University of Zurich[UZH]

Department of Informatics

Parallel Rendering and Large Data Visualization

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

May 10, 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, May 10, 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

NOTATIONS

Acronyms

CPU	Central Processing Unit
FPS	Frames per Second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
LB	Load Balancing
LOD	Level of Detail

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

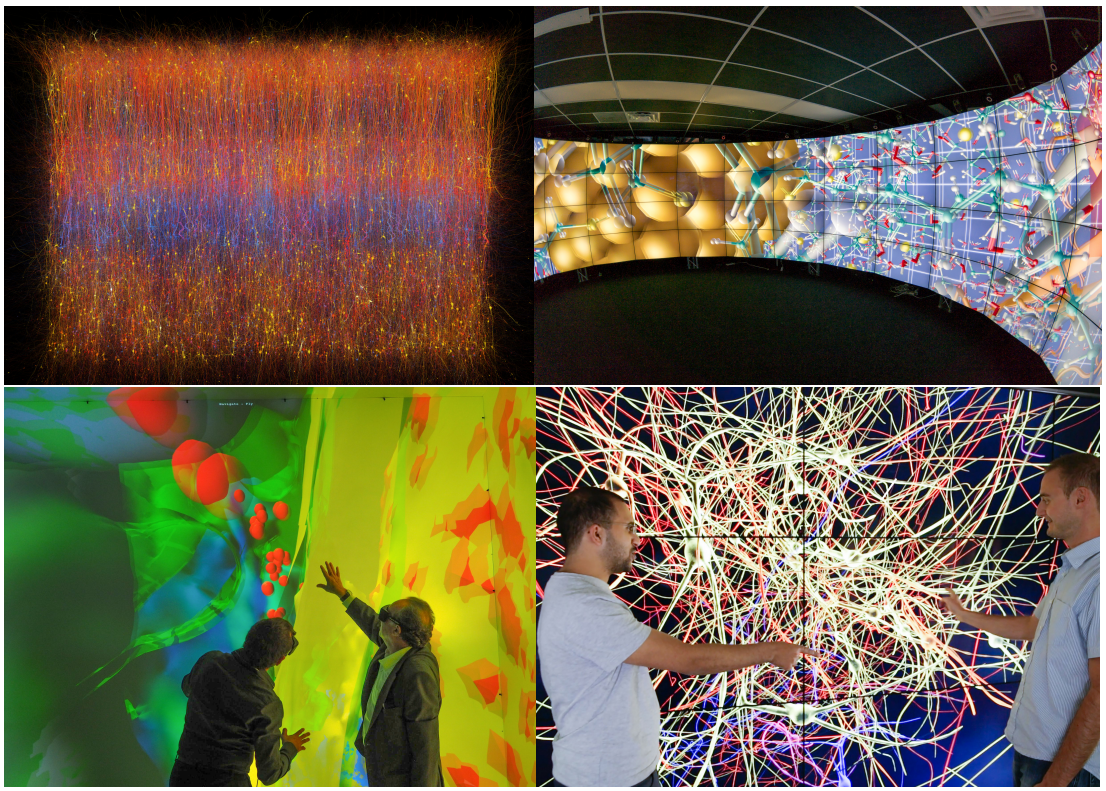


Figure 1.1: *Large Data Visualization: Large data visualization of a brain simulation, molecular visualization in the Cave*[2]*, exploration of EM stack reconstructions in a Cave, collaborative data analysis on a tiled display wall.*

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

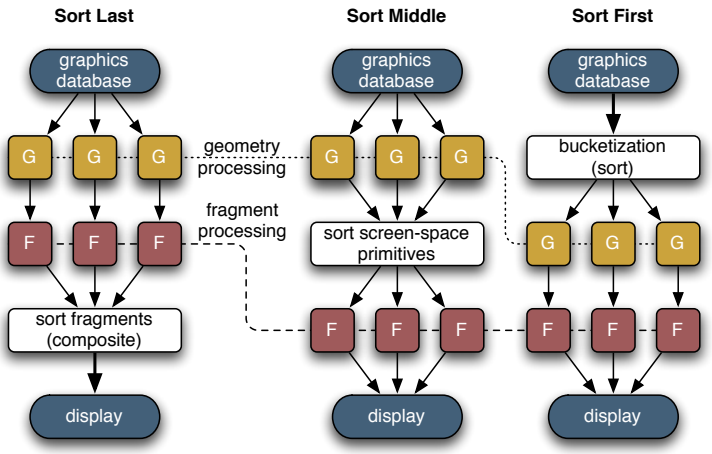


Figure 1.2: *Sort-last, sort-middle and sort-first parallel rendering*

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; Eyles et al., 1997], 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).

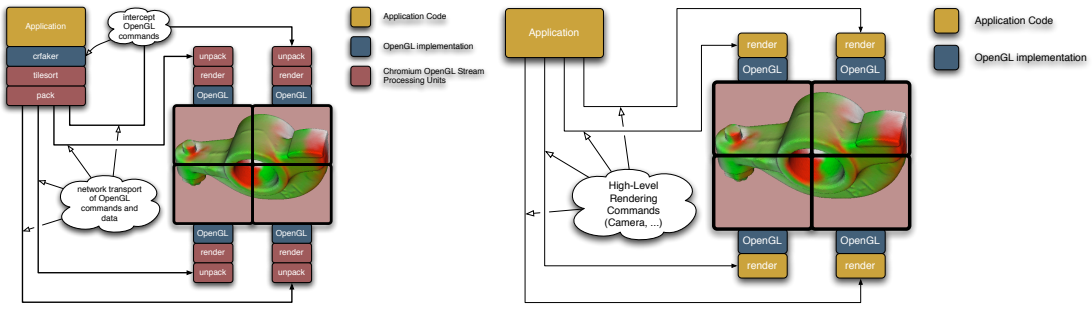


Figure 1.3: *Transparent OpenGL interception and parallelization of the rendering code*

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

Features, papers, individual contribution per paper.

2.1 Parallel Rendering Architecture

A substantial part of this thesis is the description and implementation of an architecture for a parallel rendering framework, which advances the state of the art in many aspects:

Minimally invasive API: The guiding principle when designing the API was to enable applications to retain all their rendering code and application logic. 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. The contract for the implementation of the task methods does not assume any rendering library, algorithm or technology, thus facilitating the adaptation of existing applications for parallel rendering. This parallel rendering interface is significantly different from Chromium [Humphreys et al., 2002] and more similar to VR-Juggler [Bierbaum et al., 2001] or MPK [Bhaniramka et al., 2005].

Runtime configuration: The architecture of our parallel rendering framework makes a clear separation between rendering algorithm and the runtime configuration. It defines a contract between the framework and the application

code which defines the rendering context, and uses this context to drive the application output depending on the runtime configuration. Application developers are unware of parallel rendering setups and make no assumptions on how the rendering code will be executed. This clear separation yields parallel rendering applications which can be deployed on a wide set of installations, and are often configured in ways unforeseen during their development.

Compound trees: The introduction of compounds, and their underlying formalism, provide a formalization of a flexible task decomposition and result recompostion for parallel rendering. Compound trees allow for easy specification of complex parallel task decomposition strategies which are automatically implemented and executed by the Equalizer system. They generalize parallel rendering principles without hardcoding a specific parallel rendering algorithm, thus proposing an orthogonal parameter set for decomposing rendering tasks, assembling results, and adapting these parameters at runtime.

Display abstraction: Large scale visualization setup cover a wide set of use cases from classical workstation setup to monoscopic tiled display walls, stereoscopic, edge-blended multi-projector walls to fully immersive installations such as CAVE systems. Consequently, applications running on these systems serve many different usage scenarios. Our novel canvas-layout abstraction provides a simple configuration of all these installations and empowers applications using these installations with 2D and 3D contextual information, runtime stereo configuration, and head tracking.

Modular architecture: Our architecture uses layered abstractions which gradually provide higher level abstractions. On the low level, a network library for distributed abstractions provides the substrate for Equalizer and applications. Within each library, a clear separation of responsibilities allows to combine existing algorithms easily. For example, an advanced feature such as cross-segment load-balancing relies on per-segment load-balancers, and both reconfigure the underlying compound tree each frame.

[Eilemann et al., 2009] and [Eilemann et al., 2018] publish the architectural contributions for this thesis. Any algorithmic and architectural contributions in these publications are contributed by the author, while experimental results have significant contributions from the secondary authors.

2.2 Scalable Rendering

2.3 Load Balancing

2.4 Applications

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:

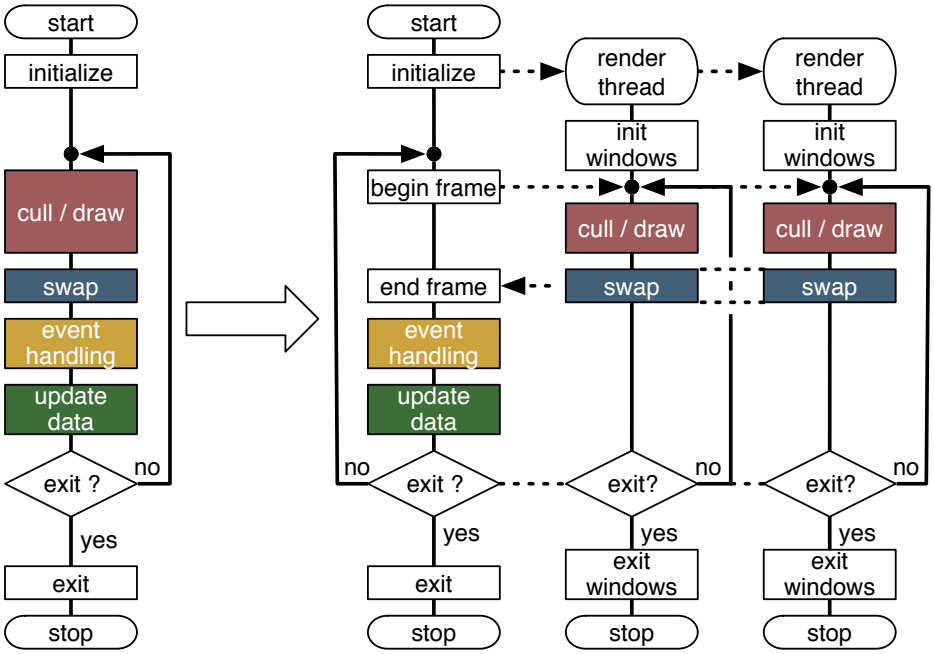


Figure 3.1: *Simplified execution flow of a classical visualization application and an Equalizer application.*

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.

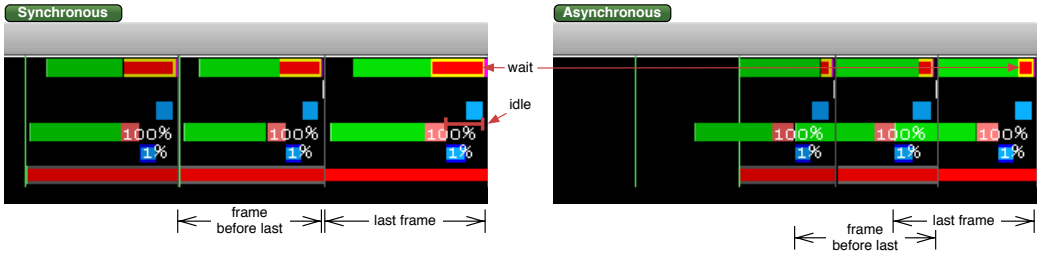


Figure 3.2: *Synchronous and Asynchronous Execution*

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.

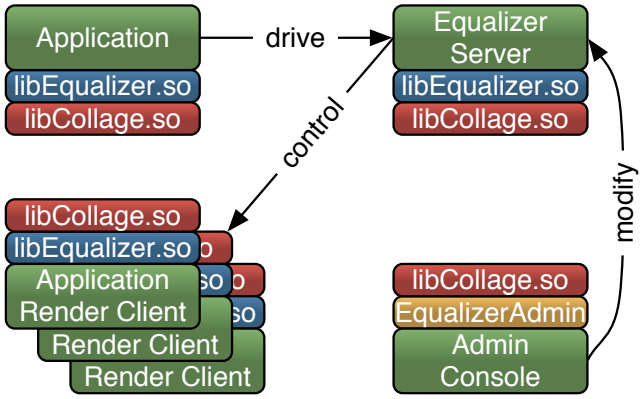


Figure 3.3: *Parallel Rendering Entities*

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.

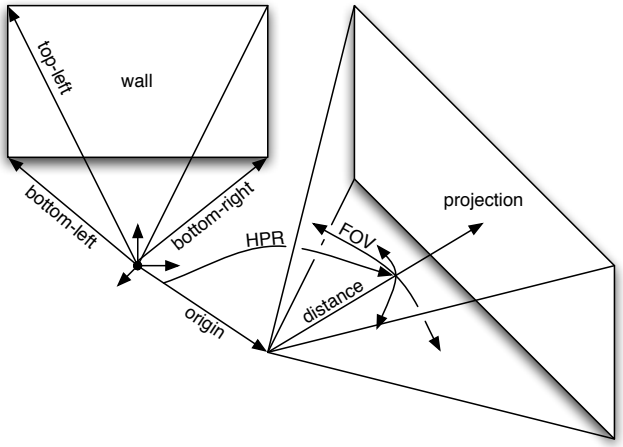


Figure 3.4: *Wall and Projection Parameters*

A canvas consists of one or more segments. A planar canvas typically has a frustum description (see Section **??**), 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.

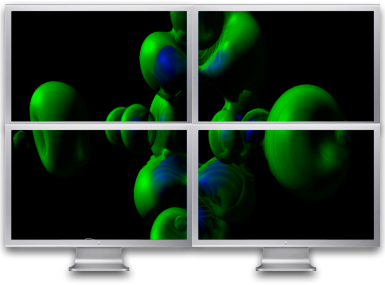


Figure 3.5: *A Canvas using four Segments*

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.

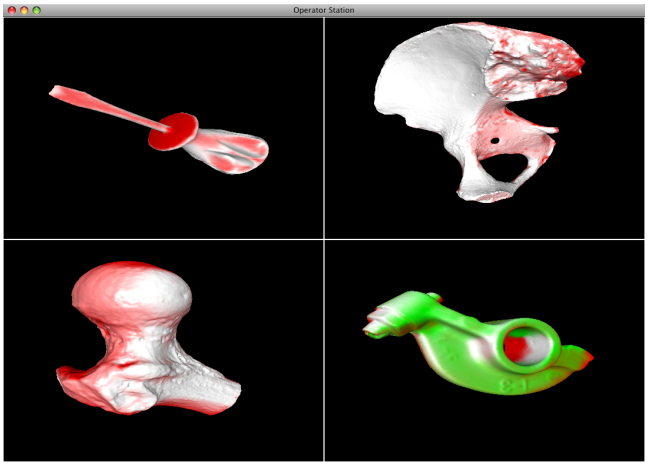


Figure 3.6: *Layout with four Views*

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 3.7 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: *Tiled 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 rendering performance. They are the core contribution enabling a flexible resource configuration.

Compounds are a data structure to describe the parallel execution of rendering tasks in the form of a tree. Each compound corresponds to some rendering tasks (clear, draw, assemble, readback) and references a channel from the resource description which executes the tasks in the given order. A compound may provide output frames from the readback task to others, and can request input frames from others for its own assembly task, and output frames are linked to input frames by name.

Compound trees are a logical description of the rendering pipeline, and only reference the actual physical resources through their channels. This allows mapping a compound tree to different physical configurations by simply replacing the channel IDs. For example, one can test the functionality of a sort-last configuration by using channels of different windows on one local workstation before deploying it to multiple physical GPUs.

A simple leaf compound description for rendering a part of the data set, given by the data *range*, into a particular region of the *viewport* is given in Figure 3.8. The data range is a logical mapping of the data set onto the unit interval and is left to the application to interpret appropriately. Hence the range [0 $\frac{1}{2}$] indicates that the first half of the data set should be rendered, for example the first $\frac{n}{2}$ triangles of a polygonal mesh with n faces. The viewport is indicated by the parameters [x y width height] as fraction of the parent's viewport, and in the example the data is thus rendered into the left half of the viewport. The resulting framebuffer data – including per-pixel color and depth – of the rendering executed on this channel is read back and made available to other compounds by the name left_half.

```
compound {
  channel "draw"
  buffer [ COLOR DEPTH ]
  range [0 1/2]
  viewport [ 0 0 1/2 1 ]
  outputframe {name "left_half" }
}
```

Figure 3.8: *Compound executing rendering of a part of the data set into a given region of the viewport.*

A non-leaf compound performing some image assembly and compositing task is indicated in Figure 3.9. Framebuffer data is read from two other compounds which supposedly execute rendering for part_a and part_b of the data set in parallel. The compound itself executes for example z-depth visibility compositing of the two input images on its channel and returns the resulting color framebuffer.

```
compound {
  channel "display"
  inputframe { name "part_a" }
  inputframe { name "part_b" }
  outputframe { buffer [ COLOR ] }
}
```

Figure 3.9: *Compound performing image compositing.*

Leaf compounds execute all tasks by default, but the focus is often on the draw task with a default assemble and standard readback task used to pass the resulting image data on to other compounds for further compositing. Hence while leaf compounds execute the rendering in parallel, non-leaf compounds often correspond to, but are not restricted to the (parallel) image compositing and assembly part. The readback or assemble tasks are only active if output or input frames have

been specified, respectively. Otherwise the rendered image frame is left in-place for further processing in a parent compound sharing the same channel.

Note that non-leaf nodes in the compound tree structure traverse their children first before performing their default assemble and readback tasks. Furthermore, compounds only define the logical task decomposition structure, while its execution is actually performed on the referenced channels. Therefore, since compounds can share channels, as often done between a parent and one of its child compounds, rendered image data can sometimes be left in place, avoiding readback and transfer to another node.

All attributes as well as the channel are inherited from the parent compound if not specified otherwise. The *viewport*, *data range* and *eye* attributes are used to describe the decomposition of the parent's 2D viewport, database range and eye passes, respectively.

A more formal classification of compound entities is:

Root compound is the top-level compound of a compound tree. It might also be a destination compound, or can be empty (not referencing a channel) when synchronizing multiple destination channels.

Destination compound(s) are the top-most compound referencing a destination channel. This destination channel determines the rendering context for the whole subtree, that is, compounds and their channels lower in the hierarchy contribute to the rendering of the destination channel be executing part of the destination render context and providing output frames which will eventually be composited onto the destination channel.

Source compounds are the leaf nodes in a compound tree. They typically use a different channel from the destination channel and configure scalability by overriding render context parameters. This decomposes the rendering of the destination channel. By adding output and input frames, the partial results are collected and composited:

- **Decomposition** On each child compound the rendering task of that child can be limited by setting the viewport, range, period an phase, pixel, subpixel, eye or zoom as desired.
- **Compositing** Source compounds define an output_frame to read back the result. This output frame is used as an input_frame on the destination compound receiving the pixels. The frames are connected with each other by their name, which has to be unique within the root compound tree. For parallel compositing, the algorithm is described by defining multiple input and output frames across all source compounds.

Intermediate compounds may be used to simplify the task decompostion or to configure parallel compositing.

3.4 Compositing

In contrast to most other parallel rendering frameworks, Equalizer decouples the compositing algorithm from the task decomposition. This is a critical component in the architecture, allowing a flexible configuration, often in many unforeseen ways. The compound tree with its task decomposition, input and output frames, is a specialized description "programming" scalable rendering across parallel resources.

Compositing if configured using output frames connected to input frames, compound tasks and eye passes, as well as frame parameters. In its simplest form, a sort-first source compound provides and output frame, which is routed to an input frame using the same name on the destination compound. The source viewport decomposes the task, and the output frame collects this partial result from the source channel and composites it using the correct offset onto the destination channel.

Frame parameters allow the customization of the pixel handling. They include the transferred buffers (color, depth), partial channel viewport, and pixel zooming (upscale and downscale). An output frame may be connected to multiple input frames. These parameters are used in conjunction with compound tasks for parallel compositing, and advanced features such as monitoring and dynamic frame resolution introduced later.

Figure 3.10 shows the pixel flow of direct send composting for a three-way sort-last decomposition, where the destination channel also contributes to the rendering. In the first step, each channel exchanges two color+depth tiles with its neighbors, and then z-composites its own tile. This yields one complete tile on each channel, of which two color tiles are then assembled on the destination channel.

The corresponding compound tree is shown schematically on the right of Figure 3.10. Each of the three channels has a child compound to execute the rendering and read back to incomplete tiles for sort-last compositing. These three leaf compounds represent the first step in the left-hand diagram. One level up, each channel receives two tiles and assembles them onto its partially rendered result, creating a complete tile. For the two source-only channels, a final color-only output image is connected to the destination channel. The arrows illustrate the pixel flow for one of the tiles.

For most rendering applications even a relatively complex setup such as this example requires very little programmer involvement. The abstractions provided by the resource description, render context and compounds enable Equalizer to reconfigure the application almost transparently. For polygonal rendering, it suffices that the application honors the range parameter of the render context to decompose its rendering. All other tasks, in particular the parallel compositing

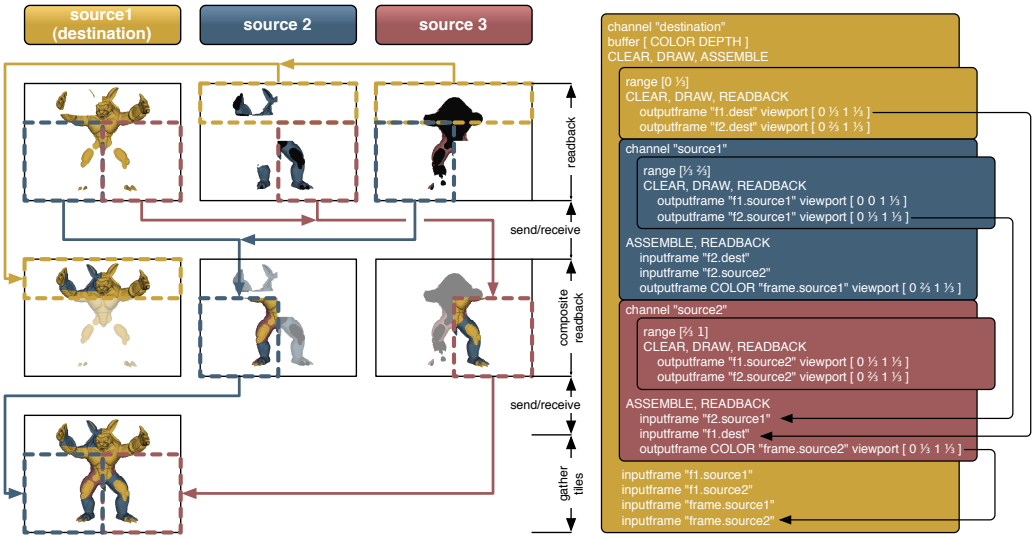


Figure 3.10: *Direct Send Compositing*

and pixel transfers, are fully handled by Equalizer. Applications which require ordered compositing, for example volume rendering, overwrite the assemble callback to reorder the input frames correctly.

The abstraction through frames is flexible, but still allows many architectural optimizations:

Unordered Compositing: Unless overwritten by the application, Equalizer will composite all input frames in the order they become available, not in the order they are configured. In the example from Figure 3.10, the destination channel will assemble its four input frames one by one as the output frame is received. Due to the asynchronous execution and resulting pipelining of operations, this availability changes each frame depending on the runtime of other tasks.

Asynchronous Compression, GPU and Network Transfers: Compressing, transmitting and receiving frames between nodes is handled by threads independent from the render thread. GPU transfers are handled by asynchronous PBO transfers. Pipelining all these operations with the actual rendering significantly minimizes the compositing overhead.

On-GPU Transfers: Occasionally, the source and destination channel are on the same GPU. Using textures as pixel buffers allows minimal overhead for the output to input frame transfer.

3.5 Load Balancing

Compounds in itself only provide a static configuration of the parallel rendering setup. Equalizers are algorithms which hook into a compound and modify parameters of their respective subtree at runtime to dynamically optimize the resource usage, by each tuning one aspect of the decomposition. Due to their nature, they are transparent to application developers, but might have application-accessible parameters to tune their behavior. Resource equalization is the critical component for scalable parallel rendering, and therefore the eponym for the Equalizer project name.

Compounds are a static snapshot of a configuration, and equalizers provide dynamic configuration on top. This separation of responsibilities is an important architectural component of our parallel rendering framework. Load balancing is one use case for equalizers, where the algorithm analysis runtime statistics to change the task decomposition each frame to equalize rendering times between all resources.

3.6 Distributed Execution Layer

An important part of writing a parallel rendering application is the communication layer between the individual processes. Equalizer relies on the Collage network library for its internal operation. Collage provides networking functionality of different abstraction layers, gradually providing higher level functionality for the programmer. The main primitives in Collage and their relations are shown in Figure 3.11 and provide:

Connection A stream-oriented point-to-point communication line. Connections transmit raw data reliably between two endpoints for unicast connections, and between a set of endpoints for multicast connections. For unicast, process-local pipes, TCP and Infiniband RDMA are implemented. For multicast, a reliable, UDP-based protocol is discussed in Section 7.3.1.

DataI/OStream Abstracts the marshalling of C++ classes from or to a set of connections by implementing output stream operators. Uses buffering to aggregate data for network transmission. Performs byte swapping during input if the endianness differs between the remote and local node.

Node and LocalNode The abstraction of a process in the cluster. Nodes communicate with each other using connections. A LocalNode listens on various connections and processes requests for a given process. Received data is wrapped in ICommands and dispatched to command handler methods. A Node is a proxy for a remote LocalNode.

Object Provides object-oriented, versioned data distribution of C++ objects between nodes. Objects are registered or mapped on a LocalNode.

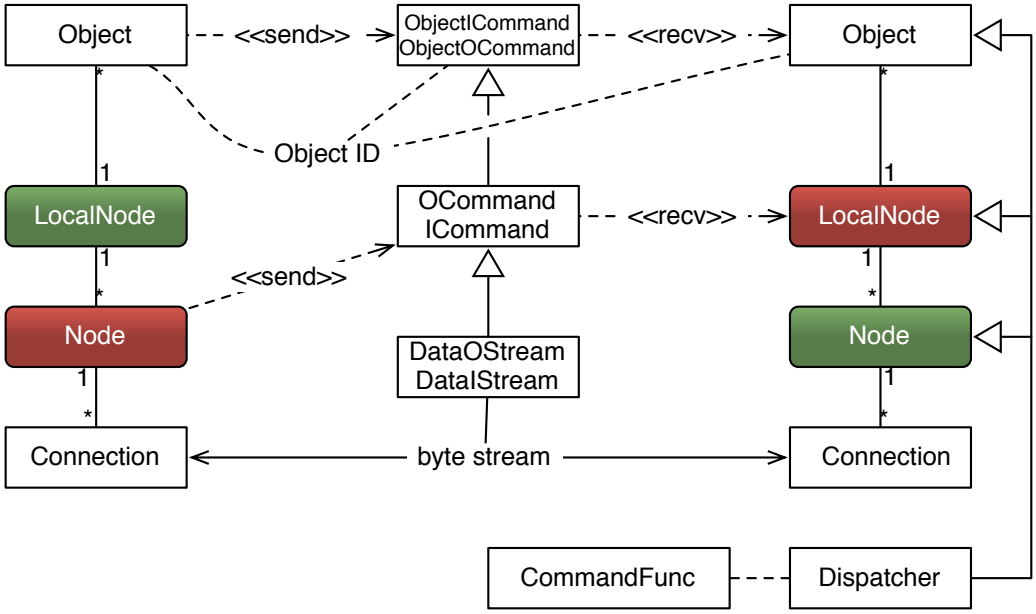


Figure 3.11: *Communication between two Collage objects*

Collage implements a few generic distributed objects which are used by Equalizer and other applications. A barrier is a distributed barrier primitive used for software swap synchronization. Its implementation follows a simple master-slave approach, which has shown to by sufficient for this use case. Queues are distributed, single producer, multiple consumer FIFO queues. To hide network latencies, consumers prefetch items into a local queue. Queues are used for tile and chunk compounds (Section **??**).

An object map facilitates distribution and synchronization of a collection of distributed objects. Master versions can be registered on a central node, e.g., the application node in Equalizer. Consumers, e.g., Equalizer render clients, can selectively map the objects they are interested in. Committing the object map will commit all registered objects and sync their new version to the slaves. Syncing the map on the slaves will synchronize all mapped instances to the new version recorded in the object map. This effective design allows data distribution with minimal application logic.

CHAPTER 4

SCALABLE RENDERING

Scalable rendering is a subset of parallel rendering which aims to improve the framerate of a rendering task by decomposing it over multiple rendering resources. Parallel rendering includes other use cases, for example to use multiple GPUs to drive individual displays of a tiled display wall.

Parallel rendering research has put a lot of focus on two of the three architectures classified by [Molnar et al., 1992]: sort-first and sort-last rendering. Sort-middle rendering is still largely confined to hardware implementations due to its high communication cost of sorting and distributing fragments to processing units.

In this chapter, we present a wide set of new parallelization schemes which break out of this standard classification. For each mode, we present its algorithm and implementation, potential impact on the application code, as well as its strengths and weaknesses. Due to the flexible architecture of our parallel rendering system, these modes are largely usable with any Equalizer application and can be combined with all other modes.

Most of these rendering modes are similar to sort-first rendering, in that they decompose the final view spatially or temporally. Stereo compounds decompose per eye pass, DPlex compounds temporally, pixel and subpixel compounds use equal spatial decompositions. Finally, tile and chunk compounds implement implicit load-balancing for sort-first and sort-last rendering using queueing of work items.

This wide set of decomposition modes for scalable rendering, embedded in our generalized compound structure, enables applications and researchers to decom-

pose the rendering task in, as far as we know, any way possible for a rendering pipeline. To our knowledge no other implementation provides this breath and flexibility, and many of these algorithms appear for the first time in Equalizer.

4.1 Stereo

Stereo, or eye decomposition, is a specialized version of sort-first rendering. Two GPUs get assigned each one of the eye passes during stereo rendering. For passive stereo, there is no compositing step needed, whereas for active and anaglyphic stereo the frame buffer for one eye pass has to be copied to the destination channel. Due to the strong similarity between both eye passes, this mode provides close to perfect load balance. Figure 4.1 shows an anaglyphic stereo compound.

While many implementations provide passive or active stereo rendering and sometimes decomposition using two GPUs, our implementation within our flexible compound structure allows to fully exploit stereo decomposition. Stereoscopic and monoscopic rendering is no special case in the architecture, but rather a configuration of the rendering resources. Among other things, this allows extending a two-way stereo decomposition with further resources and any other scalable rendering mode. One can also easily set up an application with mixed rendering, e.g., to render a monoscopic view on a control workstation while rendering stereoscopic on a larger immersive installation.

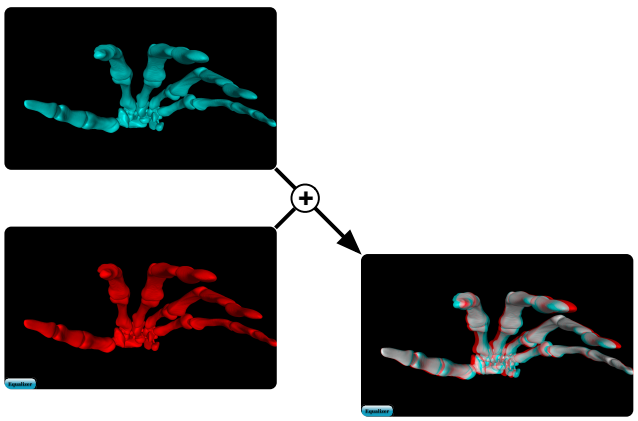


Figure 4.1: *Anaglyphic Stereo Compound*

4.2 Time-Multiplex

Time-multiplexing distributes full frames over the available resources, such that each resource only renders a subset of the visible frames (Figure 4.2). This mode is also called alternate frame rate or DPlex, was first implemented in [Bhaniramka et al., 2005] for shared memory machines. The algorithm is however much better suited for distributed memory systems, since the separate memory space makes

concurrent rendering of different frames much easier to implement. While it increases the framerate almost linearly, it does not decrease the latency between user input and the corresponding output. Consequently, this decomposition mode is mostly useful for non-interactive movie generation.

DPlex is very well load-balanced, since most applications observe a strong frame-to-frame coherence with respect to the rendering load. This decomposition mode has the pecularity that small imbalances tend to accumulate such that the concurrent frames all finish simultaneously. To provide a smooth framerate, if so desired, a framerate equalizer can be installed on the destination compound. Section 6.4 covers this functionality. It is transparent to Equalizer applications, but does require the configuration latency to be equal or greater than the number of source channels.

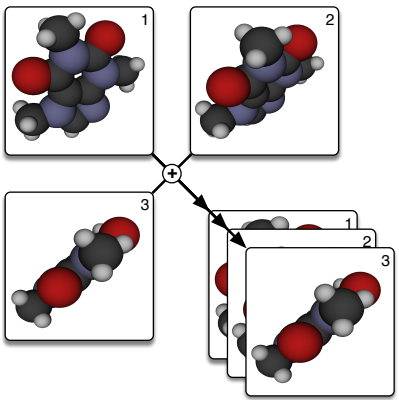


Figure 4.2: *Time-Multiplex Compound*

DPlex rendering is not hard-coded into our framework, but configured by restricting the rendering task temporally on each source compound. This is achieved by setting a period and phase parameter, which configure the number of frames skipped and starting offset on the given source compound. A simple DPlex compound would have a destination compound with n source compounds, where each source has a period of n and one phase from $0..n-1$. While this generalization may seem artificial, it opens up different use cases, for example giving a fast GPU a smaller period, thus giving it more work.

4.3 Pixel

Pixel compounds (Figure 4.3) decompose the destination channel by interleaving pixels in image space. They are a variant of sort-first rendering which works well for fill-limited applications which are not geometry bound, for example direct volume rendering. Source channels cannot reduce geometry load through view frustum culling, since each source channel has almost the same frustum. However, the fragment load on all source channels is reduced linearly and well load-balanced due to the interleaved

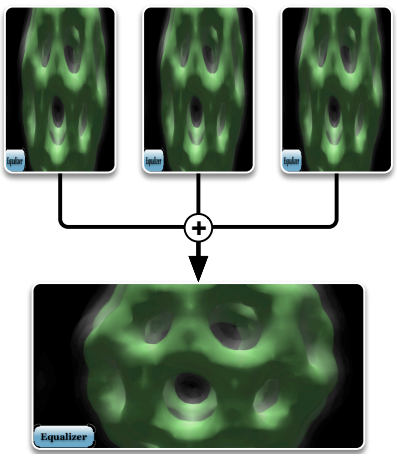


Figure 4.3: *Pixel Compound*

distribution of pixels. This functionality is transparent to Equalizer applications, and the default compositing uses the stencil buffer to blit pixels onto the destination channel.

Pixel compounds are configured by restricting each source compound with a pixel kernel. The kernel describes the size of the decomposition in 2D space, and the 2D pixel offset within this region. This follows our design philosophy of enabling features by generalizing the underlying algorithm rather then hardcoding them.

4.4 Subpixel

Subpixel compounds (Figure 4.4) are similar to pixel compounds, but they decompose the work for a single pixel, for example with Monte-Carlo ray tracing, FSAA or depth of field rendering. The default compositing algorithm uses accumulation and averaging of all computed fragments for a pixel. Like Pixel compoungds, this mode is naturally load-balanced on the fragment processing stage but cannot scale geometry processing. This feature is not fully transparent to the application, since it decomposes rendering algorithms which render multiple samples per pixel. Applications needs to adapt their rendering code, for example to jitter or tilt the frustum based on the subpixel executed in the current rendering pass.

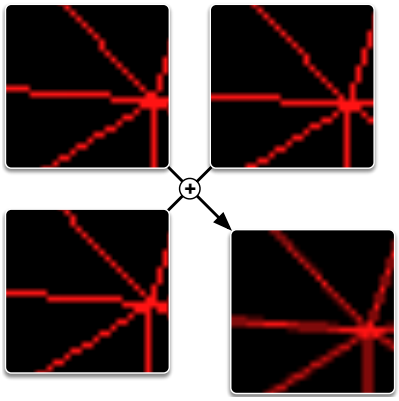


Figure 4.4: *Subpixel Compound*

4.5 Tiles and Chunks

Tile (Figure 4.5) and chunk decompositions are a variant of sort-first and sort-last rendering, respectively. They decompose the scene into a predefined set of fixed-size image tiles or database ranges. These tasks, or work packages, are queued and processed by all source channels by polling a server-central queue. Prefetching ensures that the task communication overlaps with rendering. As shown in [Steiner et al., 2016], these modes can provide better performance due

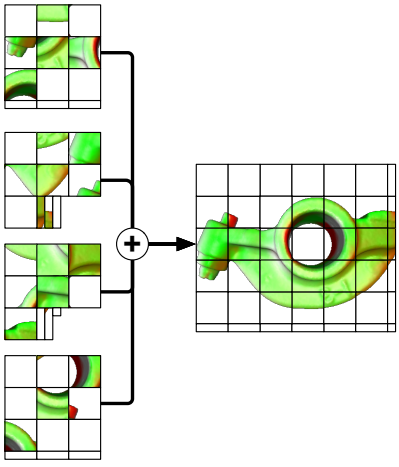


Figure 4.5: *Tile Compound*

to being implicitly load-balanced, as long as there is an insignificant overhead for the render task setup. This mode is transparent to Equalizer applications. We have used a tile compound to scale an interactive ray tracing application to hundreds of rendering nodes.

4.6 Mixed Mode Compounds

A major contribution of our parallel rendering system is the flexible system architecture. While many applications and frameworks implement a subset of the features mentioned above, all of them hardcode the algorithms to a certain degree, predetermining the number of possible configurations. In Equalizer, both the decomposition and the recomposition of the rendering task are derived through a number of orthogonal parameters, which are easily combined to configure common scalable renderig modes. For advanced usage, they can also be configured for many other use cases. We have seen many interesting and unforeseen configurations, for example:

- Reusing the period parameter used to configure number of frames in a DPlex compound, an underpowered control workstation for a large tiled display wall was configured to render only every other frame using a period of two. Due to the standard latency of one frame, this meant that the display wall rendering became the bottleneck, but could render at a substantially higher framerate than before.
- Rerouting one of the eye passes of a head-mounted display to a large display using an output and input frame, external users could observe the interaction and view of the person using the HMD. The same can be achieved by mirroring the video signal by other means, but this was not available on the given setup.
- Using combined stereo and sort-first decomposition on the central tiles of a tiled display wall. Oftentimes the central tiles of a tiled display wall receive a higher rendering load then the outer tiles. In this particular configuration, each tile was driven by a dual-GPU node using active stereo compounds, and the middle segments where given an additional machine setting up a two-way sort-first decomposition under each node of the two-way stereo compound.

4.7 Stereo-Selective Compounds

Each compound sub-tree can restrict the eye passes it renders from the default left, right, cyclop passes. Depending on the active stereo mode (stereo or mono), restricted compound trees may be skipped or activated. This is used on one hand to configure stereo compounds, but may also be used to configure different decompositions depending on the stereo mode. Figure 4.6 shows a simple example: A dual-GPU setup is used with eye-parallel rendering during stereo rendering, and a standard sort-first parallel rendering during monoscopic rendering. Note that the rendering mode is runtime-configurable, that is, the application can switch the view from monoscopic to stereoscopic rendering at any time, activating and deactivating the configured compounds and attached resources. It is also possible to configure a different set of resources (nodes and GPUs) per stereo mode, triggering the launch and exit of render client processes during the stereo switch.

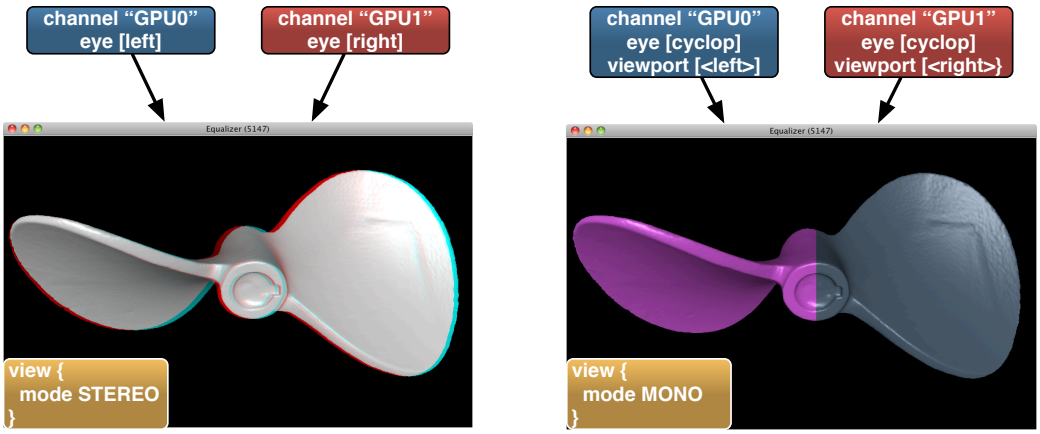


Figure 4.6: *Stereo-Selective Compound*

CHAPTER 5

COMPOSITING OPTIMISATIONS

5.1 Region of Interest

During scalable rendering using any decomposition most of the time pixel data has to be copied from the source channel framebuffer to either the destinationation channel framebuffer, or to an intermediate channel during parallel compositing. The distributed image compositing cost is directly dependent on how much data has to be sent over the network, which in turn is related to how much screen space is actively covered. Depending on the data set and viewpoint, only a subset of the framebuffer shows pixeles generated from the data. In particular for sort-last rendering, every node potentially renders into the entire frame buffer. With an increasing number of nodes the set of affected pixels typically decreases, leaving blank areas that can be omitted for transmission and compositing.

Equalizer provides an API for the programmer to provide the region of interest, i.e., the screen-space 2D bounding box enclosing the data rendered by a single resource. We have extended the core parallel rendering framework to use an application-provided ROI to optimize the load_equalizer(Section **??** as well as the image compositing. The compositing code uses the ROI to minimize image readback size, and consequently network transmission. The ROI is an output frame parameter, and is transmitted to all input frames together with the pixel data. On the input frame, the compositing code respects this parameter to place the pixel data on the right position. The application can easily compute a good

ROI estimation by computing the screen-space projection of the model's bounding sphere or bounding box.

In [Makhinya et al., 2010], this application-provided ROI was extended with an algorithm which automatically computes the ROI by analyzing the framebuffer to maximize the amount of blank framebuffer omitted during compositing.

In [Eilemann et al., 2012] we show that using ROI can quadruple the rendering performance for the costly compositing step in sort-last rendering. This is especially the case when the application rendering code selects a spatially compact subset during sort-lsat decomposition. For sort-first rendering, ROI allows the algorithm to scale to a higher number of resources before synchronization and compositing overhead eats up any potential speedup due to the task decomposition.

5.2 Asynchronous Compositing

Compositing in a distributed parallel rendering system is decomposed into readback of the produced pixel data (1), optional compression of this pixel data (2), transmission to the destination node consisting of send (3) and receive (4), optional decompression (5) and composition consisting of upload (6) and assembly (7) in the destination framebuffer.

In a naive implementation, operations 1 to 3 are executed serially on one processor, and 4 to 7 on another. In our parallel rendering system, operations 2 to 5 are executed asynchronously to the rendering and operations 1, 6 and 7. Furthermore, we use a latency of one frame which means that two rendering frames are always in execution, allowing the pipelining of these operations, as shown in Figure 5.1(a). We have implemented asynchronous readback using OpenGL pixel buffer objects, further increasing the parallelism by pipelining the rendering and pixel transfers, as shown in Figure 5.1(b).

In the asynchronous case, the rendering thread performs only application-specific rendering operations, since the overhead of starting an asynchronous readback becomes negligible. Equalizer uses a plugin system to implement GPU-CPU transfer modules which are runtime loadable. We extended this plugin API to allow the creation of asynchronous transfer plugins, and implemented such a plugin using OpenGL pixel buffer objects (PBO). At runtime, one rendering thread and one download thread is used for each GPU, as well as one transmit thread per process. The download threads are created lazy, when needed.

Asynchronous compositing is, together with ROI, one of the most influential optimizations for scalable rendering. We have shown in [Eilemann et al., 2012] that when being mostly rendering bound, pipelining the readback with the rendering yields a performance again of about 10%. At higher framerates, when the

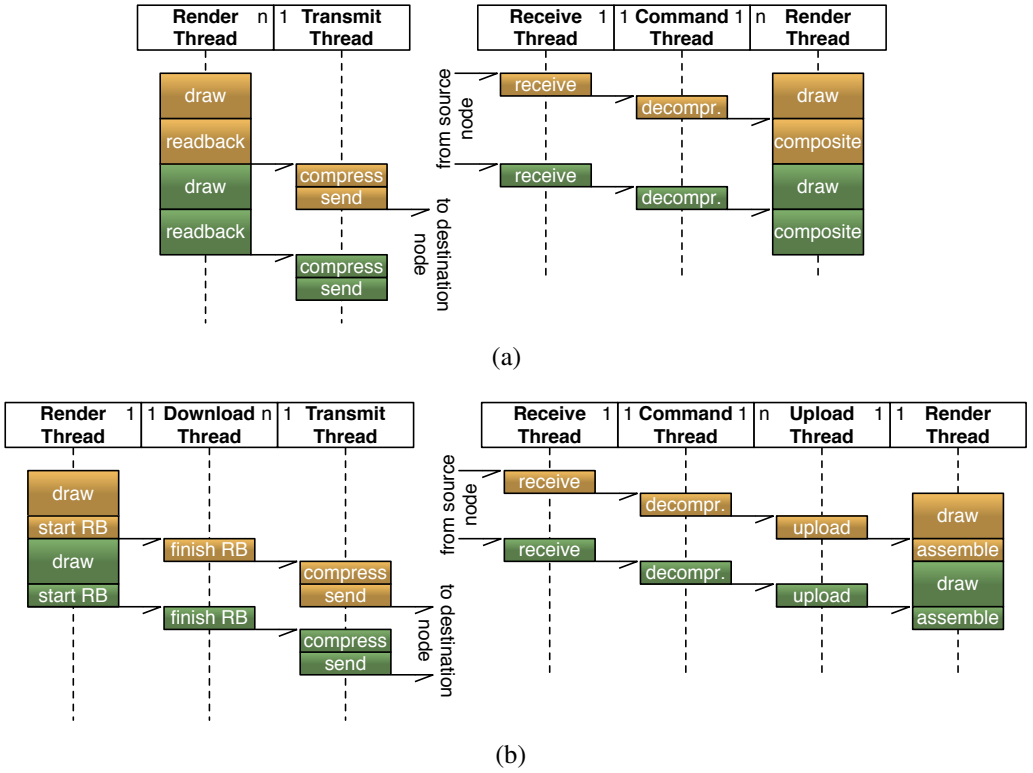


Figure 5.1: *Synchronous readback and upload (a) as well as asynchronous readback and upload (b) for two overlapping frames.*

rendering time of a single resource decreases, asynchronous readback has even a higher impact, of over 25% for large node counts.

5.3 Compression for Image Compositing

The image compositing stages in distributed rendering for gathering and combining partial rendering results into a final display frame are fundamentally limited by the GPU-to-node and node-to-node image throughput. Therefore, efficient image coding, compression and transmission must be considered to minimize that bottleneck.

Basic run-length encoding (RLE) has been used as a fast standard to improve network throughput for interactive image transmission. However, it only gives sufficient results in specific rendering contexts and fails to provide a general improvement as shown in [Makhinya et al., 2010]. RLE only works to compact large empty or uniform color areas but is often useless for non-trivial full frame color

results. We developed two enhancements to improve this, per-component RLE compression and *swizzling* of color bits.

Equalizer also implements more complex compression algorithms such as libjpeg-turbo, which are however of little practical use on modern interconnects. Their compression overhead is often to high to be amortized by the decreased network transmission time on modern 10 GBit/s or faster interconnects.

[Makhinya et al., 2010] implemented GPU-based YUV subsampling before the image download, which has negligible overhead, reasonable compression artefacts, and a good compression ratio.

5.3.1 Enhanced RLE Compression

Our first RLE method used a fast 64-bit version that compares two pixels at the same time (8-bit per channel RGBA format). While this method is very fast it shows poor compression results in most practical settings since it can only compress adjacent pixels of the same color.

The first improvement is to treat each color components separately by producing up to four RLE-compressed output streams as illustrated in Figure 5.2. While the basic RLE compression yields an up to 10% compression rate in practical scenarios, per-component RLE improves this to up to 25% since individual color components change less often than full pixels (based on results from [Makhinya et al., 2010]).

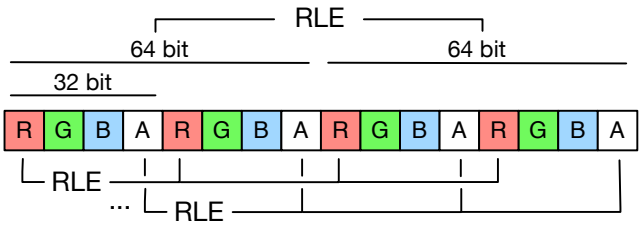


Figure 5.2: *Comparison of 64-bit and per-component RLE.*

A second improvement is *bit-swizzling* of color values before per-component compression. The swizzling step is a data pre-conditioner, which reorders and interleaves the per-component bits as shown in Figure 5.3. Now the per-component RLE compression separately compresses the higher, medium and lower order bits in separate streams, thus achieving stronger compression for smoothly changing color values since high-order bits tend to change less often.

Swizzling improves the compression rate to up to 40% for the same scenario as above. Since the preconditioning step only requires bit shift and mask operations, it is entirely executed in registers and has no measurable impact on performance, since the whole algorithm is memory bound on modern CPUs.

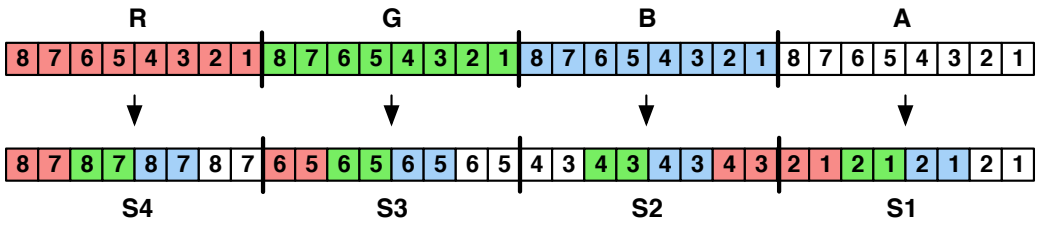


Figure 5.3: *Swizzling scheme for reordering bits of 32-bit RGBA values.*

5.3.2 GPU Transfer and CPU Compression Plugins

Equalizer uses runtime-loadable plugins to transfer pixel data from and to the GPU, as well as plugins to compress and decompress pixel data for network compression. This separation allows different code paths for multi-GPU machines where no CPU-based compression is used, and for distributed execution where data is compressed before transfer.

The GPU transfer might also apply compression. This is typically done on the GPU to reduce the amount of memory transferred. One example is YUV subsampling, where a shader implements chroma subsampling. Furthermore, a GPU transfer plugin might implement asynchronous downloads, where the download is started from the render thread an finished in a separate download thread as shown in Figure 5.1(b). CPU compression plugins are always executed from asynchronous threads concurrently to the rendering threads.

The implementation of these steps in plugins provides a clean separation and interface for users and researchers interested in experimenting with image compression for interactive parallel rendering.

CHAPTER 6

LOAD BALANCING

Equalizers are an addition to compound trees. They modify parameters of their respective compound subtree at runtime to dynamically optimize the resource usage, by each tuning one aspect of the decomposition or recomposition. Due to their nature, they are transparent to application developers, but might have application-accessible parameters to tune their behavior. Resource equalization is the critical component for scalable parallel rendering, and therefore the eponym for the Equalizer project name.

In this section we present various *equalizer* implementations: two variants for load balancing for sort-first and sort-last rendering, work packages for sort-last and sort-first rendering, cross-segment load-balancing for multi-display installations, constant frame rate rendering using dynamic frame resolution, and monitoring of large-scale visualization systems.

6.1 Sort-First and Sort-Last Load Balancing

Sort-first (Figure 6.1) and sort-last load balancing are the most obvious optimizations for these parallel rendering modes. Our load equalizers are fully transparent for application developers; that is, they use a reactive approach based on past rendering times. This requires a reasonable frame-to-frame coherence for good results, which is the case with most rendering applications. Equalizer implements two different algorithms, a load_equalizer and a tree_equalizer, which have shown to each have their advantage depending on the type of rendering load.

The load_equalizer builds a model of the rendering load in screen space or data space. It stores a 2D (for sort-first) or 1D (for sort-last) grid of the load, mapping the load of each channel. The load is stored in normalized 2D/1D coordinates using $\frac{time}{area}$ as the load, the contributing source channels are organized in a binary kD-tree, and then the algorithm balances the two branches of each level by equalizing the integral over the cost area map on each side. This algorithm is similar to [Abraham et al., 2004], which uses a dual-level tree, first decomposed into horizontal tiles and then each horizontal tile into vertical sub-tiles. Our binary split tree provides more compact tiles.

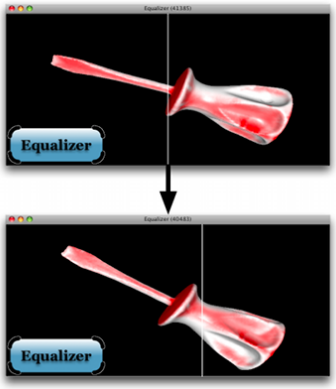


Figure 6.1: *Load-Balancing*

The load-balancer has to operate on the assumption that the load is uniform within one load grid tile. This leads naturally to estimation errors, since in reality the load is not uniformly distributed, e.g., it tends to increase towards the center of the screen in Figure 6.2. We reuse the region of interest (ROI) of each source channel to automatically refine the load grid as shown Figure 6.2. In cases where the rendered data projects only to a limited screen area, this ROI refinement provides the load-balancer with a more accurate load indicator leading to a better load prediction.

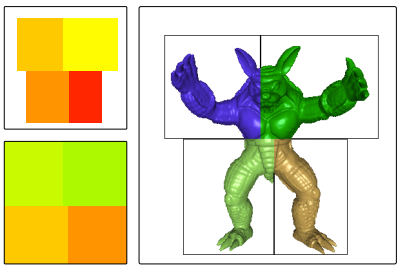


Figure 6.2: *Load distribution areas with (top) and without (bottom) using the ROI of the right-hand sort-first parallel rendering.*

The tree_equalizer uses the same binary kD-tree structure as the load_equalizer for recursive load balancing. It computes the accumulated render time of all children for each nodee of the tree, and uses this to allocate an equal render time to each subtree. It makes no assumption of the load distribution in 2D or 1D space, it only tries to correct the imbalance in render time.

Both equalizers implement tuneable parameters allowing application developers to optimize the load balancing based on the characteristics of their rendering algorithm:

Split Mode configures the tile layout: horizontal or vertical stripes, and 2D, a binary tree split alternating the split axis on each level, resulting in compact 2D tiles.

Damping reduces frame-to-frame oscillations. The equal load distribution within the region of interest assumed by the load balancers is in reality not equal,

causing the load balancing to overshoot. Damping is a normalized scalar defining how much of the computed delta from the previous position is applied to the new split.

Resistance eliminates small deltas in the load balancing step. This might help the application to cache visibility computations since the frustum does not change each frame.

Boundaries define the modulo factor in pixels onto which a load split may fall. Some rendering algorithms produce artefacts related to the OpenGL raster position, e.g., screen door transparency, which can be eliminated by aligning the boundary to the pixel repetition. Furthermore, some rendering algorithms are sensitive to cache alignments, which can again be exploited by chosing the corresponding boundary.

In [Eilemann et al., 2018] we have shown that the simpler tree_equalizer outperforms the load-grid-driven load_equalizer in most cases, except for sort-first volume rendering where the load in the region of interest is relatively uniform. This counterintuitive result seems to again confirm that simple algorithms often outperform theoretically better, but more complex implementations. The decoupling of the load balancing algorithm from the rest of the system enabled by the compound architecture opens the possibility for more research in this area.

6.1.1 Dynamic Work Packages

Load-balancing can be classified into explicit and implicit approaches, where explicit methods centrally compute a task decomposition up-front, before a new frame is rendered, while implicit methods decompose the workload into task units that can dynamically be assigned to the resources during rendering, based on the work progress of the individual resources. Explicit load-balancing can be reactive, based on load distribution in previous frames, or predictive, based on an application-provided cost function. Explicit load-balancing typically assigns a single task to each resource to minimize static per-task costs. The aforementioned equalizers implement explicit reactive load balancing.

Implicit load-balancing generally uses a finer granularity of many more task units than resources to minimize the load imbalance due to a fixed coarse task granularity. Implicit load-balancing uses central task distribution or distributed task stealing between resources. Implicit algorithms are more commonly used for off-line raytracing compared to real-time rasterization algorithms due to the practically non-existent per-tile cost in raytracing.

Our work package implementation uses a task pulling mechanism, an approach that has been employed before in distributed computing. Rather than having the server push tasks to the rendering clients, our dynamic work packages approach works by managing fine grained tasks on a server side queue, while the

clients request and execute the tasks as they become available. Every rendering client employs a local queue of work packages for caching purposes. During rendering, a client first works on packages from its local queue and concurrently requests packages from the server whenever the amount of available packages sinks below some threshold. In [Steiner et al., 2016], this basic, random task assignment was extended with client-affinity models.

Depending on the sort-first or sort-last decomposition mode, a tile_equalizer or chunk_equalizer generates tiles or database ranges of a configurable size and enqueues them at the beginning of each frame. This active equalizer is the central component of the tile and chunk compounds introduced in Section 4.5.

In [Steiner et al., 2016], we have shown that the tile_equalizer often outperforms tree_equalizer. This suggests that the underlying implicit load balancing of task queues can be superior to the explicit methods of load_equalizer and tree_equalizer in high load situations, where the additional overhead of tile generation and distribution is more justified. The relatively simple nature of our benchmark application's rendering algorithms is also favoring work packages, since they have a near-zero static overhead per rendering pass.

6.2 Cross-Segment Load Balancing

A serious challenge for all distributed rendering systems driving large multi-display installations is to deal with the varying rendering load per display, and thus the graphics load on its driving GPUs. Cross-segment load balancing (CSLB) is a novel dynamic load balancing approach to dynamically allocate n rendering resources to m output channels (with $n \geq m$), as shown in Figure 6.3.

The m output channels typically each drive a display or projector of a multi-display system. With CSLB, they are not required to be planar to each other, that is, the algorithm works equally for tiled display wall and immersive installations. Commonly, each destination channel is solely responsible for rendering and/or compositing of its corresponding display segment.

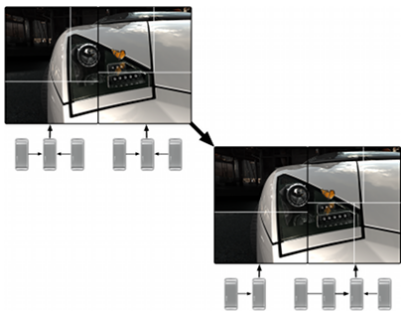


Figure 6.3: *Cross-Segment Load-Balancing*

A key element of CSLB is that the m GPUs physically driving the m display segments will not be restricted in a one-to-one mapping to rendering tasks of the corresponding display segment. In fact, CSLB performs dynamic assignment of n graphics resources from a pool to drive m different destination display segments, where the m destination channel GPUs themselves may also be part

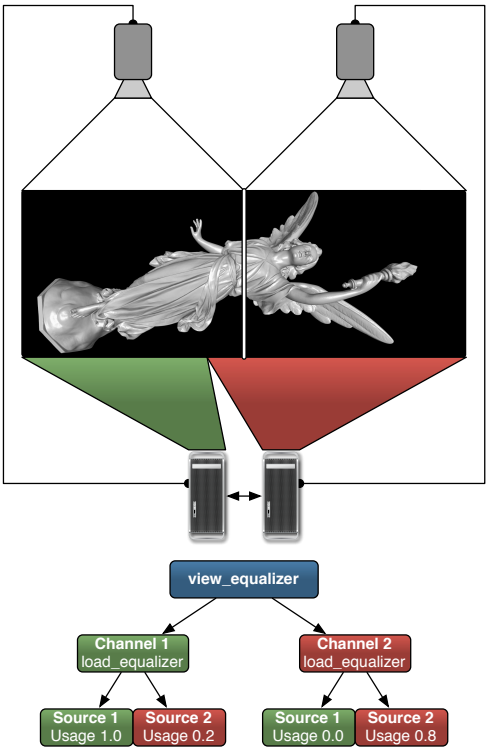
of the pool of graphics resources. Dynamic resource assignment is performed through load-balancing components that exploit statistical data from previous frames for the decision of optimal GPU usage for each segment as well as optimal distribution of workload among them. The algorithm can easily be extended to use predictive load-balancing based on a load estimation given the application.

CSLB is implemented in *Equalizer* as two layers of hierarchically organized components, specified in the configuration. Figure 6.4 depicts a snapshot of a simple CSLB setup along with its configuration file. Two destination channels, *Channel1* and *Channel2*, each connected to a projector, create the final output for a multi-projector view. Each projector is driven by a distinct GPU, constituting the source channels *Source1* and *Source2*. But each source channel GPU can contribute to the rendering of the other destination channel segment.

CSLB uses a two-stage approach: a view_equalizer is attached to the top level of the parallel rendering task decomposition compound hierarchy, and handles the resource assignment. Each child of this root compound has one destination channel, constituting one of the m display segments, with a load_equalizer or tree_equalizer each. Hence the view_equalizer supervises the different destination channels of a multi-display setup. The load_equalizer on the other hand is responsible for the partitioning of the rendering task among its child compounds. Therefore, each destination channel of a display segment has its source channel leaf nodes sharing the actual rendering load. One physical graphics resource (GPU), being assigned to a source channel, can be referenced in multiple leaf nodes and thus contribute to different displays. For performance reason, one resource is at most assigned two rendering tasks, e.g., to update itself and to contribute to another display.

The rendered data, viewing configuration and user interaction will at runtime produce different rendering loads for each segment of a multi-display system. As the slowest segment will determine the overall performance of the system, it is important to dynamically adjust load among segments. The view_equalizer analyzes the load of all segments based on past frame rendering statistics and adapts resource usage for each segment. For each rendered frame, the view_equalizer sets the usage of each leaf source channel compound, to activate or deactivate it for rendering on the given destination channel. This usage setting is picked up by the per-segment load balancers, which will only assign work proportionally to the usage, with no rendering task for 0 usage. [Erol et al., 2011] provides a detailed description of the algorithm.

Cross-segment load balancing thus allows for optimal resource usage of multiple GPUs used for driving the display segments themselves, as well as any additional source GPUs for rendering. It combines multi-display parallel rendering with scalable rendering for optimal performance. [Erol et al., 2011] provides experimental results for a six-monitor tiled display wall with up to 12 source GPUs,



(a) CSLB resources setup.

```
compound
{
  view_equalizer {}
  compound
  {
    channel "Channel1"
    load_equalizer{}
    compound {} # self
    compound
    {
      channel "Source2"
      outputframe {}
    }
    inputframe{}
    ...
  }
  compound
  {
    channel "Channel2"
    load_equalizer{}
    compound {} # self
    compound
    {
      channel "Source1"
      outputframe {}
    }
    inputframe{}
  }
  ...
}
```

(b) CSLB configuration file format.

Figure 6.4: *A dual-GPU, dual-display cross-segment load-balancing setup: For each destination channel, a set of potential resources are allocated. A top-level* view_equalizer *assigns the usage of each resource, based on which a per-segment* load_equalizer *computes the 2D split to balance the assigned resources within the display. The left segment of the display has a higher workload, so, both* Source1 *and* Source2 *are used to render for* Channel1*, whereas* Channel2 *makes use of only* Source2 *to assemble the image for the right segment.*

where CSLB can more then double the framerate over a static resource assignment. A strength of this algorithm lies in its flexibility. On one hand, it can perform dynamic resource assignment not only for a planar display system as some approaches which built a single virtual framebuffer for all output displays, but also for curved displays and CAVE installations. On the other hand, it allows a flexible assignment of potential contributing GPUs to each output channel individually, that is, each output may have a different, potentially overlapping, set of GPUs which may contribute to its rendering. [Erol et al., 2011] shows in its experiments that there is a sweet spot of the number of resources per channel.

6.3 Dynamic Frame Resolution

The DFR equalizer (Figure 6.5) provides a functionality similar to dynamic video resizing [Montrym et al., 1997], that is, it maintains a constant framerate by adapting the rendering resolution of a fill-limited application. While the aforementioned uses a now-obsolete hardware implementation, our implementation is purely in software.

Dynamic frame resolution (DFR) works by rendering into a source channel (typically on a FBO) separate to the destination channel, and then scaling the rendering during the transfer (typically through an on-GPU texture) to the destination channel. The DFR equalizer monitors the rendering performance and accordingly adapts the resolution of the source channel and the zoom factor for the source to destination transfer. If the performance and source channel resolutions allow, this will not only subsample, but also supersample the destination channel to reduce aliasing artefacts.

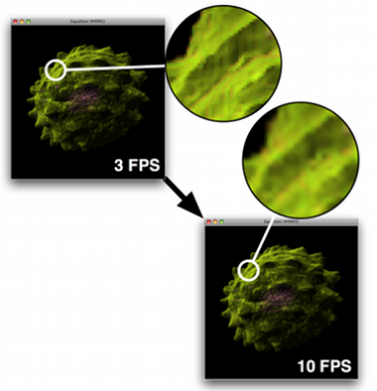


Figure 6.5: *Dynamic Frame Resolution*

As with all other features, it can be combined with other scalability features, e.g., sort-first rendering. It is also notable that it does not need any additional code in the core compound logic, it simply exploits existing functionality such as texture-based compositing frames and frame zoom with dynamic per-frame adjustments.

6.4 Frame Rate Equalizer

The framerate equalizer smoothens the output frame rate of a destination channel by instructing the corresponding window to delay its buffer swap to a minimum time between swaps. This is regularly used for time-multiplexed decompositions, where source channels tend to drift and finish their rendering unevenly distributed over time. This equalizer is however fully independent of DPlex compounds, and may be used to smoothen irregular application rendering algorithms. Due to the artificial sleep time before swap, it may incur a small performance penalty, but it greatly improves the perceived rendering quality for users in DPlex compounds.

6.5 Monitoring

The monitor equalizer (Figure 6.6, Figure 3.6) allows to reuse the rendering on another channel, typically for monitoring a larger setup on a control workstation. Output frames on the display channels are connected to input frames on a single monitoring channel. The monitor equalizer changes the scaling factor and offset between the output and input, so that the monitor channel has the same, but typically downscaled view, as the originating segments. While this is not strictly a scalable rendering feature, it optimizes resource usage by not needlessly rendering the same view multiple times.

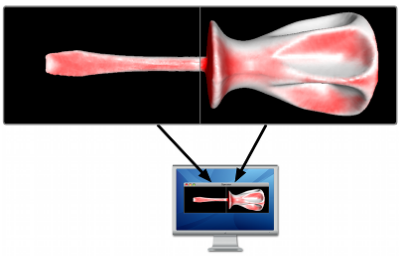


Figure 6.6: *Monitoring*

CHAPTER 7

DATA DISTRIBUTION AND SYNCHRONIZATION

Most research in parallel rendering ignores the problem of managing applications state in a distributed rendering setup. For research core scalability algorithms this problem is trivially to solve, whereas in real-world applications it often is one of the major challenges for using a distributed rendering cluster.

For this reason, we have spend a significant effort in researching, designing and implementing a distributed execution layer which is used by Equalizer and applications built on Equalizer. This *Collage* network library is an independent open source project. In the following sections we highlight core features and show how they are different from other distribution mechanisms, e.g., the MPI library.

7.1 Requirements

The Collage network library was conceived with the requirements of a dynamic parallel rendering system in mind. Some of the features implemented by Collage came about later, and are often layered on top of the basic primitives. The core requirements were:

Peer-to-peer network: While the execution model of an Equalizer application follows a master-slave approach, and Equalizer internally uses a client-server model, the core transport layer should be agnostic to these higher-

level abstractions. In Collage, each communicating process is equal to all others, and no traffic prioritization or communication pattern is enforced by a node type. This has proven particularly useful during the implementation of parallel compositing algorithms, where the compositing nodes form an ad-hoc peer-to-peer network.

Dynamic connection management: As a consequence of the peer-to-peer network, all nodes in a cluster are equivalent. Due to the heterogenous nature of a parallel rendering application we felt the need to furthermore have no constraints on the management of connections between nodes. Nodes are identified and addressed by an universally unique identifier. The network layer lazily establishes a connection to any given node, by querying its known neighbors or a zeroconf network for connection parameters. Connections may be established concurrently by both sides of a node pair (e.g. during parallel compositing), which required the implementation of a robust handshake protocol during connection establishment. For larger cluster installations, a fully connected peer-to-peer network would be suboptimal, for example on Windows operating systems there is a latency penalty once more than 64 connections are needed due to some low-level implementation details.

Transport layer abstraction: The actual network protocol is abstracted internally by an API requiring byte-oriented stream semantics. While this choice of abstraction makes it harder for RDMA-based protocols to deliver full performance, it has proven useful in supporting a large set of transports, from standard Ethernet sockets, SDP for InfiniBand, native Verbs for Infiniband, UDT to even a fully-featured multicast-over-UDP implementation. In particular the ease of integration for multicast transport is strong evidence for the usefulness of this abstraction.

Convenient to use for existing applications: The history and code structure of visualization applications is often very different from distributed applications such as simulation codes. They have been developed for years for desktop systems, often are single-threaded and have data models and object hierarchies built for their domain-specific problems and algorithms. The network library needs to provide primitives which match this reality as closely as possible by providing a modern, object-oriented C++ API for data distribution.

7.2 Architecture

Our Collage network library provides a peer-to-peer communication infrastructure, offering different abstraction layers which gradually provide higher level

functionality to the programmer. Collage is used by Equalizer to communicate between the application node, the server and the render clients. Figure 7.1 provides an overview of the major Collage classes and their relationship. The main classes, in ascending abstraction level, are:

Connection: A stream-oriented point-to-point communication line. Different implementations of a connection exists. The connections transmit a raw byte stream reliably between two endpoints for unicast connections, and between a set of endpoints for multicast connections.

DataOStream: Abstracts the output of C++ data types onto a set of connections by implementing output stream operators. Uses buffering to aggregate data for network transmission.

OCommand: Extends DataOStream to implement the protocol between Collage nodes by adding node and command type routing information to the stream.

DataIStream: Decodes a buffer of received data into C++ objects and PODs by implementing input stream operators. Performs endian swapping if the endianness differs between the originating and local node.

ICommand: The other side of OCommand, extending DataIStream.

Node and LocalNode: The abstraction of a process in the cluster. Nodes communicate with each other using connections. A LocalNode listens on various connections and processes requests for a given process. Received data is wrapped in ICommands and dispatched to command handler methods. A Node is a proxy for communicating with a remote LocalNode.

Object: Provides object-oriented, versioned data distribution of C++ objects between nodes within a session. Objects are registered or mapped on a LocalNode.

7.2.1 Connection

A connection is the basic primitive used for communication between processes in Collage. It provides a stream-oriented communication between two endpoints. A connection is either closed, connected or listening. A closed connection cannot be used for communications. A connected connection can be used to read or write data to the communication peer. A listening connection can accept connection requests.

A ConnectionSet is used to manage multiple connections. The typical use case is to have one or more listening connections for the local process, and a number of connected connections for communicating with other processes. The connection set is used to select one connection which requires some action. This can be a connection request on a listening connection, pending data on a connected connection or the notification of a disconnect.

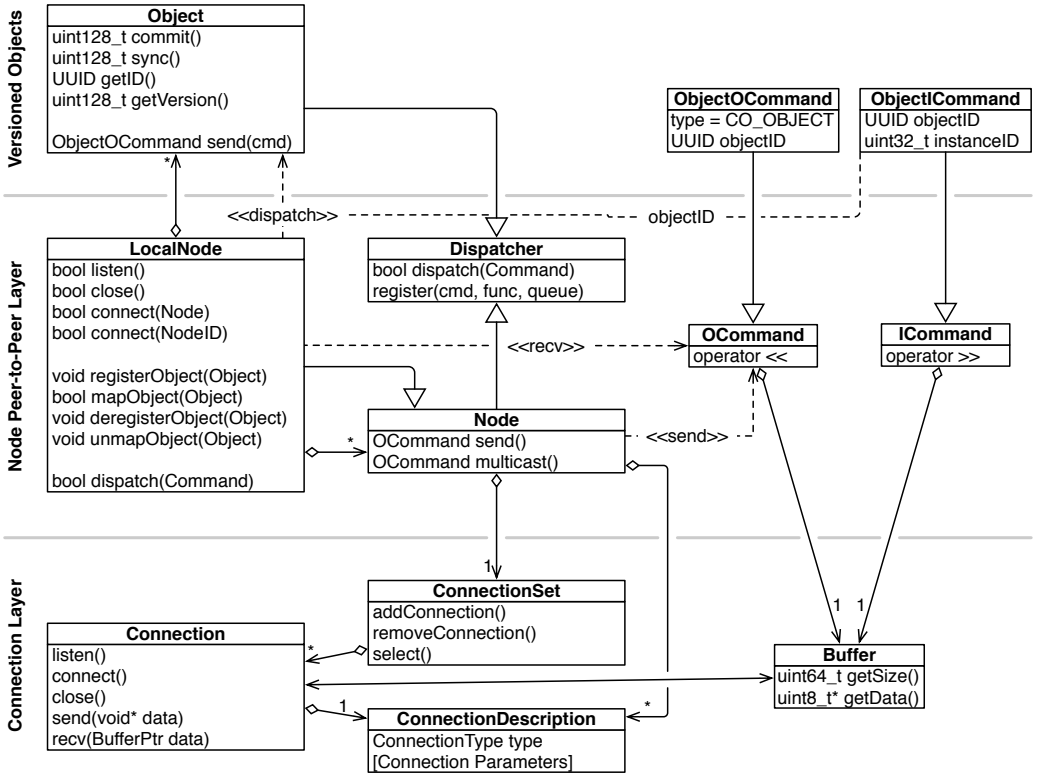


Figure 7.1: *UML class diagram of the major Collage classes*

The connection and connection set can be used by applications to implement other network-related functionality, e.g., to communicate with a sound server on a different machine. A LocalNode has a connection set and uses this to manage connections with other nodes.

7.2.2 Command Handling

Nodes and objects communicate using commands derived from data streams. The basic command dispatch is implemented in the Dispatcher class, from which Node and Object are sub-classed.

The dispatcher allows the registration of a commands with a dispatch queue and an invocation method. Each command has a type and command identifier, which is used to identify the receiver, registered queue and method. The dispatch pushes the packet to the registered queue. When the commands are dequeued by the processing thread the registered command method is invoked.

This dispatch and invocation functionality is used within Equalizer to dispatch commands from the receiver thread to the appropriate node or pipe thread, and then to invoke the command when it is processed by these threads. This dispatching provides object-oriented semantics, since C++ instances can register themselves on the dispatcher, and get automatically invoked in the correct thread when an appropriate command arrives.

7.2.3 Nodes

The Node is the abstraction of one process in the peer-to-peer network. Each node has a universally unique identifier. This identifier is used to address nodes, e.g., to query connection information to connect to the node. Nodes use connections to communicate with each other by sending OCommands.

The LocalNode is the specialization of the node for the given process. It encapsulates the communication logic for connecting remote nodes, as well as object registration and mapping. Local nodes are set up in the listening state during initialization.

A remote Node can either be connected explicitly by the application or due to a connection from a remote node. The explicit connection can be done by programmatically creating a node, adding the necessary ConnectionDescriptions and connecting it to the local node. It may also be done by connecting the remote node to the local node by using its NodeID. This will cause Collage to query connection information for this node from the already-connected nodes and zeroconf, instantiating the node and connecting it. Both operations may fail.

Zeroconf Discovery

Each LocalNode provides a Zeroconf communicator, which allows node and resource discovery. The service "_collage._tcp" is used to announce the presence of a listening LocalNode using the ZeroConf protocol to the network. The node identifier and all listening connection descriptions are announced, which is used to connect unknown nodes by using the node identifier alone.

Communication between Nodes

Figure 7.2 shows the communication between two nodes. Each LocalNode has a receiver thread, which uses a connection set to read and dispatch incoming data from the network, and a command thread used for higher-level functions such as object mapping. When the remote node sends a command, the listening node receives the command and dispatches it from the receiver thread. The dispatch will either invoke the bound function immediately, or enqueue the command into

the given queue. The queue consumer, for example the main or command thread, will read the command of this queue and then invoke the bound function.

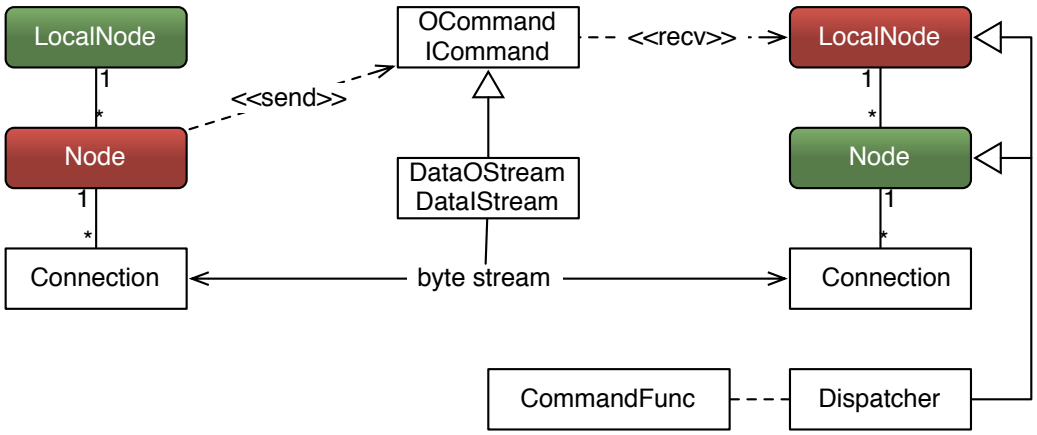


Figure 7.2: *Communication between two Nodes*

7.3 Distributed, Versioned Objects

Adapting an existing application for parallel rendering requires the synchronization of application data across the processes in the parallel rendering setup. Existing parallel rendering frameworks address this often poorly, at best they rely on MPI to distribute data. Real-world, interactive visualization applications are typically written in C++ and have complex data models and class hierarchies to represent their application state. As outlined in [Eilemann et al., 2009], the parallel rendering code in an Equalizer application only needs access to the data needed for rendering, as all application logic is centralized in the application main thread. We have encountered two main approaches to address this distribution: Using a shared filesystem for static data, or using data distribution for static and dynamic data. Distributed objects are not required to build Equalizer applications. While most developers choose to use this abstraction for convenience, we have seen applications using other means for data distribution, e.g., MPI.

Distributed objects in Collage provide powerful, object-oriented data distribution for C++ objects. They facilitate the implementation of data distribution in a cluster environment. Distributed objects are created by subclassing from co::Serializable or co::Object. The application programmer implements serialization and deserialization. Distributed objects can be static (immutable) or dynamic. Objects have a universally unique identifier (UUID) as cluster-wide address. A master-slave model is used to establish mapping and data synchronization across processes. Typically, the application main loop registers a master

instance and communicates the UUID to the render clients, which map their instance to the given identifier. The following object types are available:

Static The object is not versioned nor buffered. The instance data is serialized whenever a new slave instance is mapped. No additional data is stored.

Instance The object is versioned and buffered. The instance and delta data are identical; that is, only instance data is serialized. Previous instance data is saved to be able to map old versions.

Delta The object is versioned and buffered. The delta data is typically smaller than the instance data. The delta data is transmitted to slave instances for synchronization. Previous instance and delta data is saved to be able to map and sync old versions.

Unbuffered The object is versioned and unbuffered. No data is stored, and no previous versions can be mapped.

Serialization is facilitated using output or input streams, which abstract the data transmission and are used like a std::stream. The data streams implement efficient buffering and compression, and automatically select the best connection for data transport. Custom data type serializers can be implemented by providing the appropriate serialization functions. No pointers should be directly transmitted through the data streams. For pointers, the corresponding object is typically a distributed object as well, and its UUID and version are transmitted in place of a pointer.

Dynamic objects are versioned, and on commit the delta data from the previous version is sent, if available using multicast, to all mapped slave instances. The data is queued on the remote node, and is applied when the application calls sync to synchronize the object to a new version. The sync method might block if a version has not yet been committed or is still in transmission. All versioned objects have the following characteristics:

- The master instance of the object generates new versions for all slaves. These versions are continuous. It is possible to commit on slave instances, but special care has to be taken to handle possible conflicts.
- Slave instance versions can only be advanced; that is, sync(version) with a version smaller than the current version will fail.
- Newly mapped slave instances are mapped to the oldest available version by default, or to the version specified when calling mapObject.

The Collage Serializable implements one convenient usage pattern for object data distribution. The co::Serializable data distribution is based on the concept of dirty bits, allowing inheritance with data distribution. Dirty bits form a 64-bit mask which marks the parts of the object to be distributed during the next commit. For serialization, the application developer implements serialize or deserialize, which are called with the bit mask specifying which data has to be transmitted

or received. During a commit or sync, the current dirty bits are given, whereas during object mapping all dirty bits are passed to the serialization methods.

Blocking commits allow to limit the number of outstanding, queued versions on the slave nodes. A token-based protocol will block the commit on the master instance if too many unsynchronized versions exist.

Optimizations

The API presented in the previous section provides sufficient abstraction to implement various optimizations for faster mapping and synchronization of data: compression, chunking, caching, preloading and multicast.

The most obvious one is compression. Recently, many new compression algorithms have been developed which exploit modern CPU architectures and deliver compression rates well above one gigabyte per second. Collage uses the Pression library [Eyescale Software GmbH and Blue Brain Project, 2016], which provides a unified interface for a number of compression libraries, such as FastLZ [Jesper et al.,], Snappy [opensource@google.com, 2016] and ZStandard [Facebook, 2016]. It also contains a custom, virtually zero-cost RLE compressor. Pression parallelizes the compression and decompression using data decomposition. This compression is generic and lossless, implemented transparently for the application. Applications can also use data-specific compression.

The data streaming interface implements chunking, which pipelines the serialization code with the network transmission. After a configurable number of bytes has been serialized to the internal buffer, it is transmitted and serialization continues. This is used both for the initial mapping data and for commit data.

Caching retains instance data of objects in a client-side cache, and reuses this data to accelerate mapping of objects. The instance cache is either filled by "snooping" on multicast transmissions or by an explicit preloading when the master objects are registered. The preloading sends instance data of recently registered master objects to all connected nodes, while the corresponding node is idle. These nodes simply enter the received data to their cache. Preloading uses multicast when available.

Due to the master-slave model of data distribution, multicast is used to optimize the transmission time of data. If the contributing nodes share a multicast session, and more than one slave instance is mapped, Collage automatically uses the multicast connection to send the new version information.

In [Eilemann et al., 2018] we provide an extensive analysis of data compression, buffering and multicast in practical scenarios, and show that they can provide substantial speedups for data distribution.

7.3.1 Reliable Stream Protocol

RSP is an implementation of a reliable multicast protocol over unreliable UDP multicast transport. RSP behaves similarly to TCP; in contrast to the underlying UDP transport, it is not message-oriented but implements byte stream semantics. RSP provides full reliability and ordering of the data, and slow receivers will eventually throttle the sender through a sliding window algorithm. This behavior is needed to guarantee delivery of data in all situations. Pragmatic generic multicast (PGM [Gemmell et al., 2003]) provides full ordering, but slow clients will disconnect from the multicast session instead of throttling the send rate. Since we use multicast for distributing application data to all rendering clients we want semantics similar to TCP, that is, waiting for a client to read data is preferable over loosing this client.

RSP combines various established multicast algorithms [Adamson et al., 2004; Gau et al., 2002] in an open source implementation capable of delivering wire speed transmission rates on high-speed LAN interfaces. In the following we will outline the RSP protocol and implementation as well as motivate the design decisions. Any defaults given below are for Linux or OS X, the Windows UDP stack requires different default values which can be found in the implementation.

Our RSP implementation uses a separate protocol thread for each RSP group, which handles all reads and writes on the multicast UDP socket. It implements the protocol handling and communicates with the application threads through thread-safe queues. The queues contain datagrams filled with the application byte stream, prefixed by a header of at most eight bytes. Each connection has a configurable number of buffers (1024 by default) of a configurable MTU (1470 bytes default), which are either free or in transmission. The header contains two bytes for the datagram type (connection handshake, data, acknowledgment, negative acknowledgment, acknowledgment request), and up two six bytes of datagram-specific information (e.g. for acknowledgment: two bytes read node identifier, two bytes write node identifier, two bytes sequence number).

Figure 7.3 shows the data flow through the RSP implementation. Each member of the multicast group opens a listening connection, which will send query datagrams to the multicast socket. For each found member, a receiving connection instance is created and, similar to a TCP socket, passed to the application upon accept. Each connection instance has a fixed number (1024 by default) of fixed-size (1470 by default) buffers, each used directly as an UDP datagram. The listening connection uses these buffers for writing data, and each receiving connection uses its buffers for received data. These buffers are continuously cycled through two sets of queues: a blocking, thread-safe queue used on the application side for reading and writing data, and a non-blocking, lock-free and thread-safe queue on the protocol thread for data management.

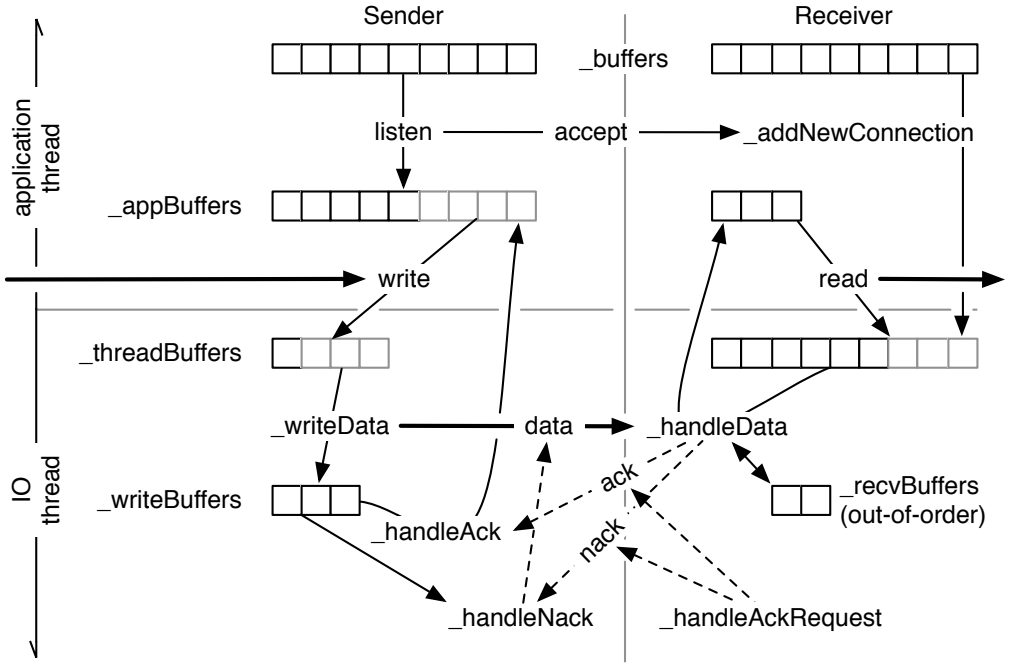


Figure 7.3: *RSP data flow*

When writing data, the application thread pops empty buffers from its queue (blocking when the data cannot be written fast enough), fills in the data datagram headerm and copies the application data piece-wise into the datagram. The datagrams are then pushed onto the protocol thread buffer queue. The protocol thread writes the datagrams unto the UDP multicast socket, and reads and handles any incoming datagrams. On the receiver side, the protocol receives the data, and pushes them in order to the corresponding application thread queue. Out-of-order datagrams are stored aside and queued in order later, and a negative acknowledgments (nack) are immediately sent for missing datagrams. The writer will repeat nack'd datagrams, recycle fully acknowledged datagrams to the application queue, and ask for missing acknowledgments if needed. When reading data, the application pops full buffers from the corresponding connection's queue (blocking when no data is available), copies the data piece-wise out of datagram into the application buffer, and recycles the cleared buffers onto the protocol thread queue.

Handling a smooth packet flow is critical for performance. RSP uses active flow control to advance the byte stream buffered by the implementation. Each incoming connection actively acknowledges every n (17 by default) packets fully received. The incoming connections offset this acknowledgment by their connection identifier to avoid bursts of acks. Any missed datagram is actively nack'ed

as soon as detected. Write connections continuously retransmit packets for nack datagrams, and advance their window upon reception of all acks from the group. The writer will explicitly request an ack or nack when it runs out of empty buffers or finishes its write queue. Nack datagrams may contain multiple ranges of missed datagrams, which is motivated by the observation that UDP implementations often drop multiple contiguous packets.

Congestion control is necessary to optimize bandwidth usage. While TCP uses the well-known additive increase, multiplicative decrease algorithm, we have chosen a more aggressive congestion control algorithm of additive increase and additive decrease. This has proven experimentally to be more optimal: UDP is often rate-limited by switches; that is, packets are discarded regularly and not exceptionally. Only slowly backing of the current send rate helps to stay close to this limit. Furthermore, our RSP traffic is limited to the local subnet, making cooperation between multiple data stream less of an issue. Send rate limiting uses a bucket algorithm, where over time the bucket fills with send credits, from which sends are substracted. If there are no available credits, the sender sleeps until sufficient credits are available.

In [Eilemann et al., 2018] we provide experimental results, showing that our implementation can achieve above 90% wire speed on 10 GBit/s Ethernet, shows good scalability with respect to the multicast group size, and is very effective for distributing structured and unstructured application data to a large number of rendering clients concurrently.

C H A P T E R 8

CONCLUSION

8.1 Future Work

BIBLIOGRAPHY

[MPK,] OpenGL Multipipe SDK.

[Abraham et al., 2004] Abraham, F., Celes, W., Cerqueira, R., and Campos, J. L. (2004). A load-balancing strategy for sort-first distributed rendering. In *Proceedings SIBGRAPI*, pages 292–299.

[Adamson et al., 2004] Adamson, B., Bormann, C., Handley, M., and Macker, J. (2004). Negative-acknowledgment (nack)-oriented reliable multicast (norm) protocol. Technical report.

[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.

[Eilemann et al., 2012] Eilemann, S., Bilgili, A., Abdellah, M., Hernando, J., Makhinya, M., Pajarola, R., and Schürmann, F. (2012). Parallel Rendering on Hybrid Multi-GPU Clusters. In *EGPGV*, pages 109–117.

[Eilemann et al., 2009] Eilemann, S., Makhinya, M., and Pajarola, R. (2009). Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452.

[Eilemann et al., 2018] Eilemann, S., Steiner, D., and Pajarola, R. (2018). Equalizer 2.0 - Convergence of a Parallel Rendering Framework.

[Erol et al., 2011] Erol, F., Eilemann, S., and Pajarola, R. (2011). Cross-segment load balancing in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50.

[Eyescale Software GmbH and Blue Brain Project, 2016] Eyescale Software GmbH and Blue Brain Project (2016). Compression and data transfer plugins. https://github.com/Eyescale/Pression.

[Eyles et al., 1997] Eyles, J., Molnar, S., Poulton, J., Greer, T., Lastra, A., England, N., and Westover, L. (1997). PixelFlow: The realization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware*, pages 57–68.

[Facebook, 2016] Facebook, I. (2016). Fast real-time compression algorithm. https://github.com/facebook/zstd.

[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.

[Gau et al., 2002] Gau, R.-H., Haas, Z. J., and Krishnamachari, B. (2002). On multicast flow control for heterogeneous receivers. *IEEE/ACM Trans. Netw.*, 10(1):86–101.

[Gemmell et al., 2003] Gemmell, J., Montgomery, T., Speakman, T., and Crowcroft, J. (2003). The PGM reliable multicast protocol. *IEEE Network*, 17(1):16–22.

[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.

[Jesper et al.,] Jesper, J., Sadakane, K., and Sung, W.-K. Fast lz-compression algorithm.

[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.

[Makhinya et al., 2010] Makhinya, M., Eilemann, S., and Pajarola, R. (2010). Fast Compositing for Cluster-Parallel Rendering. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV, pages 111–120, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[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.

[Montrym et al., 1997] Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. (1997). InfiniteReality: A real-time graphics system. In *Proceedings ACM SIGGRAPH*, pages 293–302.

[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.

[opensource@google.com, 2016] opensource@google.com (2016). A fast compressor/decompressor. https://github.com/google/snappy.

[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.

[Steiner et al., 2016] Steiner, D., Paredes, E. G., Eilemann, S., and Pajarola, R. (2016). Dynamic work packages in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 89–98.

[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