram, G. D., Westing, B., Navr'til, P., and Gaither, K. (2012). DisplayCluster: An Interactive Visualization Environment for Tiled Displays. In *2012 IEEE International Conference on Cluster Computing*, pages 239–247.

[Jones et al., 2004] Jones, K., Danzer, C., Byrnes, J., Jacobson, K., Bouchaud, P., Courvoisier, D., Eilemann, S., and Robert, P. (2004). SGI®OpenGL Multipipe™SDK User's Guide. Technical Report 007-4239-004, Silicon Graphics.

[Just et al., 1998] Just, C., Bierbaum, A., Baker, A., and Cruz-Neira, C. (1998). VR Juggler: A framework for virtual reality development. In *Proceedings Immersive Projection Technology Workshop*.

[Lever, 2004] Lever, P. G. (2004). SEPIA – applicability to MVC. White paper Manchester Visualization Centre (MVC), University of Manchester.

[Li et al., 1996] Li, P. P., Duquette, W. H., and Curkendall, D. W. (1996). RIVA: A versatile parallel rendering system for interactive scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):186–201.

[Li et al., 1997] Li, P. P., Whitman, S., Mendoza, R., and Tsiao, J. (1997). ParVox: A parallel splatting volume rendering system for distributed visualization. In *Proceedings IEEE Parallel Rendering Symposium*, pages 7–14.

[Lombeyda et al., 2001a] Lombeyda, S., Moll, L., Shand, M., Breen, D., and Heirich, A. (2001a). Scalable interactive volume rendering using off-the-shelf components. Technical Report CACR-2001-189, California Institute of Technology.

[Lombeyda et al., 2001b] Lombeyda, S., Moll, L., Shand, M., Breen, D., and Heirich, A. (2001b). Scalable interactive volume rendering using off-the-shelf components. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics*, pages 115–121.

[Marrinan et al., 2014] Marrinan, T., Aurisano, J., Nishimoto, A., Bharadwaj, K., Mateevitsi, V., Renambot, L., Long, L., Johnson, A., and Leigh, J. (2014). SAGE2: A new approach for data intensive collaboration using Scalable Resolution Shared Displays. In *Collaborative Computing: Networking, Applications and Worksharing*, pages 177–186.

[Moll et al., 1999] Moll, L., Heirich, A., and Shand, M. (1999). Sepia: scalable 3D compositing using PCI pamette. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 146–155.

[Molnar et al., 1994] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. (1994). A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32.

[Molnar et al., 1992] Molnar, S., Eyles, J., and Poulton, J. (1992). PixelFlow: High-speed rendering using image composition. In *Proceedings ACM SIGGRAPH*, pages 231–240.

[Mueller, 1995] Mueller, C. (1995). The sort-first rendering architecture for high-performance graphics. In *Proceedings Symposium on Interactive 3D Graphics*, pages 75–84. ACM SIGGRAPH.

[Mueller, 1997] Mueller, C. (1997). Hierarchical graphics databases in sort-first. In *Proceedings IEEE Symposium on Parallel Rendering*, pages 49–. Computer Society Press.

[Muraki et al., 2001] Muraki, S., Ogata, M., Ma, K.-L., Koshizuka, K., Kajihara, K., Liu, X., Nagano, Y., and Shimokawa, K. (2001). Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In *Proceedings ACM/IEEE Conference on Supercomputing*, pages 51–51.

[Neal et al., 2011] Neal, B., Hunkin, P., and McGregor, A. (2011). Distributed OpenGL rendering in network bandwidth constrained environments. In Kuhlen, T., Pajarola, R., and Zhou, K., editors, *Proceedings Eurographics Conference on Parallel Graphics and Visualization*, pages 21–29. Eurographics Association.

[Nie et al., 2005] Nie, W., Sun, J., Jin, J., Li, X., Yang, J., and Zhang, J. (2005). A dynamic parallel volume rendering computation mode based on cluster. In *Proceedings Computational Science and its Applications*, volume 3482 of *Lecture Notes in Computer Science*, pages 416–425.

[Niski and Cohen, 2007] Niski, K. and Cohen, J. D. (2007). Tile-based level of detail for the parallel age. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1352–1359.

[Rohlf and Helman, 1994] Rohlf, J. and Helman, J. (1994). IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings ACM SIGGRAPH*, pages 381–394. ACM Press.

[Samanta et al., 2001] Samanta, R., Funkhouser, T., and Li, K. (2001). Parallel rendering with K-way replication. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. Computer Society Press.

[Samanta et al., 2000] Samanta, R., Funkhouser, T., Li, K., and Singh, J. P. (2000). Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 97–108.

[Samanta et al., 1999] Samanta, R., Zheng, J., Funkhouser, T., Li, K., and Singh, J. P. (1999). Load balancing for multi-projector rendering systems. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 107–116.

[Schulze and Lang, 2002] Schulze, J. P. and Lang, U. (2002). The parallelization of the perspective shear-warp volume rendering algorithm. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 61–70.

[Staadt et al., 2003] Staadt, O. G., Walker, J., Nuber, C., and Hamann, B. (2003). A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *Proceedings Eurographics Workshop on Virtual Environments*, pages 261–270.

[Stoll et al., 2001] Stoll, G., Eldridge, M., Patterson, D., Webb, A., Berman, S., Levy, R., Caywood, C., Taveira, M., Hunt, S., and Hanrahan, P. (2001). Lightning-2: A high-performance display subsystem for PC clusters. In *Proceedings ACM SIGGRAPH*, pages 141–148.

[Stompel et al., 2003] Stompel, A., Ma, K.-L., Lum, E. B., Ahrens, J., and Patchett, J. (2003). SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 33–40.

[Vezina and Robertson, 1991] Vezina, G. and Robertson, P. K. (1991). Terrain perspectives on a massively parallel SIMD computer. In *Proceedings Computer Graphics International (CGI)*, pages 163–188.

[Wittenbrink, 1998] Wittenbrink, C. M. (1998). Survey of parallel volume rendering algorithms. In *Proceedings Parallel and Distributed Processing Techniques and Applications*, pages 1329–1336.

[Yang et al., 2001] Yang, D.-L., Yu, J.-C., and Chung, Y.-C. (2001). Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing*, 18(2):201–22–.

[Zhang et al., 2001] Zhang, X., Bajaj, C., and Blanke, W. (2001). Scalable isosurface visualization of massive datasets on COTS clusters. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics*, pages 51–58.

CURRICULUM VITAE

Personal Information

Name	Fatih Erol
Date of birth	XXXXXX XX, 19XX
Place of birth	Trabzon, Turkey

Education

Publications

Conference Publications

Journal Articles