Architectures for Ubiquitous 3D on Heterogeneous Computing Platforms

---

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Georg Tamm

Saarbrücken, May 2018

Date of Colloquium	May 9, 2018
Dean of the Faculty	Prof. Dr. Sebastian Hack
Supervisor	Prof. Dr. Philipp Slusallek
Examination Board	Prof. Dr. Antonio Krüger (chairman) Prof. Dr. Philipp Slusallek Prof. Dr. Jens Krüger Prof. Dr. Thorsten Herfet Dr. Richard Membarth (academic assessor)

Abstract

Today, a wide scope for 3D graphics applications exists, including domains such as scientific visualization, 3D-enabled web pages, and entertainment. At the same time, the devices and platforms that run and display the applications are more heterogeneous than ever. Display environments range from mobile devices to desktop systems and ultimately to distributed displays that facilitate collaborative interaction. While the capability of the client devices may vary considerably, the visualization experiences running on them should be consistent. The field of application should dictate how and on what devices users access the application, not the technical requirements to realize the 3D output.

The goal of this thesis is to examine the diverse challenges involved in providing consistent and scalable visualization experiences to heterogeneous computing platforms and display setups. While we could not address the myriad of possible use cases, we developed a comprehensive set of rendering architectures in the major domains of scientific and medical visualization, web-based 3D applications, and movie virtual production. To provide the required service quality, performance, and scalability for different client devices and displays, our architectures focus on the efficient utilization and combination of the available client, server, and network resources. We present innovative solutions that incorporate methods for hybrid and distributed rendering as well as means to manage data sets and stream rendering results. We establish the browser as a promising platform for accessible and portable visualization services. We collaborated with experts from the medical field and the movie industry to evaluate the usability of our technology in real-world scenarios.

The presented architectures achieve a wide coverage of display and rendering setups and at the same time share major components and concepts. Thus, they build a strong foundation for a unified system that supports a variety of use cases.

Zusammenfassung

Heutzutage existiert ein großer Anwendungsbereich für 3D-Grafikapplikationen wie wissenschaftliche Visualisierungen, 3D-Inhalte in Webseiten, und Unterhaltungssoftware. Gleichzeitig sind die Geräte und Plattformen, welche die Anwendungen ausführen und anzeigen, heterogener als je zuvor. Anzeigegeräte reichen von mobilen Geräten zu Desktop-Systemen bis hin zu verteilten Bildschirmumgebungen, die eine kollaborative Anwendung begünstigen. Während die Leistungsfähigkeit der Geräte stark schwanken kann, sollten die dort laufenden Visualisierungen konsistent sein. Das Anwendungsfeld sollte bestimmen, wie und auf welchem Gerät Benutzer auf die Anwendung zugreifen, nicht die technischen Voraussetzungen zur Erzeugung der 3D-Grafik.

Das Ziel dieser Thesis ist es, die diversen Herausforderungen zu untersuchen, die bei der Bereitstellung von konsistenten und skalierbaren Visualisierungsanwendungen auf heterogenen Plattformen eine Rolle spielen. Während wir nicht die Vielzahl an möglichen Anwendungsfällen abdecken konnten, haben wir eine repräsentative Auswahl an Rendering-Architekturen in den Kernbereichen wissenschaftliche Visualisierung, web-basierte 3D-Anwendungen, und virtuelle Filmproduktion entwickelt. Um die geforderte Qualität, Leistung, und Skalierbarkeit für verschiedene Client-Geräte und -Anzeigen zu gewährleisten, fokussieren sich unsere Architekturen auf die effiziente Nutzung und Kombination der verfügbaren Client-, Server-, und Netzwerkressourcen. Wir präsentieren innovative Lösungen, die hybrides und verteiltes Rendering als auch das Verwalten der Datensätze und Streaming der 3D-Ausgabe umfassen. Wir etablieren den Web-Browser als vielversprechende Plattform für zugängliche und portierbare Visualisierungsdienste. Um die Verwendbarkeit unserer Technologie in realitätsnahen Szenarien zu testen, haben wir mit Experten aus der Medizin und Filmindustrie zusammengearbeitet.

Unsere Architekturen erreichen eine umfassende Abdeckung von Anzeige- und Rendering-Szenarien und teilen sich gleichzeitig wesentliche Komponenten und Konzepte. Sie bilden daher eine starke Grundlage für ein einheitliches System, das eine Vielzahl an Anwendungsfällen unterstützt.

Acknowledgements

I would like to especially thank my first supervisor, Philipp Slusallek, and my second supervisor, Jens Krüger, for their ongoing support and guidance. Their help was crucial to ultimately arrive at this comprehensive thesis document. I would also like to thank the third reviewer Thorsten Herfet.

I further want to thank my colleagues that worked with me on various projects and often gave me useful insight and directions for improvement. These people contributed in one way or another to the success of this thesis. First and foremost, thanks are due to Alexander Schiewe, Thomas Fogal, Kristian Sons, Arsène Pérard-Gayot, and Jonas Trottnow. Thanks are also due to Christopher Butson and his team at the Medical College of Wisconsin, in particular Sanket Jain, for their commitment to deploy and evaluate our visualization software. Further, a special thanks goes to the Dreamspace project partners with whom we collaborated to refine and deploy our distributed rendering architecture.

I would also like to thank my employers, the German Research Center for Artificial Intelligence, the Cluster of Excellence on Multimodal Computing and Interaction, the Saarland University, and the Intel Visual Computing Institute, as well as the Saarbrücken Graduate School of Computer Science for the opportunity to pursue my research and project work in the rich environment for computer science that the campus of the Saarland University provides.

Last but not least, I want to thank my family and friends who supported me on the way.

Contents

Chapter 1

Introduction

1.1 Scope

Today, a huge demand for various kinds of 3D graphics applications exists. The focus in this thesis is on interactive applications that aim to provide immediate feedback to user input or scene and parameter changes, optimally in real-time at 25 frames per second (FPS) or higher.

With the rapid evolution of computer hardware, the size of data sets generated from acquisition techniques, measurements, and simulations grows continuously [YWG+10]. Visualization is essential and in some cases mandatory to analyze and interpret the vast amount of information contained in the data sets. There is a wide variety of applications. In the medical field [PB13], the visualization of patient data, for example obtained from a CT-scan, can aid in diagnosis and decision making [BTJ+13]. Other applications are the analysis of molecules in biology or chemistry [MHLK06, MDG+10], flow and fluid simulation [Krü07, RCSW14], and terrain visualization [DKW09].

Further, the visualization of technical systems, manufacturing plants, architecture, or production processes supports the monitoring, analysis, and planning of either existing or upcoming installations [PV06, Yan06, GGH12, BBB+14]. The automotive industry uses visualization techniques to drive design decisions or to simulate the properties of a car [SE01, BWDS02], for example in a crash scenario.

While scientific and technical visualization applications deal with a specific set of target users, there are more common use cases accessible to a broader audience. These include virtual environments, training applications like flight simulators, 3D maps, serious games, and educational content. For example, imagine a Wikipedia page that provides a 3D model of a city for visitors to explore.

In the entertainment sector, video games saw a boom with the advent of mobile tablet and smartphone devices capable of 3D rendering. Also, there are special effects in the movie and TV industry and even entirely computer-generated movies. Adding and finalizing special effects has traditionally been a process performed after filming using high-quality offline ray-tracing. But there is a trend and demand to already incorporate interactive rendering on the set to increase the flexibility in the creative collaboration of directors, actors, designers, and computer generated imagery (CGI) experts [GHJ+16]. The integration of ray-tracing in games is of further interest [FGD+06, Bik07, KKW+13].

At the same time, the heterogeneity in the devices and platforms that can run and display 3D applications is higher than ever today. In addition to the desktop computers and workstations in use throughout homes, offices, and research institutions, the spread of tablets and smartphones has put devices capable to display and to some extent render 3D graphics in use virtually anywhere and anytime. Further, there are large displays or tiled display walls that provide content to a larger, potentially public audience or aid in the collaborative examination of data sets at extremely high resolutions [TSK11, LPHS12].

While native applications may be maintained for each target platform independently, browsers continuously increase the capabilities they provide, including 3D graphics, media processing, and networking [TS15, MAN+14]. This enables the development and deployment of portable applications in standard web pages that users can access from anywhere.

To generate and provide the data for visualizations and to perform or support expensive rendering tasks, servers and data centers can provide services for less capable clients [TS16, Ama16]. However, a centralized cloud at the core of the network may impose considerable latency to clients that are geographically far away. Fog computing [BMZA12] reduces the dependence on the cloud as it places location-aware resources close to the client devices, thus facilitating latency-sensitive applications such as interactive rendering.

To conclude, there is a huge demand and also opportunity for 3D applications in various areas, which we define as “Ubiquitous 3D”. To tackle the heterogeneous requirements, we must investigate different kinds of rendering architectures, display setups, and methods of interaction. A scientific visualization of a medical data set may require volume rendering [FK10, BTJ+13], while a rasterizer is adequate for a simpler illustration of an architectural model [SDHS13]. To judge the effects of realistic lighting on a movie set, a ray-tracer producing physically correct images may be employed [GHJ+16]. Some applications may target a single user on a desktop workstation, others may allow collaborative scene editing via mobile devices, and still others may present content in public on a large display or display wall [TSK11]. There are augmented reality applications [KRS+13] and applications designed for stereoscopic and

virtual reality (VR) displays [PZB16] where ultra-low latency feedback is mandatory, again requiring a dedicated methodology for rendering and interaction. Rendering may be performed on the display device, remotely on a server or server farm, or even on both sides in a hybrid fashion [TK14].

The vision of this thesis is a compute continuum [DJX14] that utilizes the available client, server, and network resources to provide consistent and scalable visualization experiences across heterogeneous display devices. While we could not cover all the use cases and application areas, we set out to develop and analyze a comprehensive set of rendering architectures and methods in the scope of ubiquitous 3D. We address a wide range of display platforms and collaborated with experts from the medical and movie field to evaluate the applicability of the solutions in real-world scenarios. Contributions involve visualization on large, tiled displays (Chapter 2), medical visualization on mobile devices (Chapter 3), hybrid visualization utilizing client and server machines (Chapter 4), visualization and 3D applications using the browser as the platform (Chapter 5), and a framework for server-based and distributed rendering that supports real-time ray-tracing and load balancing (Chapter 6 and 7).

1.2 Challenges

"Ubiquitous 3D" is a term that involves a substantial number of use cases and challenges across different kinds of rendering architectures. We first give a general description of the challenges and then introduce the thesis contributions to address them. We can separate the challenges into the following areas.

Rendering

In an interactive context, a renderer must not only uphold a minimum image quality as required by the visualization application, it must also provide this quality at interactive or even real-time frame rates. We consider 25 FPS and above as real-time. Given the possible complexity and size of 3D scenes and data sets generated today, this can put a strong performance requirement on the renderer and the hardware it is running on. Not all devices that shall display the visualization may have the capability to process or even fully store a data set. Some devices may not be able to run a specific rendering algorithm due to the limitations of the hard- or software environment. Especially mobile devices may suffer from such limitations. In case of the browser as the execution platform, persistent storage is restricted and rendering algorithms are bound by the limited subset of OpenGL that WebGL [Khr17c] provides. The features of JavaScript (JS) are constrained by the the secured browser environment [YSD+09], while native code has direct access to the operating system and any external library. JS performance

still lacks behind native code in some cases, especially for parallel applications [KFBK+14]. In a future exascale scenario, it may not even be feasible to download simulation data from a supercomputer. Instead, the supercomputer may generate and visualize the data sets in-situ [Ma09, YWG+10]. Lastly, the scenes or data sets may be confidential, which prohibits a download for local rendering on the display devices.

A solution for these restrictions is to move the heavy lifting to a stronger remote machine that is better suited for the task. A server can maintain the data sets, perform rendering, and send generated images to the client for display [TS15]. In this setup, the client acts as the user interface only and requirements on the hard- and software environment are typically minimal. However, a single server might not be able to provide the desired level of service to multiple clients that connect simultaneously. To increase the scalability, considerable investments in hardware and the deployment of multiple servers may be necessary depending on the number of target users and the demands of the visualization. Today, large-scale cloud computing services exist that support rendering among other applications [Mic16, Ama16, NVI16a].

An approach to tackle the scalability issue is to employ hybrid rendering [TK14]. While the server component is necessary to provide large-scale visualization to common devices, a purely server-based setup leaves a considerable amount of hardware idle on the client side. Today, even commodity mobile devices usually have the capabilities to perform rendering tasks to some degree. In hybrid rendering, server and client work together to generate results. This requires to split the rendering process into pieces of work and assigning these pieces appropriately to both sides. Performing some tasks on the client can increase the overall performance of the visualization and frees server-side resources for other clients. The client hardware is available without additional investment from the server provider.

Still, the visualization of potentially large scenes at very high quality, either by employing an expensive rendering algorithm like ray-tracing or by increasing the display resolution, can put a server machine under much pressure even for a single client. To uphold interactive frame rates, the distribution of rendering tasks to a cluster of servers may be required [TS16, GHJ+16]. The crucial process to scale a renderer on such an architecture is load balancing [TS16]. The load balancer aims to split tasks in a way that achieves maximum utilization of all participants during the collaborative rendering of a frame.

Network

While remote and distributed rendering can overcome client limitations, it adds the network as another layer that must be able to keep up with the desired frame rate and rendering resolution. The server must send the images it produces to the client for display. Should the client participate in the rendering, it must receive the scene data the server stores or even

generates. A client running in the browser must perform the download each time it connects, since the browser does not allow to store large binary data persistently [TS16]. Vice versa, the client must forward user input such as moving the camera and other possible scene changes to the server. A collaborative application may require scene synchronization between several parties. When rendering in a cluster, scene data and updates must be distributed to the nodes, the distributed rendering process must be coordinated, and rendering results must be assembled.

Consequently, the chosen network protocol and the connection latency, bandwidth, and reliability can substantially impact the quality and responsiveness of the visualization. These considerations are especially important when there is potentially an unreliable best-effort network like the Internet or a wireless link between client and server. Allowing the client-side generation of images in a hybrid rendering approach is one way to reduce the dependency on the network.

Further, the compression of scene data and images is crucial to reduce the required bandwidth [SSS14, TS15], but there is a tradeoff between compression ratio and en-/decoding overhead and latency. In addition, the use of lossy image encoders is prevalent as a feasible compression ratio can otherwise not be achieved for high resolutions. The result may be a perceivable loss of image quality. Moreover, encryption can play a role when accessing sensitive scene and image data stored or generated on a server via a public network.

In a rendering cluster, the nodes may send their results as raw pixel data to be accumulated at a central master node [TS16]. This requires a dedicated high-bandwidth cluster network, especially when images should be rendered at high resolutions. The master displays the results directly or encodes them for sending to a display device outside the cluster. Further, finding a load balancer that minimizes the inter-node communication to coordinate the rendering is crucial to overcome intra-cluster network latency as a bottleneck.

Display and Interaction

While hybrid and distributed rendering frameworks can provide large-scale visualization of highly detailed scenes and data sets, the user must be able to view and interpret the information appropriately. The resolution of a single desktop display, let alone mobile display, might not be adequate to capture enough details at once for a proper analysis [BN05, JAW+12]. It can even be beneficial to display multiple views simultaneously to discover correlations in the data or to allow several users to focus on different regions. A single user might find it hard to interpret the data alone, which makes collaborative review an important feature.

A solution to limited resolution and space are distributed display systems [TSK11, LPHS12].

The prime example is a display wall that aligns multiple displays to act as one large unit. Such a system provides enough resolution and physical space to present highly detailed 3D content to multiple individuals. The generation of content for such high-resolution displays further increases the demands on the rendering architecture, which promotes a distributed approach with several machines feeding the display. A synchronization layer must make sure that results are presented seamlessly corresponding with the physical alignment of the displays and considering possible input from multiple users. Moreover, the traditional mechanisms to interact with a single-user desktop application do not map well to a display wall setup with several users in front of it.

The advent of mobile devices provides an opportunity to tackle the interaction issue. Employing the smartphones and tablets that users are familiar with to interact with the display wall increases acceptance and accessibility of such a system [TSK11]. In general, allowing users access on mobile devices is an important concept to establish visualization systems non-invasively into new environments, facilitate on-site usage, and lower the entry barrier for non-experts [BTJ+13, SRCW13]. For example, a physician could review patient data at the point of care without having to interrupt the daily routine by visiting a specialized computer laboratory. This requires not only a simplified user interface but also the means to reliably distribute the visualization results or data sets to the target devices in a time-critical, productive environment.

A visualization application may consist of several components including client devices, storage and rendering servers, network infrastructure, and possibly distributed displays. To reach its users, the application may target heterogeneous platforms for deployment and access. The requirement to develop portable code bases and user interfaces can thus be crucial to minimize the maintenance effort and to maximize availability.

An approach for the portable development of the client-side display application is to use the web browser as the platform [TS15, TS16]. Modern browsers increasingly expand the functionality they provide and are thus suitable for a range of use cases including 3D graphics. Developing an application within web standards and functionality widely supported across browsers enables a unified client interface running homogeneously across various devices. Users can access the application simply by visiting a web page on the device of their choice without requiring a platform-specific installation, which substantially improves the ease of access. However, while browser APIs close in on the features of native libraries, they still restrict or provide a less comprehensive access to the crucial resources storage, CPU, GPU, and networking. This poses an additional challenge when implementing local and remote rendering support in the browser.

Summary

Table 1.1 summarizes the challenges addressed in the thesis and links to the corresponding

chapters. The table lists a chapter in normal text if the chapter's core contribution targets the corresponding problem description and in brackets if we consider the problem a secondary topic. Some chapters also touch on areas they are not listed for or reuse parts developed earlier, which we do not explicitly state here. While the major focus is on server-backed and scalable rendering architectures, we also address the display and usability side. The comprehensive coverage is in line with the vision of a compute continuum for ubiquitous 3D.

The next section outlines the core contributions of each chapter with reference to the challenges.

Table 1.1: High-level overview of the challenges addressed in the thesis.

Challenge	Chapters	Identifier
Rendering		
Limited client capability	4, 5, 6, 7	R.1
Limited server capability		
To handle multiple clients	4, (6)	R.2.1
To uphold required quality and frame rate	6, 7	R.2.2
Network		
Latency and unreliable link		
Within rendering cluster	6	N.1.1
To display client	4	N.1.2
Limited Bandwidth		
Within rendering cluster	7	N.2.1
To display client	4, 5	N.2.2
Display and Interaction		
Limited space and resolution	2	DI.1
Limited usability and thus acceptance	2, 3, (5), (6), 7	DI.2
Limited portability and thus availability	5, 6	DI.3

1.3 Contributions

The thesis establishes a comprehensive set of architectures and methods in the scope of ubiquitous 3D. We deal with most of the challenges described in the previous section as Table 1.1 showcases. Here, we give a summary of the contributions and reference the corresponding publications and projects. We also derive the contributions from the corresponding problem

statements by referencing the identifiers in Table 1.1.

Chapter 2: The ZAPP Distributed Display System

The first chapter deals with distributed displays. We describe the Zero Administration Powerwall Package (ZAPP), which is a framework to connect several displays and run visualization applications on the resulting larger display (DI.1). The term "powerwall" encompasses both the rendering machines required to power the high-resolution visualization and the tiled display wall to present the results.

While previous frameworks focus on the distributed rendering techniques needed to generate the visualization, the main purpose of ZAPP is to improve the user, administrator, and developer experience in dealing with distributed displays (DI.2). Even a single user with no administrative knowledge can operate a ZAPP-managed display system. The goal is that practically anyone is capable of operating the display with less than two minutes of training. To achieve this, we present a lightweight management framework that links and controls the render workstations that drive the displays but also connects to the users' very own mobile devices, such as smartphones or tablets, to enable convenient control over the display. Consequently, the users never need to leave their familiar hardware and operating system environment when accessing the display wall.

Publication: [TSK11]

Chapter 3: Mobile Visualization for the Selection of Deep Brain Stimulation Parameters

In this chapter, the focus is on the mobile sector. In particular, we present the scientific visualization software ImageVis 3D Mobile (IV3Dm) and its deployment in a real-world environment to aid in the selection of parameters for the Deep Brain Stimulation (DBS) treatment of Parkinson's disease patients. We provided the technical backbone while the Medical College of Wisconsin conducted the evaluation of the system.

In recent years there has been significant growth in the use of patient-specific models to predict the effects of neuromodulation therapies such as DBS. However, translating these models from a research environment to the everyday clinical workflow has been a challenge, primarily due to the complexity of the models and the expertise required in specialized visualization software. We deploy IV3Dm, which has been designed for mobile computing devices such as the iPhone or iPad, in an evaluation environment to visualize models of Parkinson's disease patients who received DBS therapy. To provide new patient models to the clinicians with minimal disturbance of their daily routines, we developed a flexible data distribution architecture based on instant messaging.

Selection of DBS settings is a significant clinical challenge that requires repeated revisions to achieve optimal therapeutic response and is often performed without any visual representation of the stimulation system in the patient. We provided IV3Dm to movement disorders clinicians and asked them to use the software to determine: 1) which of the four DBS electrode contacts they would select for therapy; and 2) what stimulation settings they would choose. We compared the stimulation protocol chosen from the software versus the stimulation protocol that was chosen via clinical practice (independently of the study). Lastly, we compared the amount of time required to reach these settings using the software versus the time required through standard practice. We found that the stimulation settings chosen using IV3Dm were similar to those used in standard of care but were selected in drastically less time. We show how our visualization system, available directly at the point of care on a device familiar to the clinician, can be used to guide clinical decision making for selection of DBS settings (DI.2).

Publication: [BTJ+13]

Chapter 4: Hybrid Rendering of Multi-Resolution Data Sets in Dynamic Environments

In this chapter, we propose a hybrid rendering method that utilizes both server and client resources in the interactive visualization of potentially very large data sets. It is a common approach to represent such data sets in a hierarchical format to allow the rendering of different levels-of-detail. While rendering at the highest resolution may take arbitrary time, the lowest levels are intended for interactive performance. The renderer refines the view progressively with levels of increasing detail.

When processing every level of a multi-resolution data set on the server, requirements on the client side are minimal as the client only displays the results it receives. However, the client may have a considerable amount of hardware available that is left idle. Further, the visualization is put at the whim of possibly unreliable server and network conditions. Server load, bandwidth, and latency may substantially affect the response time on the client.

Our method assigns visualization workload in terms of levels-of-detail to both server and client and supports any renderer that can map its data accordingly. A capable client can produce images independently (N.1.2, N.2.2). The goal is to determine a workload schedule that enables a synergy between the two sides to provide rendering results to the user as fast as possible (R.1, R.2.1). The algorithm generates the schedule based on processing and transfer timings obtained at run-time. The probabilistic scheduler adapts to changing conditions by shifting levels between server and client and accounts for the performance variability in the dynamic environment.

Publications: [TFK12, TK14]

Chapter 5: The Browser as the Platform for Remote Visualization

The previous three chapters described architectures that employ a native client-side application. In this chapter, we describe how portable and accessible remote visualization applications can be developed in the browser (DI.3).

Today, users access information and rich media from anywhere using the web browser on their desktop computers, tablets, and smartphones. But the web evolves beyond media delivery. Interactive graphics applications like visualization or gaming become feasible as browsers advance the functionality they provide. However, to deliver large-scale visualization to thin clients like mobile devices, a dedicated server component is necessary. Ideally, the client runs directly within the browser the user is accustomed to, requiring no installation of a plugin or native application. We present the state-of-the-art of technologies that enable plugin free remote rendering in the browser (R.1, N.2.2). Further, we describe a remote visualization system that unifies the technologies. The server transfers rendering results to the client as images or as a video stream. We utilize the World Wide Web Consortium (W3C) conform Web Real-Time Communication (WebRTC) standard and the Native Client (NaCl) technology to deliver video with low latency.

Publication: [TS15]

Chapter 6: Distributed Real-time Ray-Tracing for Declarative 3D in the Browser

The previous chapter presented browsers as a viable platform for portable application development within a web page. To further facilitate the development of browser-based 3D applications, frameworks that allow a declarative scene description in line with the HTML markup exist. The goal is to decrease the entry barrier for the common web developer who is accustomed to HTML but does not necessarily have domain-specific graphics API knowledge. However, the existing approaches utilize client-side rendering and are thus limited in scene complexity and rendering algorithms they can provide on a given device.

This chapter presents an extension of the declarative 3D framework XML3D to support server-based rendering (R.1, DI.3). The server back-end enables distributed rendering with an arbitrary hierarchy of cluster nodes in an InfiniBand or standard network (R.2.2). In the back-end, we deploy a custom real-time ray-tracer that supports additional material properties and effects compared to XML3D's client-side rasterizer. To distribute the ray-tracer, we present a load balancing method that exploits frame-to-frame coherence in the real-time rendering context. The load balancer achieves strong scalability without inducing communication overhead during rendering to coordinate the nodes (N.1.1).

Publication: [TS16]

Chapter 7: The Dreamspace Distributed Rendering Architecture for Virtual Production

The traditional pipeline for TV and movie virtual productions, which combine the real world and CGI, is to add the visual effects during post-production after the on-set filming. However, this separation limits the creativity and flexibility in the collaboration of directors, actors, and CGI experts.

The Dreamspace project developed a platform to combine the virtual and the real world on-set in real-time and give interactive control over the CG components. The ability to experiment with virtual assets while filming can substantially enhance the creative process and cut post-production costs. The Dreamspace pipeline involves multiple parts including real world lighting and depth capture, high-quality rendering of the virtual scene, compositing of virtual and filmed content, and means to collaboratively edit the parameters of the on-set visualization.

This chapter details a major component in the pipeline: the distributed rendering framework that provides high-quality and interactive previews of the scene in the on-set environment (R.1, R.2.2). The design focus is on a maximum level of performance but also on a high level of usability and flexibility to be readily employed even on commodity hardware (N.2.1) and by the possibly non-technical staff prevalent on the set (D.2). The main renderer of the framework is a global illumination ray-tracer that can generate physically correct lighting, thus allowing the professionals to judge the outcome of visual effects realistically.

Project report: [GHJ+16]

1.4 Thesis Structure

The thesis proceeds with the six main chapters as outlined in the previous section. The last chapter is the conclusion and discussion of future work.

A good amount of the developed rendering infrastructure and technology resurfaces in variants across the thesis, which we indicate via cross-references. This includes a library for local and remote rendering used in the visualization applications from Chapter 2 to 5. It also includes the distributed rendering framework that we describe in Chapter 6 and build on in Chapter 7. It further includes the image transport methods that we describe in Chapter 5 and re-utilize in the distributed rendering framework and its web client.

However, rather than describing one overall system, each chapter contains an experiment in the scope of ubiquitous 3D that can stand on its own. Therefore, we present related and future work along with each chapter, and the chapters can be read mostly independent from each other.

Chapter 2

The ZAPP Distributed Display System

2.1 Introduction

Driven by the rapid evolution of computer simulations, acquisition technologies, and computing hardware, data sets are growing at a rapid pace, and even with advanced analysis and visualization techniques it becomes ever harder for a single user or specialist to interpret the data alone. Moreover, the display resolution has increased at a substantially lower pace than computing power, storage, and network bandwidth [WAB+05]. The limited resolution of a single display can complicate the analysis as the display can only show a small area of the data set at high detail or a large area at low detail at any one time. Important structures and correlations may remain hidden.

A method to allow for the real-time collaboration of multiple users is a distributed display system. Figure 2.1 illustrates a tiled display wall that provides significantly more area and resolution than a simple computer screen or projection and thus promotes the collaboration of multiple individuals to examine complex data sets. Ball and North [BN05] demonstrate the benefit a large display can have to navigate through finely detailed visualizations. The ability to connect further display resources from remote locations, such as other large displays, simple workstations, or even mobile devices, provides a flexible platform for visual analysis.

While the software systems required for such a scenario are very complicated, the user interfaces should not be. In this work, we do not focus on the user interface of a specific visualization application running on such a display but on user interfaces to manage the display and its applications in general. To our best knowledge, this part of the user experience has been neglected. While a number of very mature frameworks exist for communication, rendering, and synchronization in distributed display environments, administrators and users in addition

Figure 2.1: A distributed weather analysis and emergency warning system, an exemplary visualization scenario running on top of the ZAPP framework.

require a control layer for installation, maintenance, and configuration and to select, start, and terminate applications. We developed this framework with the concept in mind that any user can launch and interact with an application on the display with minimal training and with no help by experienced staff controlling the display in the background, which we call "director of the institute proof". In addition to this stable platform for everyday use of the display, we also want the system to support rapid development of new applications without compromising the stability of existing software. With these goals in mind we developed the ZAPP framework.

The remainder of the chapter is structured as follows: In the next section we take a look at previous work for distributed displays. In Section 2.3 we give an overview of ZAPP. In the following section we outline ZAPP from the usage perspective and thereby focus on three main areas: administration, development, and application usage. Section 2.5 then details the implementation and architecture of ZAPP, focusing on the different components involved and the communication flow between them. We close the chapter with a summary of the results and give directions for future work.

2.2 Related Work

There is a substantial amount of research in the area of distributed rendering, where partial results from each render workstation are either reassembled on a single display or routed to multiple displays. Humphreys et al. [HBEH00, HEB+01] propose WireGL, a framework for distributed OpenGL rendering. WireGL distributes OpenGL commands and corresponding geometry across a cluster of rendering workstations. On top of WireGL, Chromium [HHN+02a] adds a more general approach to arrange the workstations by utilizing a modular streaming model. Employing this model also removes the bottleneck of constantly transferring geometry

required by the rendering servers over the network, which made efficient fine-grained load balancing difficult with WireGL. ClusterGL [NHM11] improves on Chromium by only transmitting compressed command buffer differences between consecutive frames utilizing UDP multicast. They show the bandwidth savings can outweigh the additional CPU overhead.

Independently of the WireGL branch of systems, Doerr et al. [DK11] developed the Cross Platform Cluster Graphics Library (CGLX), which specifically targets distributed, high-performance visualization via a transparent OpenGL interface. Garuda [NHN07] has a similar approach to Chromium and ClusterGL but targets Open Scene Graph (OSG) applications. The framework employs culling and caching for each display tile to achieve an efficient distribution of the per-frame scene graph changes.

Equalizer [EMP09] is another parallel rendering framework based on OpenGL, with a focus on scalability, flexible configuration, and a developer friendly programming model. Equalizer supports an arbitrary amount of workstations and displays and provides several improvements over Chromium such as decentralized geometry access and built-in thread management. Parts of the framework build on the OpenGL Multipipe SDK [BRE05].

Allowing for both rasterization based approaches as well as ray-tracing, DRONE [RLRS09] is a flexible framework for interactive visual applications rendering and displaying on multiple workstations.

SAGE [RJJ+06, JRJ+06, NDV+10] implements a more general approach than the above frameworks, which focus mostly on the rendering side. SAGE targets collaborative visualization applications that explore large-scale scientific data sets and may utilize a variable number of workstations and displays. A windowing system allows to execute several applications on the display wall in parallel. The image streaming employed in SAGE builds on the previous work TeraVision [SJR+04], which is a solution for high-resolution image streaming between an arbitrary number of workstations. SAGE requires a high-bandwidth network to operate, limiting it to local collaboration. However, it has been extended to support collaboration between multiple distant endpoints with Visualcasting [Jeo09]. The successor to SAGE is SAGE2 [MAN+14, RMA+16], which utilizes the browser as the platform to greatly enhance the portability and accessibility of the system. Unlike SAGE, SAGE2 supports simultaneous multi-user interaction with shared content.

DisplayCluster [JAW+12] is another general solution that allows the streaming of high-resolution content to tiled display walls. The system can display images and videos. It also supports real-time streaming of desktop screens and of image buffers from custom rendering applications. Similar to SAGE, DisplayCluster implements a windowing system to run multiple applications freely on the wall. The framework supports user interaction via mobile devices as

well as scripted events. Nachbaur et al. [NDB+14] present a software stack for parallel rendering applications on large displays that incorporates DisplayCluster and Equalizer.

Both SAGE and DisplayCluster use image-based methods to transport pixels over the network. Bundulis and Arnicans [BA15] show the advantage hardware-accelerated video streaming can have. Display as a Service (DaaS) [LPHS12] is a framework that utilizes video streaming. Using the concept of virtualization, applications can write into software resources called virtual frame buffers (VFB) that output to virtual displays (VD). Both resources are network-attached. VFBs provide video streaming capability, while VDs receive the stream and can map to potentially multiple physical displays. A similar concept has been presented earlier in Hagen's work [Hag11]. DaaS can scale and place any number of VFBs anywhere on the display, only relying on standard IP networking. DaaS employs a tight synchronization layer based on the work of Miroll et al. [MLM+12] and supports stereoscopic content.

Scheidegger et al. [SVK+12] present a framework to integrate existing scientific visualization applications with display walls. As an example, they implemented support for VTK. VTK is a visualization library that packages a suite of visualization tools under a common interface. Moreland and Thompson [MT03] extend VTK to support cluster-based parallel rendering and delivering results to a single display or a display wall. The solution to render to a display wall builds on Chromium and IceT. Implementations for both APIs are included. IceT is a parallel rendering framework that targets display walls as output and builds on algorithms outlined by Moreland et al. [MWP01]. DisplayCluster uses IceT to output to a tiled display. Fogal et al. [FCS+10] use IceT to connect distributed memory multi-GPU clusters for large data set visualization.

Omegalib [FNM+14] is a framework that combines high-resolution display walls with immersive experiences to create so called hybrid reality environments. Omegalib uses Equalizer internally and supports OpenGL, OSG, and VTK applications.

Related to this work is the TileViewer visualization framework [Kim06], which supports multiple application types including image viewing and video display. TileViewer includes a graphical user interface (GUI) to manage the displays and deploy files to the workstations.

In contrast to the above generic frameworks, which are designed to run various types of visual applications, domain-specific solutions exist that have been tailored to a particular area. Application areas include rendering and exploration of large geometric or volumetric data sets, high-resolution image display, video display and information visualization.

Vol-a-Tile [SVR+04] is a distributed volume rendering application able to display high-resolution data sets on a tiled display wall. Correa et al. [CKS02] present an extension to

the iWalk out-of-core rendering system to enable distributed rendering of large static geometric data sets. Nam et al. [NJR+09] describe the integration of ParaView into the SAGE framework to allow the visualization of high-resolution rendering results on a tiled display wall. JuxtaView [KVV+04] is a distributed, parallel image viewer for ultra-high resolution image data. JuxtaView distributes data across a cluster of rendering workstations and employs a caching and pre-fetching strategy to reduce the impact of network latency. Consequently, the user can view and interact with potentially arbitrary sized images on tiled display walls. Like JuxtaView, Magic View [LZWH13] is an image viewer ported to SAGE. Magic View provides considerably better performance than JuxtaView. With Giga-stack, Ponto et al. [PDK10] propose a technique to explore multi-dimensional, giga-pixel images in a high-resolution display environment.

While the previous generic and domain-specific solutions describe particular distributed rendering and pixel distribution techniques and thereby in some cases have proven to be an especially valuable contribution, none have a major focus on how to setup, manage, and operate a large distributed display system from the usability perspective in a real-world, productive, and possibly public environment. In this situation it is highly desirable for the system to be installed and accessed by non-expert users, for example for presentation purposes. Thus, we present ZAPP, a management framework to deploy, maintain, and run distributed visualization applications in a flexible, stable, and user-friendly way. Like ZAPP, SAGE2 and DisplayCluster emphasis the usage of personal and mobile devices to interact with the display wall. However, these frameworks have been introduced at a later time.

2.3 Architecture Overview

ZAPP is a lightweight management framework to run and collaboratively interact with distributed visual applications that can output to multiple displays. We designed the framework specifically to make development, deployment, and maintenance of applications, as well as setting up and configuring the framework, and finally running applications easy, flexible, and stable.

Before starting with ZAPP, an initial hardware setup has to be available to deploy the framework on. The most common architecture is a tiled display wall as Figure 2.2 outlines. The setup consists of a number of rendering workstations or nodes, each running the distributed application and powering a single or multiple displays. A rendering node may generate the visualization for its displays locally or connect to further server machines to offload work. The right side of Figure 2.1 shows the display wall we used to implement and test ZAPP. The setup

consists of five rendering nodes, each connected to four 2560x1600 resolution displays, yielding an overall resolution of around 82 megapixels.

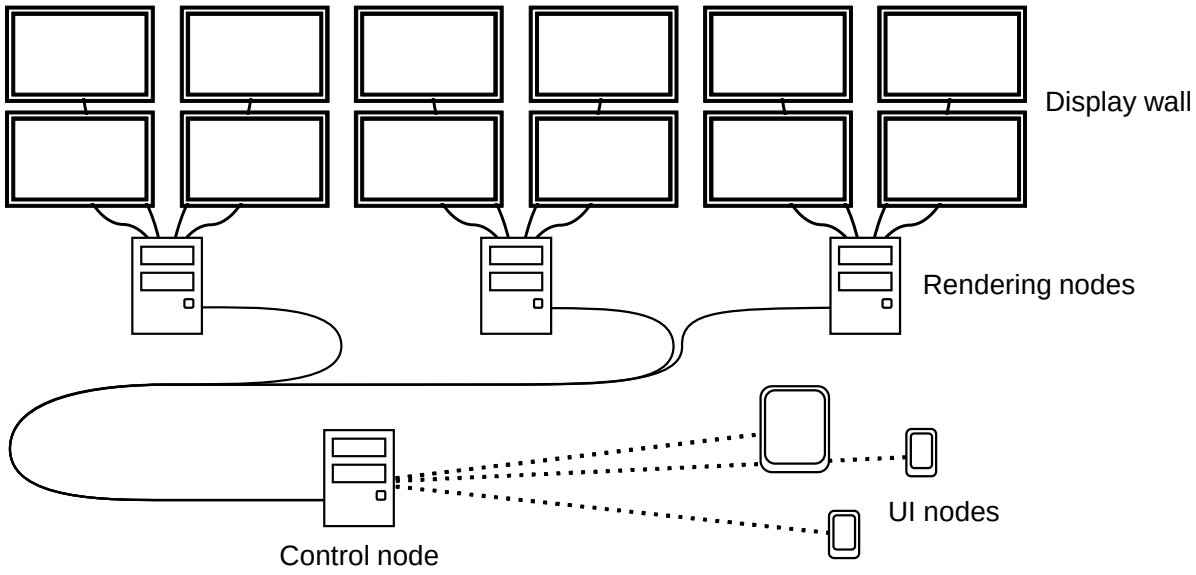


Figure 2.2: Conceptual setup of a distributed display environment. Nodes do not necessarily correspond to separate systems. For example, the control node can also be a rendering node.

To coordinate the distributed application, one machine must run control software; we call this machine the control node. The control node is responsible to launch applications and to configure the overall display alignment. It communicates with and keeps track of the available rendering nodes and their displays. Therefore, the rendering nodes register with the control node and periodically notify it about their current state. We recommend to connect rendering and control nodes via a reliable high-bandwidth network.

Users can issue commands to the control node via one or multiple machines, which we call user interface (UI) nodes. The UI nodes provide an interface to launch and terminate applications and, optionally, an application-specific interface to interact with a running application. UI nodes are usually mobile devices connected wirelessly to the control node. The control node can be a rendering and UI node at the same time, thus a minimal ZAPP configuration already runs on a single machine. We detail the different nodes and the software components running on them in the following sections.

2.4 Usage

This section describes the usage of ZAPP from the perspective of an administrator, a developer, and a user who runs and interacts with applications.

2.4.1 Administration

Given the hardware setup depicted in Figure 2.2, the installation of ZAPP requires only a few steps. ZAPP provides an installer, which should be copied to a USB-device or a network share. First, the installer prepares the control node. A GUI guides through the initial setup, which involves setting up the ports the control node uses, defining an optional password that is required to start and interact with applications, and restricting permission to a range of IP addresses. It is possible to skip all these steps. In this case, the installer chooses default settings automatically.

After installation, the control node process starts and waits for rendering nodes to connect. The next step is the setup of the rendering nodes, which is done using the same installer that is used for the control node. The procedure is automatic if the installer was run from a writable medium on the control node. In this case, the installer remembers the control node's configuration to setup the rendering nodes. If automatic setup is not possible, another GUI lets the administrator configure these settings manually. Also, the administrator can choose a name for each rendering node. The name simplifies identification of the rendering nodes at the control node. When the ZAPP process starts on a rendering node, it automatically detects the number of displays attached to it and notifies the control node.

The final step is the configuration of the overall display alignment, also referred to as the display grid. For this purpose, we provide a GUI-based management software that runs on the control node. The administrator can map each physical display to a two-dimensional coordinate that defines its position within the virtual grid. A coordinate identifies a tile in the grid. A tile is empty if there is no display linked to it. This allows to account for holes in the physical display wall. In addition, the administrator can set up the physical size of each display and its bezel individually or collectively for all tiles. An application can later choose whether to ignore the bezel or consider it a hidden part of the available display space. Consequently, our framework enables to combine different kinds of tiles flexibly to setup a distributed display environment. Of course, the recommended setup is a grid representing a homogeneous tiled display wall.

After the alignment procedure, the ZAPP installation is complete. Maintenance is possible at run-time. The rendering nodes periodically confirm their presence and display setup with the control node. Should a rendering node or displays attached to it become unavailable, ZAPP automatically detects this and disables the corresponding tiles in the display grid so application behavior can remain consistent. The administrator can add rendering nodes at run-time by repeating the rendering node installation process. At any time, the administrator can edit the display grid to adapt to missing or additional displays.

Installing ZAPP enabled visualization applications is straightforward. The management software on the control node allows to select packaged application binaries and settings for deployment on the rendering nodes. An application can be flagged as experimental indicating it should not run in a production environment yet. The user can launch the application directly from the control node or indirectly through a UI node. The software to launch applications from a UI node has to be installed separately. Since a UI node never directly communicates with the rendering nodes, the only required setup is the address and port of the control node and possibly a password. The control node ultimately processes the user input and forwards it to the rendering nodes. Currently, ZAPP provides mobile launchers for Apple's iOS device family.

2.4.2 Development

Every ZAPP application consists of two parts. The first is the server that runs distributed on the rendering nodes and displays the visual content. The second is the client that runs on the control node and is responsible to manage and synchronize the application state as well as issue interactive commands from a user to the servers. A ZAPP application may have a third part, the UI, which runs on UI nodes and provides the user with an interface for interaction.

ZAPP provides several APIs to support developing each part of an application. Templates integrating the APIs are available for server, client, and UI. The templates are a ready-to-go entry point for development.

The network and synchronization API takes care of communication between servers and clients, as well as between clients and the UI. The API includes functionality to synchronize the frame rate across the rendering nodes and make sure that the nodes display their content at the same time. The network library is platform-independent and applies to any context.

The display grid API works on the client to query information about the rendering nodes and their display alignment. It maps a 2D grid coordinate to the corresponding rendering node and display so commands can address a specific tile easily. ZAPP automatically passes information about the current display grid to an application on start-up.

To simplify the usage, both the network and the display grid library are self-contained and do not have third-party dependencies.

The multi-monitor rendering API is available for the servers. It provides information for each available graphics adapter and the displays attached to it. The developer can use the API to

manage rendering to multiple displays, which the server application template already incorporates. The API also supports generating offset projection matrices for each tile to represent a correct global projection on the grid. The application has the option to either ignore or consider a tile's bezel when generating a matrix. Furthermore, the library provides an abstraction layer that encapsulates common functionality of DirectX 10 and 11.

Finally, the mobile rendering API works on the UI nodes. It encapsulates common functionality of OpenGL ES1.1 and 2 to support developing rendering features on the UI side of an application. An application may want to provide the user with a preview while rendering the final result on the display wall.

While using the aforementioned APIs and application templates greatly simplifies creating or integrating applications, none of these features are mandatory. Our framework is ultimately a management solution. Thus, in the end, the developer is responsible to efficiently distribute an application across the available rendering nodes and displays, which may include distributing data, assigning rendering tasks, performing load balancing, and choosing the appropriate rendering technique. In contrast, other generic frameworks [HHN+02a, EMP09] focus on the rendering side, and features like data distribution or the rendering technique may be built-in.

ZAPP is able to fully operate on a single machine with any number of displays. This facilitates local development and debugging before deploying the application to the production environment.

2.4.3 Interaction

While ZAPP allows to start applications directly from the control node, this is only intended for developers. A casual user's entry point is a UI node, which provides a launcher to execute and terminate applications. The launcher connects to the control node and provides a GUI to select from the list of applications. Figure 2.3 illustrates the options of the launcher. ZAPP allows to run an application on a subset of the display grid. Multiple applications can be active at the same time as long as their display space does not overlap. The launcher consequently lets the user select the displays the application should use.

If an application provides a specific UI for interaction, the launcher starts this part automatically on the UI node. The user has to install the UI of a ZAPP application separately to have access to the intended interactive features. The user can also join an already running application. ZAPP does not limit the amount of UI nodes that connect to an application. Consequently, the application may allow multiple users to collaborate. Both launcher and UI ideally run on a device familiar to the user.

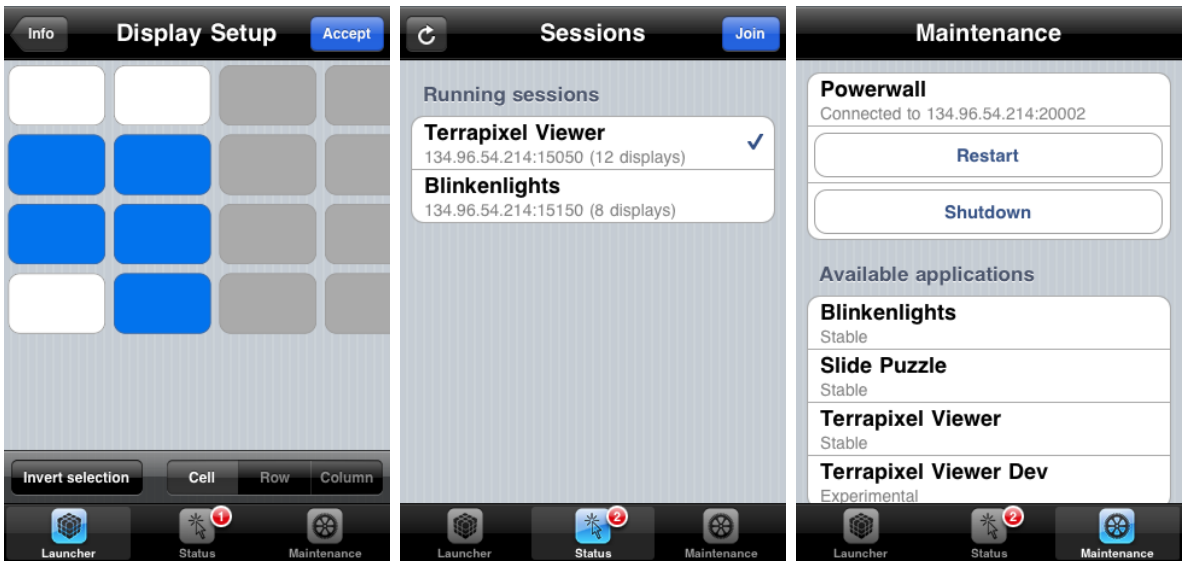


Figure 2.3: The launcher UI running on an Apple iPhone: selecting the displays for an application (left), joining an existing session (middle), and maintaining the run-time environment.

An additional feature of the launcher is to send a shut down command to the entire display hardware. This enables to quickly power down the display wall after a presentation. It is recommended to combine the feature with an automatic start-up of ZAPP on the control and rendering nodes, which enables to boot the entire system by simply turning on the power.

2.4.4 Applications

There are several existing applications for the ZAPP framework, ranging from a simple 2D sliding puzzle game to distributed, high-resolution volume rendering.

Figure 2.1 showcases the YottaPixel Viewer, which allows to explore arbitrary sized 2D images on the display wall. To quickly access an image at any level-of-detail, the application stores the image in a multi-resolution hierarchy with a quadtree. Currently, the theoretical tree depth and thus data set size is limited by the 64-bit indexing to address image tiles in the tree. To navigate through an image, the application provides a mobile, touch-based interface with a dual interaction mode that enables relative navigation if the user's focus is on the wall and additional absolute navigation features, for example via buttons, if the focus is on the UI device. The implementation contains a generic data input API to allow sources other than on-disk image files. Figure 2.4 shows a procedurally, in-situ generated fractal visualization.

An extension to the YottaPixel Viewer is the Multi-Resolution Painter, which in addition enables to interactively modify images at an arbitrary level-of-detail.

On the 3D rendering side, there is an interactive fish tank demonstrator that incorporates the

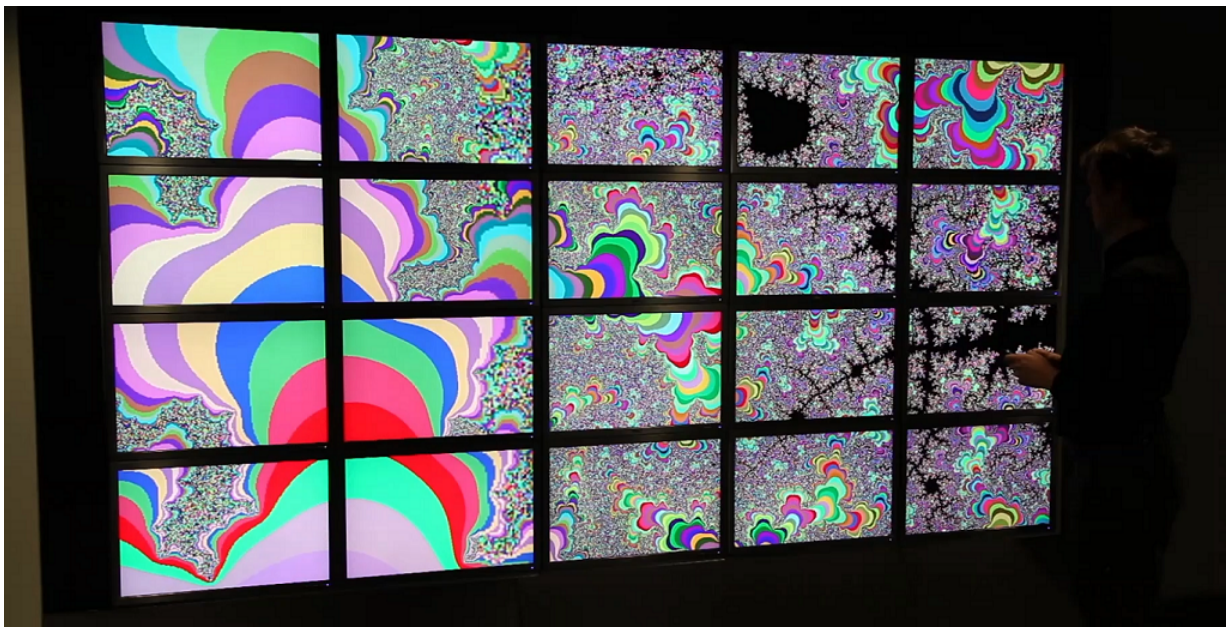


Figure 2.4: A user explores an in-situ generated fractal with the YottaPixel Viewer.

MorphableUI [KGB+16] library to implement its user interfaces.

We implemented a distributed visualization application for multi-resolution data sets, showcased in Figure 2.5. A rendering node may be connected to multiple physical displays. This can cause a performance bottleneck if the node has to generate the visualization for each display on its own. We therefore implemented a rendering library that allows both local and server-based rendering. The library is portable and flexible. Since we reuse it for the visualization applications in Chapter 3, Chapter 4, and Chapter 5, we outline the most prominent features here.

The library provides a generic API that enables to plug in different renderers. To support volume rendering, we integrated the *Tuvok* [FK10] framework. Library functionality includes progressive rendering of multi-resolution data sets, transfer functions for volume rendering, and the transfer of server-side data sets to enable client-side and hybrid rendering. To transport generated images to the client, the server supports JPEG and S3TC encoding. Chapter 5 gives details about S3TC, which is a very fast, parallel encoder with a fixed compression ratio.

In addition, the server integrates with DaaS [LPHS12] to allow streaming the rendered content to virtually any display. To improve the bandwidth efficiency and quality of the video stream, the server supports an application-specific importance mask that guides the encoding in a manner similar to Tizon et al. [TMP11]. The video encoder consults the mask to encode pixels based on their priority. In case of DaaS, this means adapting the H.264 quantization per macroblock. The renderer may generate the mask for each image based on information like depth, object curvature, edges, or important structures within the visualization. The mask

may also build on user input such as selected objects or a region of interest (ROI). A future direction is the use of eye tracking to determine the ROI.

Figure 2.5: A user explores a volume data set on the display wall.

Using the library, the ZAPP visualization application supports to set up a dedicated server for each display. However, *Tuvok* is an out-of-core system that can handle very large data sets, which require substantial resources to render at high detail. Therefore, in a future revision, we want to increase the scalability by allowing multiple servers per display. Even an adaptive approach that dynamically assigns servers to displays based on how much of the data set covers the display area is possible.

To interact with the visualization, we provide a touch interface on a mobile device that allows rotating and moving the data set. The UI displays a low-resolution equivalent of the visualization on the wall. For this purpose, the UI uses the same server-based rendering library as the rendering nodes.

2.4.5 Evaluation

To confirm that using ZAPP applications is indeed "director proof", we tested our mobile launcher installed on an iPod Touch with 15 non-expert users, who were asked to run the YottaPixel Viewer on a 82 megapixels wall consisting of 5x4 displays. The application showed a high-resolution world map. We asked the users to find and zoom in on Germany on the map. No user had prior experience with a display wall environment or ZAPP.

Each user was able to browse through the list of applications and then select, run, and interact with the YottaPixel Viewer. Especially users with prior iPod knowledge were able to finish the task instantly, while others were able to operate ZAPP with minimal instructions (for example explaining the touch display of an iPod).

We also noted some areas that could be improved. One user accidentally started another application at first. After closing the application, the user had to manually restart the launcher to return to the selection menu. However, we feel that launching and terminating applications consecutively should be one consistent workflow. We therefore want to allow the launcher to pop up again automatically after the user closes the current application. Also, even though feedback indicates that the interaction is intuitive, we want to incorporate help sections in the launcher and especially the custom applications to accommodate inexperienced users and give guidance for applications with less self-explaining interaction.

2.5 Implementation

This section details the software components running on the different nodes and the communication flow between the components.

2.5.1 Control Node Processes

The control node is the center point of the ZAPP framework. It is responsible for managing the available rendering nodes and their displays as well as all configuration settings. It also runs the client part of an application, which maintains an application's state. The control node runs a persistent controller process, which is a TCP server listening for incoming connections from rendering and UI nodes. Rendering nodes establish a persistent connection to the controller and periodically confirm their presence and the displays they have available. Accordingly, the controller updates the display grid configuration. When a rendering node first connects, the controller uniquely registers it using the node's name and physical address.

On initial start-up, the display grid is empty. The administrator thus needs to start another process, the manager GUI, to adapt the display grid to newly registered rendering nodes. Once the grid setup is complete, the controller automatically accounts for missing displays at runtime by deactivating the corresponding tiles in the grid. This happens when a rendering node or some of its displays become unavailable. Since there is no reliable way to identify a single display on a rendering node, should a display become unavailable, the controller does not know

the exact tile linked to it and therefore deactivates the tile with the topmost coordinate assigned to a display of the rendering node. Ultimately, the administrator should use the manager to revise the display grid if a permanent change occurs. For example, the removal or addition of a full column of displays requires resizing the grid. When rebooting the system, the controller automatically restores the tiles for a registered rendering node once it connects, so there is no need to setup the grid again.

The controller stores all settings in plain, human-readable text files. An expert user could thus quickly change settings without having to consult the manager. As the control node is the only point that holds the settings, a ZAPP configuration can easily be backed up and re-established by copying a few text files and replacing the files of another installation.

2.5.2 Rendering Node Processes

A rendering node is responsible for outputting visual content to its displays and thus runs the server part of an application. A rendering node runs a persistent daemon process in the background, which periodically iterates through the available displays and sends status information to the controller. The daemon is also a TCP server that listens for connections from the controller. The controller establishes a persistent connection to the TCP server to issue commands to the daemon. The commands include launching an application, forcing an application to exit, and shutting down the rendering node all together.

The current ZAPP daemon is a Windows service that builds on DirectX to obtain information about the available displays. Future extensions to ZAPP will include a platform-independent solution.

2.5.3 UI Node Processes

A UI node is responsible for providing the user with an interface for interaction and thus runs the UI part of an application. There is no persistent process running here. The user can start a launcher process that connects to the controller to request a list of available applications and the dimensions of the display grid. The request may or may not include applications flagged as experimental. The user can select an application for launching. By default, an application runs on the whole display grid. However, it is perfectly valid to restrict an application to a sub-grid as Figure 2.3 demonstrates.

The manager GUI on the control node also allows to browse through and launch applications,

but we intend this feature for developers only who are authorized to access the control node directly. In the following, we focus on the scenario involving a UI node.

2.5.4 Application Launch Communication Flow

Figure 2.6 illustrates the steps involved to launch an application. First, the launcher on a UI node sends a request to the controller to start a particular application on a sub-grid or the whole grid. In response, the controller starts the client process on the control node. The controller passes information about the sub-grid to the client, which for each tile includes the attached rendering node and display, as well as the tile's bezel. A tile may be flagged as deactivated if it is not currently linked to a display. Some displays might be unavailable or there might be a physical hole in the display wall, which the client needs to know to guarantee a consistent application behavior. ZAPP can easily be extended to support launching clients remotely on a different machine than the control node.

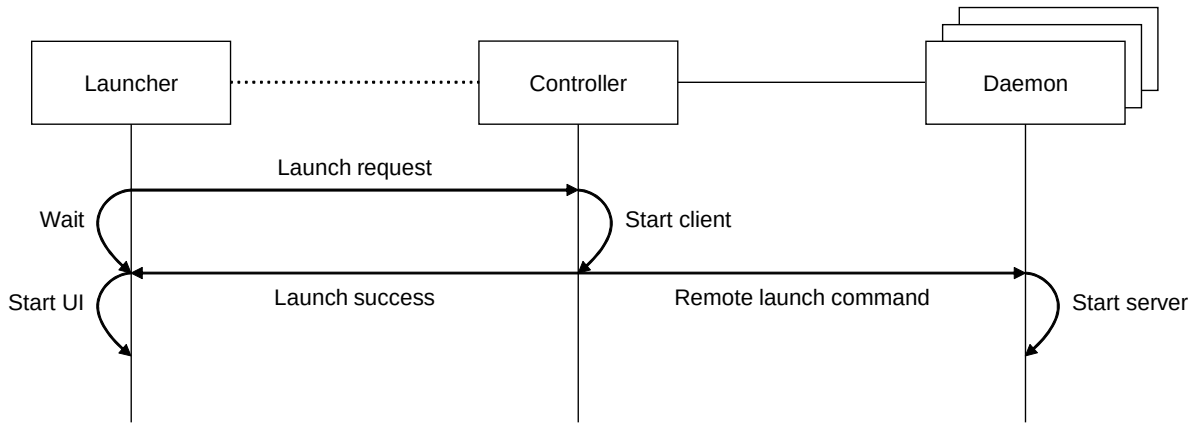


Figure 2.6: Communication flow between launcher, controller, and daemons when starting an application.

The controller also determines the rendering nodes whose displays are part of the sub-grid and then remotely starts the server process on these nodes. For this purpose, the controller sends a launch command to the daemon running on each rendering node. The command includes information on the displays to use, which may be a subset of the available displays of a rendering node in case the application runs on a sub-grid or some displays have not yet been linked to a tile. The daemon then starts the server process of the application.

Furthermore, the controller responds to the launcher to confirm the start of client and servers. The launcher finally starts the application-specific UI process to enable user interaction.

2.5.5 Application Run-time Communication Flow

After the start of the UI process, the involvement of launcher, controller, and daemons is over, and the application takes over. The servers and the UI establish a persistent connection to the client, which is the central point of communication and runs a TCP server of its own. From here, communication and features are application-specific and may be implemented in different ways. We focus on the common scenario depicted in Figure 2.7. The UI sends user input to the client, which processes it, updates the application state accordingly, and instructs the servers to update their displays. The client is especially responsible to keep the state of the distributed application synchronized. It therefore locks the application state until all servers have finished their work before accepting new input for processing.

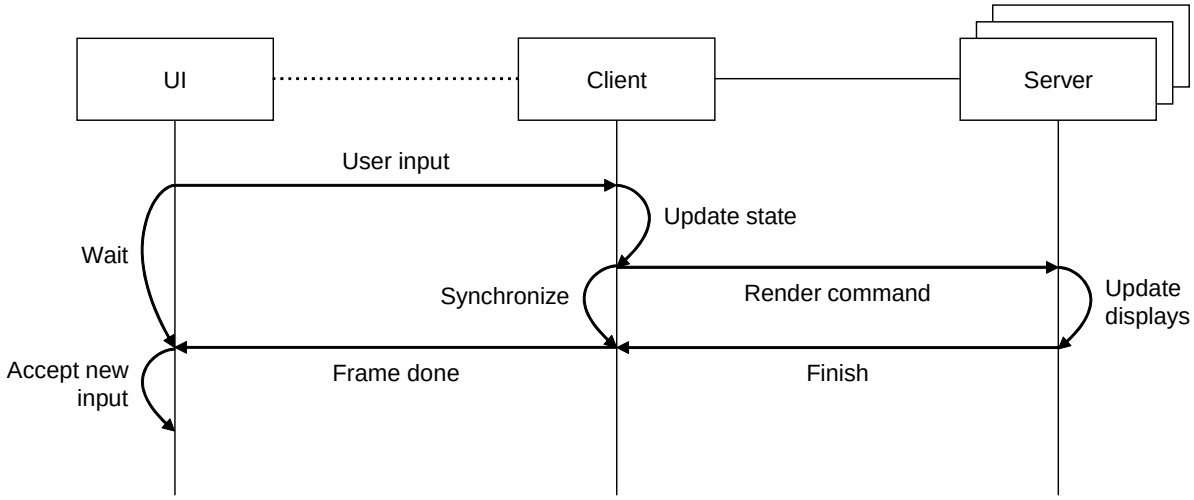


Figure 2.7: Communication flow between UI, client, and servers during application run-time.

In contrast to the reliable, high-bandwidth connection between client and servers, the connection between client and UI may be wireless. Should a user get disconnected or close the UI while an application is running, the user can reconnect at any time to regain control over the interactive features. The UI automatically attempts to recover if it loses connection.

To connect to the client, the servers and UI must know the client's address and listening ports. The controller is aware of the ports used by each application's client and attaches the information when remotely launching servers and UI. Other users may also start the UI on their device to join and interact with the running application. A user can consult the launcher to connect the UI automatically. The controller confirms the running application to the launcher, which then starts the UI and passes the client information. Alternatively, the UI may allow to manually enter address and port of the client, which enables to interact with ZAPP applications without the need to install the launcher.

For the UI node, it makes no difference whether the application was freshly started or already

present. However, if a consecutive user attempts to launch an application on a sub-grid that overlaps the grid another application already runs on, the controller rejects the request. As long as there is no overlap, it is valid to launch several applications or instances of a single one on the display wall. The launcher can request the space on the grid that is still available from the controller. The controller keeps track of the active applications on the grid and rejects or accepts new launch requests accordingly.

An application is responsible to terminate gracefully, and the UI should provide the user with a way to do this. The UI issues an exit request to the client, which forwards it to the servers. Since the controller needs to know about the exit to maintain the list of active applications, it holds a persistent, idle TCP connection to each client. Given that the client runs on the control node, this is a loopback connection. A disconnection indicates an application has exited. The user can also forcefully terminate an application through the launcher or the manager on the control node. The option enables recovery from a hanging application or should the UI get permanently disconnected from the client.

2.6 Conclusion and Future Work

In this chapter we presented ZAPP: a management framework for distributed, high-resolution visualization systems. To our best knowledge, this is the first solution that specifically targets the administration and launch process of programs on such a system. In an informal survey we validated that arbitrary casual users are able to operate our framework with minimal instructions.

There are several directions to improve ZAPP in the future. To simplify the installation process further, we plan to incorporate an automatic mechanism that allows the control node to discover the rendering nodes. We also plan to add an automatic system to layout the display grid. The system will incorporate the camera found in mobile devices to detect the alignment of displays.

Since we intend ZAPP to be a portable open source solution, the entire framework should run platform independently. This requires to revisit some parts like the multi-monitor rendering API that currently utilizes DirectX only and the rendering node background daemon that is a Windows service. In addition, we want to provide our demonstrative applications such as the volume rendering as open source.

A feature that requires a more thorough extension of ZAPP is the decoupling of the rendering nodes from the displays. Here we intend to allow a rendering node to stream its output to any display that is accessible over the network. Chapter 5 examines a set of image encoding methods

that we could incorporate. The functionality would substantially increase the flexibility of the system and provide a built-in mechanism to include rendering machines in addition to the ones directly attached to the displays.

Chapter 3

Mobile Visualization for the Selection of Deep Brain Stimulation Parameters

3.1 Introduction

Neuromodulation is the alteration of neural activity by means of implanted devices. Most neuromodulation systems consist of a multi-electrode lead that is surgically implanted in the brain and connected to a subcutaneous implantable pulse generator (IPG) in the torso. The basic concept behind neuromodulation is that stimulation-induced current flows from electrode(s) through surrounding brain tissue, which in turn causes a therapeutic functional response. One important example of this approach is deep brain stimulation (DBS), which is an established therapy for treating the motor symptoms of Parkinson's disease (PD) [DSBK+06, WFS+09], as well as a variety of other disorders [SH08]. Figure 3.1 illustrates the Medtronic DBS system, which consists of an electrode lead with four cylindrical contacts and an IPG that delivers continuous stimulation. The stimulation parameters are selected post-operatively and are titrated to provide good therapeutic benefit with minimal side effects.

A persistent problem with neuromodulation techniques such as DBS has been the selection of stimulation settings for optimal response. To achieve this, patients must often undergo lengthy and repeated clinic visits to determine the best settings. A study by Hunka et al. [HSW+05] found that the total time spent programming the stimulator and assessing DBS patients ranged from 18-36 hours per patient. Part of the reason for this length of time is the amount of trial and error involved in choosing the best stimulation protocol without any visual guidance on the location of the electrode or the effects of stimulation on nearby brain tissue.

This approach has persisted for decades, primarily because the computational tools necessary to

Figure 3.1: Overview of the DBS system. The DBS electrode is implanted in the brain during stereotactic surgery. The electrode is attached via an extension wire to the IPG, which is implanted in the torso. The entire system is subcutaneous and designed to deliver continuous stimulation for several years at a time.

visualize the effects of stimulation were not available. While significant progress has been made over the last years in the sophistication of computational models available for neuromodulation, very few of these have been introduced into clinical practice for several possible reasons: complex software that lacks a simple interface; complex visualizations that are difficult to interpret; and new software is perceived as increasing the demands on clinicians who are often under intense time pressure.

In this study we set out to demonstrate how patient-specific models of DBS can be combined into a decision support system that clinicians can easily use at the point of care. We developed ImageVis 3D Mobile (IV3Dm), a visualization system that runs on commodity mobile devices. We also developed a data distribution framework that interfaces with IV3Dm and allows users to access new data sets with minimal effort.

We hypothesized that IV3Dm would enable clinicians to choose DBS parameters that are comparable to standard of care but in much less time. Furthermore, simplified interfaces common to mobile platforms would lower the barrier to entry and be more readily accepted. We chose devices that clinicians are accustomed to in their daily routines: their smartphones. In the current setting we use iPhone class hardware (this includes iPod touch and iPad devices), but the concepts presented here are in no way restricted to this platform. The IV3Dm visualization environment consists not only of interactive volume and geometry rendering implementations but in particular also contains means of receiving and exploring data as well as sharing it between devices. Both renderers can coexist, enabling seamless interleaving of volumes and

semi-transparent geometry. To transfer data to a device we utilize techniques from instant messaging systems, which allows the user to view a new data set with literally a single touch.

We tested our hypothesis by asking five clinicians who have extensive experience with DBS programming to choose stimulation settings using patient-specific computational models of DBS visualized with IV3Dm. We compared the DBS settings and time required to retrospective data that was gathered during standard clinical care independent of the study. We show that mobile visualization of patient-specific DBS models has compelling features for clinical decision making.

The remainder of the chapter is structured as follows: The next section presents related work. Section 3.3 then gives an overview of the study. In Section 3.4, we describe the functional details of our visualization environment. Section 3.5 and 3.6 describe the evaluation and discuss the results achieved. We conclude with predictions based on our results and possible future directions.

3.2 Related Work

A body of work has emerged on computational methods to predict the effects of neuromodulation therapy. However, it has proven difficult to create a visualization system that can be integrated into clinical care.

3.2.1 Computational Models of DBS

Computational models have been developed to predict and visualize the effects of DBS on an individual patient basis [MMN+07, BM05, BMM06, BM06, BM07, BM08, MMS+04]. Briefly, finite element models that are derived from patient medical image volumes are used to determine the location of the electrode in the brain, calculate the bioelectric fields produced during stimulation, and predict the neural response to the applied electric field. The primary outcome of this approach is a model-predicted volume of tissue activated (VTA), which is the region of neural tissue that is affected by DBS. Figure 3.2 shows the visualization of a VTA and its surroundings.

The computational models have been validated by comparing model-predicted outcomes to clinically measured responses in PD patients [BCHM07, BCHM06], have been used retrospectively and prospectively to determine how activation of certain anatomical regions is correlated with motor [BCH+11, MBW+09] and neuropsychological [BDH+10] outcomes in PD, and have been

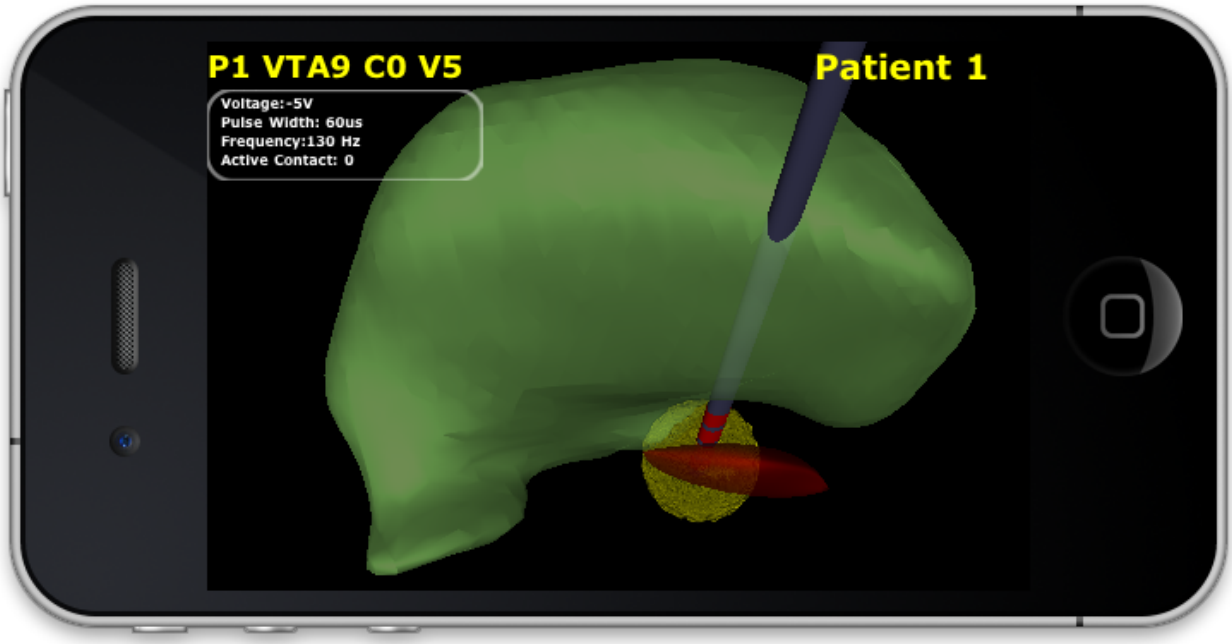


Figure 3.2: IV3Dm visualizes a patient-specific model of DBS. The model shows the location of the electrode lead relative to the surrounding nuclei in a Parkinson's disease patient. The model-predicted VTA during DBS (yellow part) surrounds the distal electrode contact. With this model it is possible to view the overlap between the VTA and nearby anatomical structures, which is a key feature in clinical decision making when choosing stimulation settings.

shown to guide clinicians to select stimulation parameters that improve cognitive and motor outcomes [FWN+10]. However, two problems persist:

- DBS programming is still often performed without any visual guidance on the location of the DBS electrodes or the effects of stimulation on surrounding structures.
- The software required to perform the visualization can require significant training and is not widely available in a clinical setting.

Hence, there is a need for a simple, intuitive application that can visualize the effects of DBS on an individual basis to facilitate clinical decision making.

3.2.2 Visualization on Mobile Devices

Preim and Botha [PB13] outline the importance of visualization for medicine in various areas like diagnosis, treatment, and education. While early volume rendering systems required supercomputers and expensive graphics subsystems, over the years hardware requirements have become more and more relaxed. Nowadays, commodity PCs and even notebooks are sufficient to visualize even large data sets interactively [FK10].

In parallel with commodity hardware, mobile devices have caught the attention of the visualization community as another viable and interesting platform. Even before today's powerful mobile devices were available, Encarnação et al. [EFK95] discussed the general issues in using mobile devices to obtain and access data. Paelke et al. [PRR03] discuss user interface design aspects for mobile devices. Chittaro [Chi06] focus on the general issues of visualizing content on mobile devices.

Burigat and Chittaro [BC05] describe a VRML-based system for visualizing what users see as they roam a city. Park et al. [PKI08] developed a system for collaborative medical visualization, using parallel server-based volume rendering techniques. Meir and Rubinsky [MR09] investigate the use of mobile devices as a cost-effective component of a distributed system for performing ultrasounds. Their system employs simple-to-use, inexpensive client-side devices that generate ultrasound data. The client sends the data to a server, which performs volume rendering at pre-defined camera angles and sends the images back to the mobile devices for analysis in the field. Lluch et al. [LGCV05] present a server-based surface rendering system. The server holds a scene graph and uses it, along with client view information, to select an appropriate resolution from a multi-resolution representation on disk. Scene access is done in an out-of-core fashion, allowing the server to visualize very large models.

Even when rendering is done on the server, for demanding visualizations a single machine may not be able to provide updates to the mobile device quickly enough. For this reason, Lamberti and Sanna [LS07] introduce a Chromium-based [HHN+02b] rendering system that encodes generated images as MPEG video and streams the video to the mobile device for decoding and display. In Chapter 6 we present a distributed rendering framework that allows a browser-based client to access a real-time ray-tracing rendering cluster. Using the browser as a platform for portable and mobile visualization applications becomes increasingly popular [CSK+11, MF12, MPJ+13]. Chapter 5 describes a visualization system that supports server-based volume rendering in a web page.

When capable mobile devices became available, Chang and Ger [CG02] implemented a ray-caster for opaque geometry on PocketPC devices. Their system realizes a server-backed rendering model that allows desktop machines to accelerate rendering on the mobile device. They argue that the performance of ray-casting and ray-tracing approaches is dominated by the number of pixels, and therefore mobile devices, where hardware capabilities are expected to grow but screen sizes will remain relatively stagnant, are a perfect fit for these approaches. However, a more recent study [RA12] demonstrates that limitations of the mobile hardware and graphics APIs as well as the introduction of high-resolution screens such as Apples retina displays can hamper ray-casting compared to texture-based volume rendering.

Texture-based volume rendering has positioned itself as a powerful tool for interactive visual analysis of volumetric data sets [Fer04, HKR+06, Krü10]. Hadwiger et al. [HKR+06] describe a method that is particularly efficient for mobile devices that lack 3D texture support, which was the prevalent case at the time of this study. Our volume renderer therefore builds on their work. Also related to our rendering subsystem is a contribution by Moser and Weiskopf [MW08]. In particular, our OpenGL ES 1.1 volume renderer is based on their findings.

Schiewe et al. [SAK15] reuse the mobile visualization system presented in this chapter. With the advent of mobile devices that support modern graphics APIs like OpenGL ES 3.0 with 3D texture support and even low-level APIs such as Metal [App16a], they extend IV3Dm's local volume rendering features with ray-casting. Ray-casting does not suffer from the view-dependent artifacts that texture-based approaches can produce. A comparison between the different rendering techniques shows the performance advantage of Metal over OpenGL ES 3.0 and of server-based rendering to save battery life.

In addition to the rendering subsystem that offers an intuitive touch-based user interface, this chapter presents a flexible, easy-to-use data set management and distribution framework that enables the minimally invasive integration of the software in the medical environment. We achieve a broad acceptance of the system from the clinicians that used it during the study.

3.3 Overview

In this chapter, we present the mobile visualization system IV3Dm and its deployment in a real-world environment to support clinical decision making in DBS for PD patients. Using our system, clinicians were able to make decisions similar to current standard practice but in substantially less time.

Under the current standard of care for DBS, patients return to the clinic a few weeks after implantation of the system for their initial programming. If the results for the initial stimulation settings are not satisfactory then more complex stimulation protocols are considered. The process can include substantial trial and error, which is partly attributable to the lack of visualization of the patient anatomy or the effects of stimulation.

In this study we evaluate the accuracy and speed of DBS programming using IV3Dm compared to standard of care. To do so we identified four Parkinson's disease patients who previously received DBS leads implanted in the subthalamic nucleus and were good responders to the therapy. We then constructed patient-specific models of DBS and provided them to the clinicians in IV3Dm. The clinicians were blinded to the actual identity of the patients and were asked to

use IV3Dm to determine the best electrode contact to use for monopolar stimulation as well as the stimulation amplitude that would provide the best therapeutic benefit with minimal side effects. We compare the values chosen in the study to those used for each patient's clinical DBS settings, which were determined through standard medical care outside of this study. Lastly, we determine the amount of time necessary to program patients using IV3Dm compared to the time required for standard clinical practice, which was estimated using data gathered from the patients' medical records.

3.4 Visualization System

We prepared the technical framework and the data sets to enable the study that was conducted by our partners at the Medical College of Wisconsin.

IV3Dm is a mobile, interactive visualization system for volume and geometry data, implemented for Apple's iOS software platform. iOS runs on a large number of devices and is the platform of choice for our target users in the evaluation, who are familiar with the user interface and interaction metaphors the platform provides. As the hardware specifications of iPhone, iPod and iPad reflect the design of many other mobile devices, our findings in this evaluation are applicable to a wide range of mobile hardware.

In the following subsections we outline the main components of IV3Dm, the rendering system and the data transfer. We conclude with a description of the evaluation data sets rendered in IV3Dm as seen by the clinicians.

3.4.1 Rendering

IV3Dm provides volume and geometry rendering capabilities, which have been implemented to support both OpenGL ES 1.1 and 2.0. We decided to support the fallback to OpenGL ES 1.1 to achieve wide availability even on legacy devices. Due to the lack of support for 3D textures in OpenGL ES at the time of the study, the renderer uses three axis aligned stacks of 2D textures to access volumetric data on the GPU as described by Hadwiger et al. [HKR+06]. The volume renderer implements manual trilinear filtering and volumetric lighting in OpenGL ES 2.0.

A key feature of IV3Dm is to render multiple data sets interleaved. Rendering volumes and geometry together is required for the evaluation since geometric data of a patient's nuclei including the placed electrode shaft needs to be overlapped with VTAs, which indicate the effects of DBS. The requirement favors texture-based volume rendering over ray-casting as the

former allows the straight-forward combination of both rendering techniques. The renderer sorts semi-transparent geometry in back-to-front order in each frame and inserts the geometry in-between the volume texture slices.

IV3Dm is a flexible visualization environment with a focus on usability. Further feature highlights are an editor to draw and apply transfer functions for volume rendering, landscape mode rendering, means to iterate over data sets quickly without leaving the rendering view, and a 2D rendering mode that allows to examine the individual 2D texture slices of a 3D volume. IV3Dm can automatically translate user interactions in the 2D view to the 3D view and vice versa, which facilitates alternating between both views when examining a data set.

When using a touch enabled display such as our target platform, the user expects the data to move in sync with their finger, otherwise the fingers and the data set feel decoupled. Therefore, IV3Dm provides a number of options to increase rendering speed during periods of interaction, such as a reduction of the rendering target resolution, the texture sampling quality, the volume quality, and an option to disable lighting on interaction. In addition to these means in the volume renderer, the precision of visibility sorting can be reduced to speed up the geometry rendering as well.

While for this study, a stripped-down visualization that utilizes IV3Dm's local rendering capability is adequate and even beneficial due to its simplicity, the software also supports server-based rendering. For the implementation, we use our rendering library introduced in Section 2.4.4. IV3Dm can connect to a rendering server that hosts a list of data sets for remote visualization. The server-side out-of-core volume renderer allows to interactively view arbitrary sized data sets on the mobile device. Figure 3.3 shows the display of a volume that is impractical for local storage and rendering in IV3Dm.

3.4.2 Data Transfer

IV3Dm is the mobile counterpart of the desktop visualization system ImageVis 3D, which builds on the volume rendering library Tuvok [FK10]. To generate data sets for rendering on the mobile device, we extended Tuvok's modular IO subsystem with the capability to write out IV3Dm files. This allows our pipeline to accept a number of volume and geometry formats and convert those automatically into IV3Dm data. Amongst the formats are SCIRun [CS11] volumes and geometry in which the input data for this study is stored.

While IV3DM provides flexible rendering options and an intuitive interface for interaction, users must also be able to obtain new data sets. In our particular use case, simple and fast data transfer not requiring any technical expertise is a key feature to embed the system seamlessly

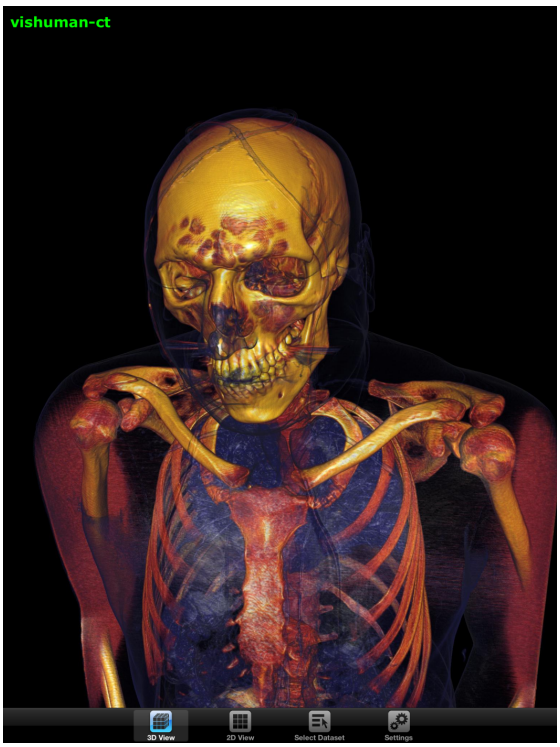


Figure 3.3: IV3Dm displaying the Visible Human data set [U.S12] (512x512x1884 8-bit voxels) via server-based volume rendering.

into the time-critical clinical workflow. Therefore, transfer via a direct cable link from a storage machine is not desirable for several reasons. First, it requires direct access to the machine, which is most likely located in a computer laboratory away from the point of care. Second, the solution requires expertise on how to connect a device to the storage and then how to use additional software to select and transfer data sets.

IV3Dm therefore provides several ways to access and receive data sets on the go. In all cases, the user's single point of interaction is IV3Dm.

In addition to built-in example data that comes with a new IV3Dm installation, the user can download data sets from a server over a wireless network connection. For this purpose, IV3Dm provides a simple menu that allows to select one or multiple data sets for downloading. The user can continue using the application while downloads are ongoing. If desired, the user can abort the download at any time. As an alternative to data servers, IV3Dm supports the exchange between devices with a Bluetooth connection. To speed up the transport in the wireless network, IV3Dm can receive *Deflate* [Deu96] compressed data sets.

Even though data servers are already a feasible option, they still require the user to visit a custom menu to set up the connection and another menu to select data sets. The inter-device exchange detects compatible devices automatically but restricts the exchange to a close range, which is impractical for the clinicians who must remain mobile in their daily routines.

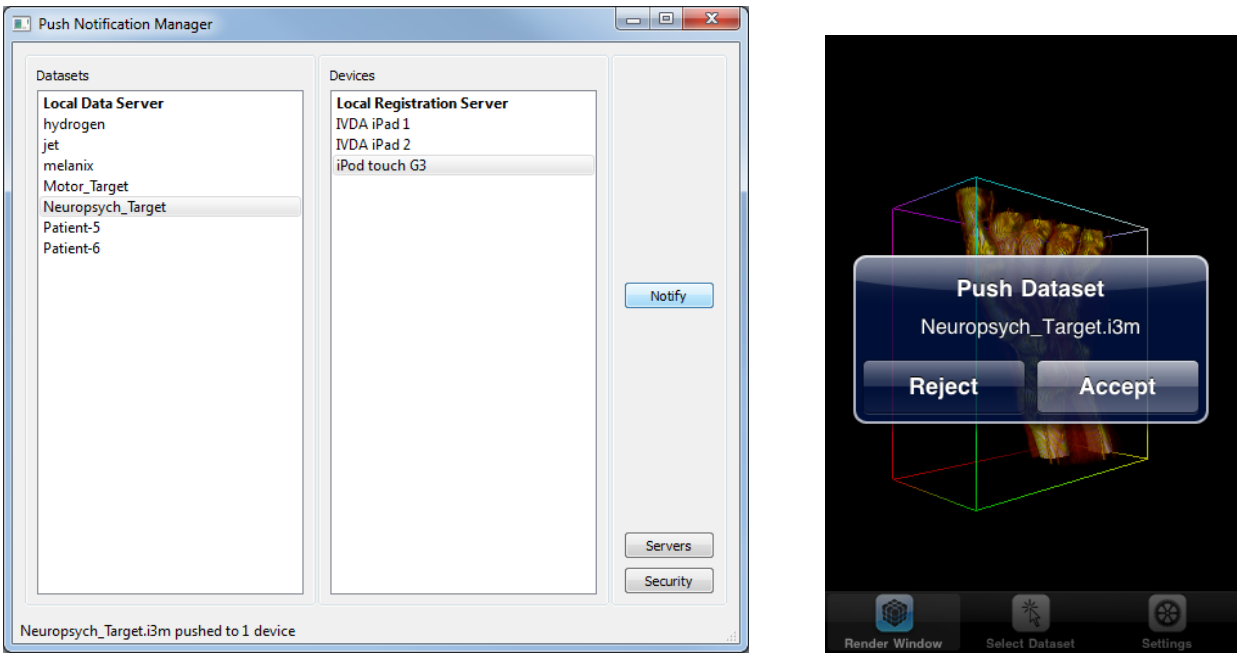
Ultimately, both approaches require the clinician to open IV3Dm and check for new data sets.

To avoid these limitations, IV3Dm additionally uses instant messaging technology to automate the data transfer from the user perspective as much as possible. The data distribution system thereby builds on a particular technology of the iOS called *Push Notifications* [App16b]. Push Notifications are Apple's means of being able to send messages to devices without the need to constantly run custom receiver software on the device. Instead, a single daemon runs persistently and distributes instant messages to any application that supports the feature. The general concept also applies to mobile devices that do not offer Push Notifications or a similar alternative. On these devices, IV3Dm could run in daemon mode while listening on a network port.

Using Push Notifications, we support a transfer mechanism that is initiated by the sender. A central management application keeps track of the available user devices and data sets and notifies devices about the availability of new data. Figure 3.4 a) depicts the management GUI. The manager connects to one or several data servers to obtain the list of data sets. These data servers are the same servers IV3Dm is able to connect to. In addition, devices that run IV3Dm are able to register with one or several registration servers, which provide the manager with the list of devices. Optionally, the operator can set up a password to prevent unwanted devices from registering. The operator can select one or multiple data sets and push a notification about these data sets to one or multiple devices. The manager passes notifications to the Apple Push Notification Service, which delivers them via an accredited and encrypted IP connection to a device immediately or as soon as the device comes online. When the notification arrives, a dialog window appears allowing the user to accept or decline the download as illustrated in Figure 3.4 b). Acceptance automatically starts IV3Dm if it is not already running and initiates the download and display of the data. The whole process requires just a single tap from the user who is only interested in reviewing the data sets. Should a user decline the download in the notification dialog, it is still possible to access the data later by connecting to the corresponding data server. This way, the user can access the data sets at any point without requiring another Push Notification.

To simplify the initial setup of the management application, the manager comes with a built-in data and registration server. The operator could therefore simply run the manager on the machine that hosts and probably even generates the data sets. However, the possibility to set up multiple data and registration servers on different machines gives the flexibility to represent a distributed infrastructure with various groups of data sets and devices.

The push-based data distribution framework moves the task to select and distribute data sets from the user to the service provider, thus achieving the goal of a minimal invasive integration



a) b)

Figure 3.4: a) The management application enables to notify registered devices about new data sets. b) A notification dialog appears on the device and allows the user to obtain and immediately view a new data set with a single tap.

in the clinicians' demanding schedule. In a future revision, we plan to relieve the operator by introducing delivery features that push certain data sets automatically to their target users.

3.4.3 Evaluation Data Sets

The patient data sets for the evaluation have a geometric and a volumetric component, which Figure 3.5 illustrates. The geometric component consists of surface representations of nearby anatomical nuclei (thalamus and subthalamic nucleus) as well as the DBS lead and electrode contacts. We deliberately chose geometrically simple surfaces of common anatomical structures that mimic the types of atlas representations that physicians are likely to be familiar with. We constructed the anatomical surfaces by coregistering each patient's magnetic resonance imaging to an atlas brain using a 3D nonlinear warping algorithm [CJM97]. We constructed surfaces for the DBS lead and electrode contacts using SCIRun [CS11]. The volumetric component is the VTA. In total, we provided 36 VTAs for each patient (9 for each electrode contact, representing a range of voltages from -1V to -5V, all at 130 Hz, 60 μsec pulse width). While indeed special desktop software is required to produce IV3Dm-compatible visualization data from raw input, the process is independent from IV3Dm's simplified interface and can be automated.

Figure 3.6 shows both components interleaved in IV3Dm. The rendering view provides annotations to distinguish between patients (the geometric component, top right), as well as to convey

a) b)

Figure 3.5: a) Patient thalamus (green), subthalamic nucleus (red), and DBS lead with four electrode contacts. b) Volume of tissue activated (VTA).

the VTA (top left). The view also provides the DBS stimulation settings for the VTA. The user can simply iterate through the VTAs for a patient by tapping the VTA annotation. Tapping the geometry label allows to switch between patients. The user can also leave the rendering view and go to the data set selection menu to browse through and load data sets.

The combination of the visualization components is ideally suited to our evaluation for several reasons. First, the use of geometric and volume components allows us to visualize each in their native format as generated in SCIRun. Second, the text annotations provide details necessary for the users to know which patient and stimulation settings are being evaluated. Third, the overlay of volume and geometry data allows the user to quickly determine the amount of overlap between the VTA and nearby anatomical structures, which is the feature that most strongly guides the decision making.

The focus in this study is on a simple visualization that highlights the crucial features and is suitable for distribution over a wireless network. As a result, the total data size per patient that needs to be transferred over the network is only around 1.48 MB.

3.5 Evaluation

In order to evaluate the utility of IV3Dm for clinical decision making, we constructed patient-specific models of four PD patients who were good responders to DBS. The models were created in SCIRun using previously described methods [BCHM07] and subsequently transferred to IV3Dm. We provided the models to five clinicians (three movement disorders neurologists, one

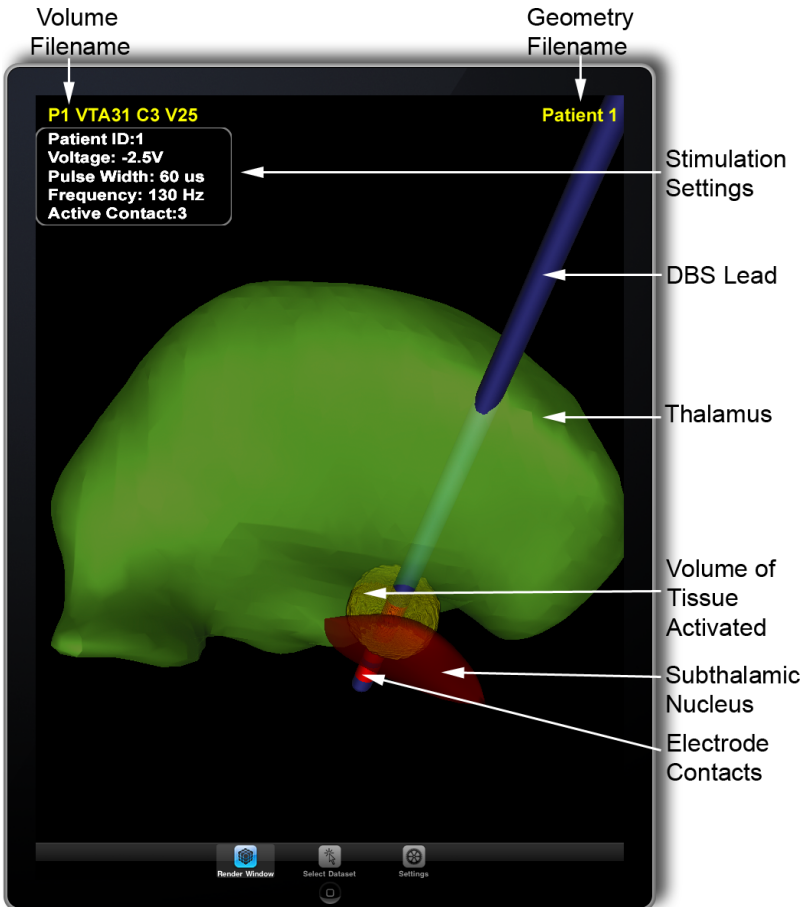


Figure 3.6: The interleaved rendering view in IV3Dm shows the VTA for -2.5V at contact 3 for a specific patient.

neurosurgeon, and one nurse) who have extensive experience with programming DBS systems for PD patients. We asked each clinician to select DBS parameters using IV3Dm on an iPad. We compare their selections to data collected via standard of care, along with the amount of time required.

3.5.1 Standard of Care

PD patients who were examined in this study received DBS via standard of care. Four to six weeks after surgery the IPG is turned on for the first time. The clinician works with the patient to determine the stimulation parameters that provide the best therapeutic response with minimal side effects. This is done through a process of activating each of the four individual electrode contacts and testing a range of stimulation parameters (voltage, pulse width, and frequency). The process is usually performed over several visits to the clinic. The patients examined in this study had an average of three to four visits requiring over four hours of time with a clinician to perform DBS programming.

3.5.2 Training

Prior to the experimental protocol, we trained each clinician as follows:

1. We informed the clinician of the objectives of the study.
2. We showed the clinician an example data set in IV3Dm on an iPad and demonstrated the following interactions: rotating, translating, and scaling models in the rendering view; loading individual patient models of anatomical nuclei and electrode location; overlaying and selecting VTAs.
3. After the demonstration, we gave the clinician the opportunity to have hands-on experience with IV3Dm.

The total training time was approximately ten minutes for each clinician.

3.5.3 Experimental Protocol

After training, we conducted the following experimental protocol with each clinician to evaluate IV3Dm for DBS parameter selection:

1. We announced a patient DBS model via Push Notification so that the clinician could transfer the model to an iPad running IV3Dm.
2. The clinician loaded the patient model in IV3Dm. While the clinicians involved in the evaluation previously treated the patients in our study, patients were anonymized, and clinicians were blinded to their identity. It was not possible to determine the patient identity from the IV3Dm visualization.
3. We asked the clinician to select the most appropriate electrode contact for stimulation based on the location of the DBS electrode relative to nearby anatomical nuclei (thalamus and subthalamic nucleus (STN)).
4. The clinician loaded VTAs for the chosen electrode contact, starting with -1V amplitude. On-screen annotations provided verification of stimulation settings.
5. The clinician stepped through a range of VTAs from -1V to -5V in 0.5V increments for the chosen electrode contact. From the range, the clinician selected the most appropriate voltage value.

6. The clinician could choose a different electrode contact and repeat the previous two steps if none of the VTAs seemed appropriate.

We repeated and timed these steps for each patient.

3.6 Results and Discussion

We found that the amount of time required to choose stimulation settings is significantly lower using IV3Dm compared to standard clinical care. Selection of stimulation settings required an average of 1.7±0.8 minutes per patient across all clinicians using IV3Dm, compared to an average of 4±1.4 hours required for programming via standard of care to reach stable settings with good therapeutic response (usually within three to four clinic visits). In addition, we found that the stimulation settings chosen using IV3Dm are very similar to those selected via standard of care. Table 3.1 shows that the voltages selected using IV3Dm are generally equal to or smaller than the voltages selected using standard of care, and in fact this is a trend that has been observed previously [FWN+10]. Table 3.2 shows that the active electrode contacts chosen using IV3Dm are either the same as or adjacent to the contacts chosen using standard of care. Prior studies have noted comparable therapeutic benefit from more than one electrode contact [OFW+09]. Hence, we consider this degree of variability to be within the range that is observed clinically.

Table 3.1: DBS voltages chosen with IV3Dm versus standard of care.

Patient ID	Standard of Care	IV3Dm (Average)
1	4.1V	2.35±0.34V
2	2.3V	2.4±0.74V
3	2.5V	2.05±0.76V
4	2.2V	2.0±0.71V

In addition, feedback on this system from clinicians is very positive. The user interface is intuitive, especially for existing iPhone users. The ability to interactively visualize patient models provides a level of understanding that is not currently available. The clinicians perceive the system as a welcome alternative to the current process and are optimistic about the long-range potential to provide the optimal DBS therapy more rapidly than previously possible. Hence, the salient features of IV3Dm for clinical decision making are to easily retrieve data, view the DBS electrode location relative to surrounding anatomy on an individual patient

Table 3.2: Electrode contact (C) chosen with IV3Dm versus standard of care.

		Clinician Number				
Patient ID	Standard of Care	1	2	3	4	5
1	C2	C2	C3	C3	C3	C3
2	C2	C2	C2	C1	C1	C1
3	C2	C1	C2	C1	C1	C2
4	C1	C2	C2	C2	C1	C2

basis at the point of care, view how the DBS-induced VTAs overlap with nearby anatomical structures, and interact with the visualization using an intuitive touch screen interface.

In this study the clinicians were not provided with any information on how VTAs should be selected relative to their overlap with surrounding anatomical structures. In fact, the verbal feedback they provided during the experiment indicates slightly different approaches to parameter selection: three of them tried to maximize VTA overlap with the STN; one chose VTAs that were superior to the STN; two tried to avoid VTA overlap with thalamus as much as possible. This reflects ongoing discussion in the DBS community about optimal target locations for stimulation, and we feel that this accounts for some of the variance in our results. Consequently, even with detailed visualization of patient-specific data, there is not currently a consensus on the best stimulation target for PD patients.

3.6.1 Interpretation and Potential Influence on Clinical Workflow

Our results show a dramatic decrease in the time required to select stimulation settings using IV3Dm compared to standard of care. However, an important question remains: what is responsible for the time difference? There are several possible explanations that may not be attributable to IV3Dm. First, during standard of care patients often receive a brief motor exam after each change in DBS parameters. This was not part of our study design because the clinicians were blinded to the patients' identities, and the goal was to evaluate the utility of the software to select DBS parameters.

Second, while our study focused on the selection of DBS voltage alone, clinicians sometimes also explore pulse width and frequency during initial programming. These variables were fixed in our study. Current guidelines suggest that good response to DBS can be achieved with pulse widths ranging from 60 μsec to 210 μsec and frequencies from 130 to 185 Hz. Hence, the

parameter space that is explored during standard of care is somewhat larger than the range that we tested. However, this is not a limitation inherent to our system, which we can extend to support additional variables.

Third, each patient model required between 30 and 60 minutes of preparation by a trained technician prior to transferring the data to the mobile device, though we anticipate that this amount of time could be reduced in the future by creating a semi-automated system for model generation.

Despite the differences in the two approaches and the difficulties of making direct comparisons, we believe that the above factors cannot completely account for the large effect size we observe (a 99.5% reduction in the amount of time required to choose DBS parameters). In addition, our approach could facilitate a fundamentally different clinical workflow. Specifically, the use of IV3Dm and patient-specific DBS models could allow the clinicians to quickly converge on a small range of parameters that are likely to provide good therapeutic response. From these initial settings we anticipate that the clinicians will evaluate motor outcomes while exploring nearby settings. Thus, instead of performing a comprehensive review of motor outcomes at a wide range of stimulation settings for all DBS contacts, clinicians could focus their effort on a much smaller parameter space prior to beginning motor exams.

The ability to access the visualization on mobile computing devices is an important feature. As indicated earlier, clinicians became proficient at using IV3Dm for DBS parameter selection in very little time. We believe this is a reflection of the simplified means to obtain and interact with data sets as well as the representation of information such as electrodes, anatomical nuclei, and VTAs in a familiar manner.

While we did not compare IV3Dm to an equivalent desktop-based system, we anticipate that the latter would require clinicians to spend substantially more time to become used to the interface and access their data sets for review. The use of mobile devices with wireless data delivery is convenient for the clinical workflow and does not require clinicians to rely on stationary computers that might not be available at the point of care. Significant attention has been paid recently to the role of mobile computing devices in a clinical environment for this very reason. Also, utilizing the users' very own mobile devices could enable to establish a system like IV3Dm without considerable investments into dedicated rendering hardware. Hence, we believe that our implementation could be a welcome addition to a healthcare delivery system that is attempting to reduce reliance on desktop-based architectures.

3.6.2 Insights into Visualization Applications

We believe that some of the developments in this study are of much broader interest. In particular:

- **Use of Instant Messaging for Data Distribution** Most of the systems outlined in Section 3.2.2 focus mainly on the renderer and present efficient ways of visualizing data fast and at high quality. While this is a important characteristic of a visualization environment, the deployment in a productive setup also requires sufficient means for the users to access the data that they are interested in. We provide a flexible data distribution architecture that moves the effort to manage data sets from the user to the publisher side. We employ a push-based transfer mechanism that builds on instant messaging technology to notify users about new data anywhere over a wireless network.

- **Natural Multi-Touch Interfaces** While multi-touch technology dates back to the early eighties [Bux07], only in the last decade, after the introduction of the iPhone, have such devices become popular. In a short period multi-touch displays have become available for a wide range of hardware (for example large display systems, workstations, mobile devices). While we are certainly not the first to point out this fact, we believe that in particular visualization applications and their acceptance can benefit significantly from the integration of touch-based interaction metaphors.

- **User Familiarity** As most people spend quite a decent amount of time per day using their smartphones, it seems only natural to use the smartphones for as many tasks as possible, sometimes even if this means passing on a more capable hardware environment. Interestingly, in this work we found that clinicians are more than willing to ignore the disadvantages of the small display in favor of working with their own well-known handheld devices on the go.

3.7 Conclusion and Future Work

In this chapter, we presented the mobile visualization environment IV3Dm and evaluated its deployment to support the selection of parameters for DBS therapy of PD patients. We anticipate that the system could provide a significant step forward in clinical practice for several reasons: mobile devices have generated significant interest in the clinical community, and physicians already widely use these devices in their daily life; computational models are gaining acceptance by practitioners and are being used more often for clinical decision making; IV3Dm has

a simple, intuitive interface, flexible means to access data sets, and can be used at the patient bedside.

This study is retrospective, and we did not test the stimulation settings chosen using IV3Dm in each patient. Consequently, it is possible that the chosen settings are better than, equal to, or worse than the settings chosen via standard of care. Therefore, one important area of future work is to assess whether use of the system improves patient outcomes. A follow-up study to that effect is already ongoing at the time of writing this thesis. The study will span multiple years and builds on the DBS decision support system presented here. The goal is to prospectively measure the effectiveness of the approach. For this purpose, the system will be deployed to treat PD patients in an established PD clinic as well as in a home environment. Outcomes will be tested for non-inferiority to standard of care. The hypothesis is that the use of the mobile support system will enable substantial time savings to manage the patients and reduce the burden for patients, family caregivers, clinicians, and nurses.

In a future study we will examine whether the inclusion of evidence gathered from other patients results in further improvements in the selection of DBS parameters. Previous work has begun to develop methods to define optimal stimulation targets from multi-patient studies [BCH+11, MBW+09]. We will extend the visualization with these target locations.

While previous attempts have been made to provide interactive visualization of patient-specific DBS models, these systems require significant amounts of training and domain knowledge to become proficient. An advantage of IV3Dm is the minimal amount of expertise required and its attractive features for the clinical workflow. We predict that the approach could have significant impact not only in DBS for PD but also in other neuromodulation methods where patient-specific models could provide useful insights into the best way to prescribe the therapy.

To further improve the ease-of-access, a future generation of our system could incorporate a portable browser-based interface that users can access from a range of mobile and stationary devices without requiring a platform-specific software installation. Chapter 5 and 6 establish the browser as a viable platform for graphics applications.

The current system is in a prototype state and requires further testing before introducing it into a clinical environment. This is especially true for the data distribution subsystem when considering protected health information and security aspects.

Chapter 4

Hybrid Rendering of Multi-Resolution Data Sets in Dynamic Environments

4.1 Introduction

Today, simulations and measurements regularly generate large scientific data sets. Often these data sets can only be understood by means of visual analysis. However, the interactive visualization of high-resolution data sets requires significant computing and storage resources. These resources may not be available on all devices that shall display the data. Mobile devices may especially lack the capabilities to *fully* store and process large or even medium-sized data sets. Another more extreme case is an exascale scenario [exa10], where simulations run on highly parallel, dedicated supercomputing architectures. As it is predicted that processing power will increase more rapidly than storage capacity and I/O bandwidth in such systems [exa11], it may not even be feasible to permanently store exascale simulation results. The supercomputer may therefore generate and visualize data sets *in-situ*.

A solution to provide visualization to devices with limited capability is to outsource processing. A rendering server or a cluster of servers processes part of or the whole visualization pipeline and then transmits the results to the client for further processing. The approach reduces client-side requirements, which are minimal if the client is used for display only. Bethel et. al. [BCH12] and Luke et. al. [LH02] classify how the visualization pipeline can be distributed across server and client machines. However, network bandwidth, latency, and reliability can affect the user experience. The consideration is especially relevant for best-effort networks like the Internet or wireless connections. Further, scalability on the server becomes an issue if multiple clients request rendering services simultaneously. The user experience suffers if an

overloaded server cannot complete requests in a reasonable time. To increase server scalability, considerable investments in hardware may be necessary. This particularly applies to distributed architectures that employ multiple servers to power the visualization and that we address in Chapter 6 and 7 and also in Section 2.4.4.

In this chapter, the client connects to a single server. We refer to remote rendering if the client is used for display only.

While the server component is necessary to provide the resources for large-scale visualization, a purely server-based approach may leave a considerable amount of client hardware idle. Nowadays, even mobile devices often have capabilities to support interactive visualization to a certain degree; and using them for such tasks becomes increasingly popular as we demonstrated in Chapter 3.

Hybrid rendering techniques utilize both server- and client-side resources. When the server initially holds the data, the three core aspects are: Splitting the visualization process into units of workload, assigning workload to client and server, and transferring the data required for client-side processing. Performing work on the client reduces server load and may also reduce network load compared to remote rendering (for example there may be no need to constantly transfer image data; see Section 4.2 for references). Further, the cooperation of server and client can result in faster image generation. The increased scalability by using each client hardware is available without investment from the server provider.

We present a hybrid rendering method that adaptively adjusts what rendering workload needs to be done and where it is done based on server, client, and network conditions. We identify workload in terms of quality levels (QL) of a data set. Displaying higher QLs progressively refines the view. Any renderer adhering to the QL concept can plug into our system. The scheduler selects QLs for rendering on server and client. The goal is to provide the client with the next QL as soon as possible. The client supports a subset of the QLs depending on its capabilities. For the implementation, we utilize the rendering library introduced in Section 2.4.4, which is designed for multi-resolution data sets.

Our approach uses a probabilistic scheduling model. The scheduler acquires probability distributions for rendering and transfer timings at run-time for each QL to determine what QLs are to be rendered on each side. This method allows to account for the uncertain conditions that affect the performance. A client with arbitrary capabilities may connect to a server via a network link with arbitrary characteristics. Load on server and network may vary depending on how many clients are active and what outside traffic is on the link. We make no assumptions about these conditions, which are subject to change. Additional factors like other applications that run in parallel, background tasks performed by the operating system, hardware character-

istics, and rendering system events (for example allocating memory) may result in fluctuating timing observations. To account for the dynamic variables, our method updates and compares timings in terms of a probability distribution to adapt the schedule.

The remainder of the chapter is structured as follows: First, we present related work in the area of hybrid rendering and scheduling under uncertainty. The core part then describes our hybrid rendering method. We address how the system acquires timing distributions, uses the distributions to schedule QLs, and adapts to changing conditions. The results section presents different test scenarios and comparisons. Further, we underline the suitability to use probability distributions. The chapter concludes with a discussion and finally future directions.

4.2 Related Work

4.2.1 Hybrid Rendering

We divide hybrid rendering methods into three categories as illustrated in Figure 4.1. First, server and client cooperatively render every image. Second, the client renders a number of images independently after an initial input from the server. From time to time, the server provides additional input to uphold the rendering on the client. Third, server and client produce individual images independently.

Falling into the first category, Aranha et. al. [ADD+07] distribute ray-tracing workload. They use a cost function to decide on the number of pixels to be rendered on each side.

Several techniques have been developed for volume rendering. To progressively render unstructured, tetrahedral grids, Callahan et. al. [CBPS06] utilize the client as the rendering unit while the server stores the data and pre-processes the geometry. Prohaska et. al. [PHKH04] use a hierarchical volume renderer on the client for CT-scan exploration. The client accesses data blocks remotely to progressively refine the view. The system supports a user-defined region-of-interest (ROI) for which the highest resolution is chosen.

Diepstraten et. al. [DGE04] describe a server-centered approach for line rendering on mobile devices. The server extracts 3D lines and projects them to image space. The client then renders a package of 2D lines received from the server. Okamoto et. al. [OOI11] propose a system where the server stores geometric data and a repository of pre-rendered images. When the client requests a view, the server sends a selection of images most closely matching the view along with a coarse version of the mesh. The client then reconstructs the view.

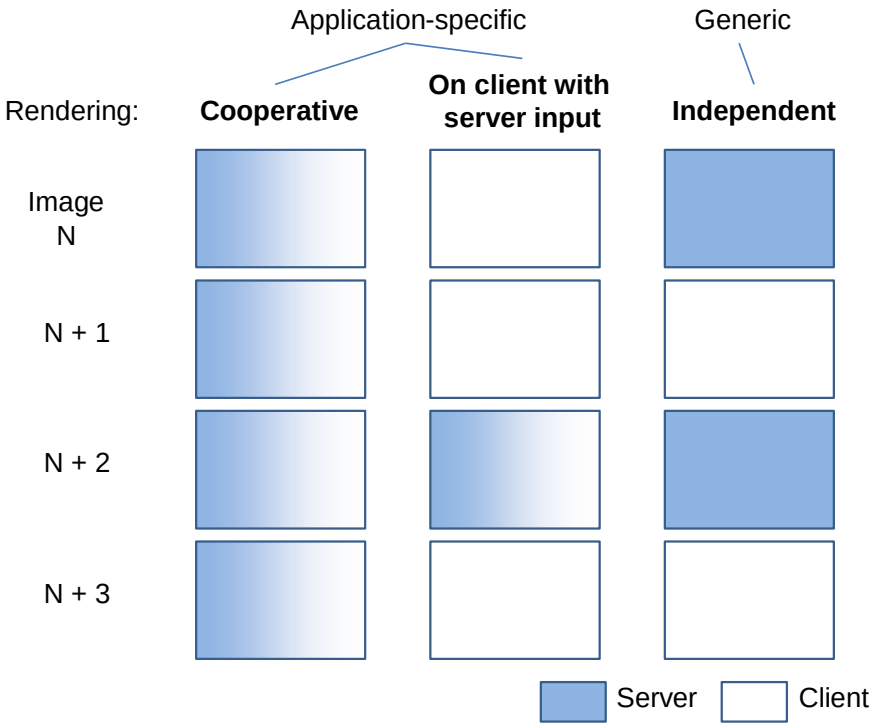


Figure 4.1: Classification of hybrid rendering approaches.

Noguera et al. [NSOJA11, NSOJA13] present a method for hybrid terrain rendering. The client receives geometry for terrain close to the viewer for local rendering. For the far-away parts, the server renders panoramic images that the client merges with the local rendering result.

Falling into the second category, Engel et. al. [EWE99] discuss distributed isosurface reconstruction. Their approach employs a hierarchical level-of-detail (LOD) concept. The user can define a ROI, which the system reconstructs at the highest quality. After the reconstruction completes, local interaction on the client is possible until the isosurface changes. Engel et. al. [EOEI00] outline similar approaches for the visualization of multi-dimensional chemical data sets.

Li et. al. [LSK11] render an image on the server and extract a mesh from the depth buffer. The server textures the mesh with the color buffer and sends it to the client for rendering. Should the error due to view changes get too high, the client requests a new mesh from the server. Luke et. al. [LH02] outline a similar approach. Visapult [BTl+00] produces images using a parallel volume renderer on the server and then interactively constructs new views from the images on the client. Client-side rendering is decoupled from receiving image updates.

Shi et al. [SNC12] present a method that allows the client to synthesize a range of views based on a set of server-generated depth and color reference images. However, fast and arbitrary user interaction causes the frequent transfer of new reference sets and thus limits the approach.

Doellner et al. [DHK12] use the server to generate G-Buffer cube maps for views requested by the client. The client uses the latest received G-Buffer to reconstruct the rendering while the server asynchronously produces more up-to-date maps.

The techniques outlined above require an application-specific algorithm to distribute the rendering workload. Further, a persistent connection to the server is required. High latency reduces the interactivity. Connection loss terminates the visualization on the client immediately or when new input from the server is required.

Our approach falls into the third category: generic hybrid rendering. Engel et. al. [EEH+00] show a medical application using two different volume renderers. The client performs low-quality rendering while the server provides higher quality on demand. Dyken et. al. [DLS+12] describe an application that supports a browser-based client. The client interactively draws an illustrative, coarse representation of a geometric data set with WebGL. The server processes the detailed original model and transfers rendering results to the client as images. The framework presented by Schinko et al. [SBEF14] also targets the browser as the client platform. The client interactively renders low-resolution proxy geometry with X3DOM [BEJZ09], and the server back-end provides higher-quality renderings with additional material properties for static views.

Generic hybrid rendering allows to maintain interactivity on a capable client should the connection suffer from latency or limited bandwidth or the server be occupied or unavailable. In addition, the generic approach allows to support applications independent of the rendering algorithm.

4.2.2 Scheduling under Uncertainty

In various scheduling problems, uncertainty is a significant factor. The time required to complete a task may be difficult to estimate before actual observations. Resources may be unreliable, occupied, or even become unavailable. Probabilistic approaches have been employed to deal with such scenarios. Instead of absolute values, probability distributions represent the state of the system. Related methods find application in the areas of project management [HA89], maintenance and production [VP01, BBQL16], and process planning [IL09]. Handling uncertainty also plays a role in domains that are not directly linked to scheduling like clinical decision making [BIHGO16].

Ierapetritou and Li [LI08, IL09] distinguish two scheduling approaches and give several references for each. First, a preventive scheduling system models the behavior of uncertain factors based on historical data and statistics gathered previously. Thus, the scheduler either knows or can estimate the characteristics of the probability density function (PDF) to determine

the schedule in a pre-process. Janak et. al. [JLF07] as well as Balasubramanian and Grossmann [BG01] describe examples.

In contrast, in a reactive scheduling system, not enough information about the uncertain factors is available before the tasks are performed. Therefore, the schedule constantly adapts in response to changing conditions.

Our method can be classified as reactive. The timings for QL rendering and transfer are subject to fluctuation due to several dynamic factors (see Section 4.4.2), which the scheduler cannot know *a priori* for any client-server connection. Therefore, the system obtains timing distributions at run-time to determine the schedule. However, our approach adds additional complexity because we consider to skip certain QLs to keep load from the system and allow the rendering of QLs to be aborted to maintain responsiveness. Consequently, the scheduler is not guaranteed to obtain timings for every QL in frequent intervals.

Our approach is also related to resource-aware scheduling on an abstract level. Resource-aware scheduling partitions data-parallel problems into pieces of load and then distributes the pieces for processing on possibly heterogeneous computing resources. Similar to our hybrid rendering, knowledge about the availability, capability, and performance of the resources is essential to balance the load.

Viswanathan et. al. [VVR07] implement a resource-aware system based on the divisible load theory [BGR03]. They use cluster nodes that are connected in a local area network (LAN). Source nodes generate load at run-time, and sink nodes perform the processing. A control node coordinates the distribution of load based on the sinks' estimated memory and processing capabilities as well as the sources' load size and deadline requirements, which are not real-time. The goal is a maximum utilization and throughput and a maximum acceptance rate of new load. The algorithm runs iteratively until all load has been admitted and processed. Unlike our system, the approach regards network overhead as negligible and requires to pass a substantial amount of control information between the nodes.

Teresco et. al. [TFF05] schedule load for large simulations in a heterogeneous computing environment. This includes possibly non-dedicated nodes that process additional tasks apart from the target application, which is similar to a scenario in our setup: a loaded server that processes rendering tasks for multiple clients simultaneously. Teresco's scheduler discretizes the problem domain for cooperative processing using a mesh partitioning algorithm. The system obtains performance characteristics of computing and network components to guide the partitioning. Run-time monitoring allows to adapt the partitioning. Developers have to write their applications within a specific architecture to plug them into the scheduling system.

4.3 Scheduling Fundamentals

First, we describe the fundamentals required for the hybrid rendering method. The goal of the scheduler is to provide the client with a new QL for display as fast as possible. We therefore start with the definition of quality levels. Next, we describe the requirements on the rendering system, which includes addressing limitations of the QL concept. Last, we outline the transfer of the data required to render QLs from server to client to enable client-side rendering.

4.3.1 Quality Levels

Our method builds on the concept of *quality levels*, QLs. A QL is a representation of a data set that client or server can render independently to produce an image. We require a total ordering of the QLs, with detail increasing with the level number. The client requests QLs consecutively within a frame to progressively refine the view. When we use the term "frame" in the context of QLs, it includes all QLs that are to be displayed for a view. We often refer to either server or client instance of a QL when using the term "QL".

The underlying rendering system provides the classification of a data set into a set of totally ordered QLs. The classification is thus ultimately up to the application developer who integrates the renderer into our framework. Our hybrid scheduling layer on top of the rendering system must not know the format of the QL data, including possible compression and the rendering algorithm to produce images from the data.

We assume that the server can render the highest QL. In contrast, a resource limited client may not support some QLs. There may be QLs tailored for a specific renderer. For example, a different or feature-reduced renderer may be used on mobile devices.

4.3.2 Underlying Rendering System

Our approach supports any renderer that can map its data to QLs, for example using a multi-resolution hierarchy, different sampling rates, or some other unforeseen representation. The approach is therefore applicable to any visualization system that already uses a multi-resolution data set representation.

We acknowledge the mapping to QLs may not be straightforward. This especially applies to view-dependent rendering systems where the LOD may vary over regions of the data set. Also, to integrate renderers that offer a continuous refinement, it is required to discretize the

continuous representation to distribute QLs for scheduling. Here it is feasible to create a QL hierarchy that is not too fine considering images for QLs rendered server-side must be sent over the network. Finally, we will investigate how to adapt or extend the QL concept to support renderers that generate an additive refinement where an iteration adds to the previous image instead of replacing it. The ray-tracers presented in Chapter 7 are examples for such renderers.

We expect the renderer to be interactive. The renderer should provide at least one QL per data set that can be completed at interactive frame rates. The renderer should also be interruptible to maintain responsiveness.

We allow the renderer to reject QLs above a certain level. For example, the renderer may use the screen space size of the visualization to limit the rendering of higher QLs to avoid processing unnoticeable details and consequently save resources.

We have integrated two rendering systems so far. The first is the *Tuvok* volume rendering library [FK10]. *Tuvok* provides a hierarchical renderer that divides a data set into LODs. These LODs can be independently stored and rendered. We can map the LODs directly to our QLs. Moreover, to underline the flexibility of the concept, in Section 4.5.3 we divide each LOD into multiple QLs to improve interactive rendering and enable a more fine-grained view refinement . The second system is a proprietary geometry renderer based on progressive meshes [Hop96]. Figure 4.2 and 4.3 show example data sets for both renderers.

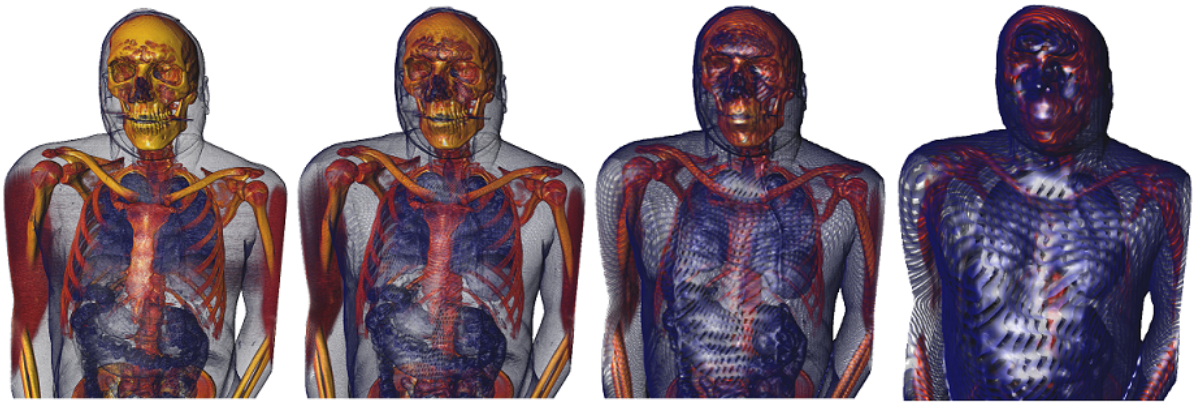


Figure 4.2: QLs 4 to 1 of the male *Visible Human* volume data set [U.S12] rendered with *Tuvok*.

4.3.3 Interleaved QL Data Transfer

We make no assumptions as to the availability of QL data on the client. Initially, the server solely stores the data and performs the rendering. The server encodes images as JPEG and sends them to the client via a TCP connection.

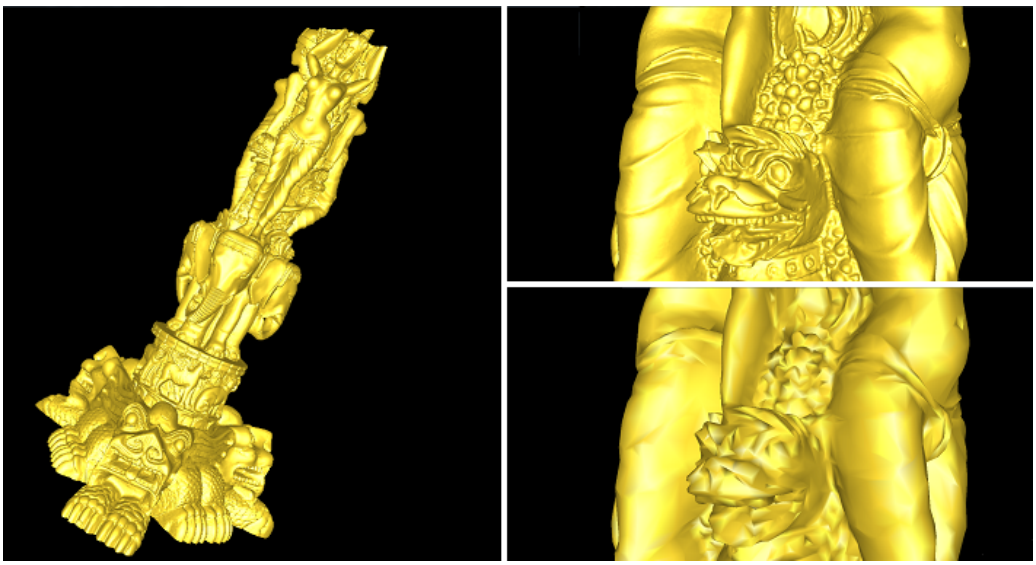


Figure 4.3: Rendering of the *Thai Statue* mesh data set [Sta11] with six QLs. The right side shows QL6 (top) and QL1.

To enable the ability to switch where QLs are rendered, the client needs to obtain the QL data for local rendering. Figure 4.4 outlines the process. First, the system determines the QLs the client supports in a handshake phase. The client queries the server-side renderer for QL specifications of a data set. This meta-data contains application-specific information such as the QL data size and required renderer features (like 3D textures). Using the specification, the client-side rendering system determines whether a QL can be supported.

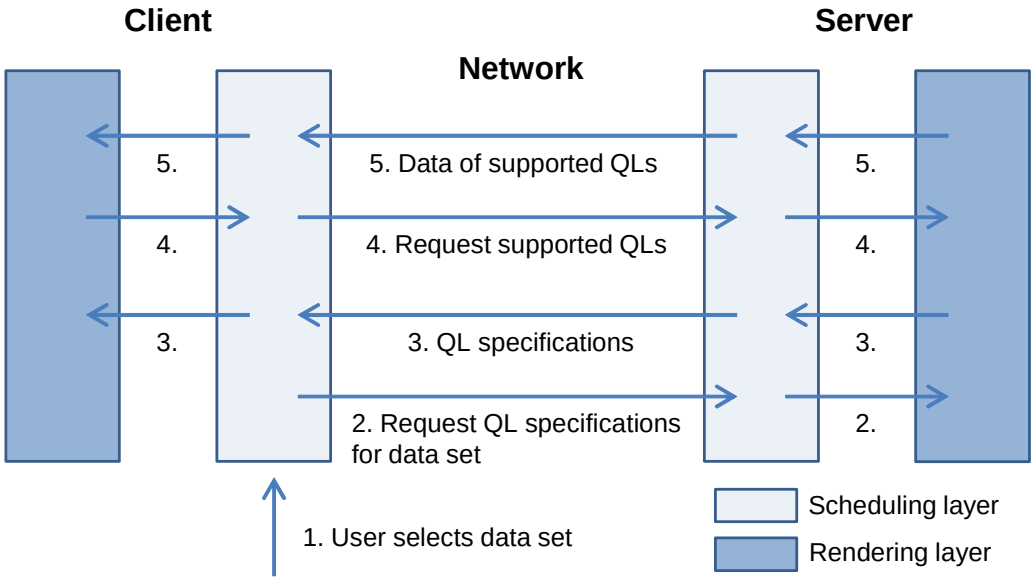


Figure 4.4: Determining the QLs a client supports during the handshake phase, and transferring the corresponding data to enable rendering on the client.

The client may not support a QL due to limited memory and processing power. Required renderer features may not be present. The client may also deliberately disapprove a QL to save

memory, network bandwidth, or battery time. It may not be feasible to transfer large QL data, for example to a mobile device over a wireless link. Further, a small display resolution could make it infeasible to even consider a high-resolution QL on either side. Finally, the QL data may be confidential and should not leave the server.

After the handshake, the server sends the supported QL data to the client using the *Deflate* compression algorithm. The server uses time division multiplexing to interleave QL and image data. Since the prompt display of images has the highest priority, the server inserts chunks of QL data in phases where the image transfer is idle. There are two suitable scenarios. First, if user interaction stops, the server renders QLs of increasing number to progressively refine the view. Rendering a high QL can take time during which the client waits for the next image. Second, after the client received the final QL, both sides are idle waiting for a new interaction event. The server increases the QL transfer bit rate over idle time but does not exceed a maximum to guarantee seamless interruption should an image transfer come up.

4.4 Scheduling Quality Levels

The scheduler runs on the client. For simplicity of the explanation, we assume in this section that the client has all supported QLs available. Figure 4.5 gives an architecture overview to set the stage for the following detailed description: First, we describe the timing distributions the client holds to base the scheduling decision on and the fluctuation expected to occur in the timings. Next, we present the details of the scheduling algorithm and how it connects to the rendering process. We then describe how the scheduler initializes the distributions and updates them with timings at run-time. Finally, we discuss how the system obtains timings.

4.4.1 Timing Distributions

For each QL on server and client, the scheduler maintains a continuous processing and transfer time *normal distribution* (ND) with millisecond accuracy. A discussion on why we choose the ND as the distribution type follows in Section 4.6. The scheduling algorithm uses the NDs to decide which side should render which QLs. The scheduler builds NDs by accumulating timings. Processing time includes the rendering time and in case of the server also the image encoding time. Transfer time only applies to server QLs. It includes the time to request a QL for rendering and to send the rendered image to the client in return.

We simply use the term “random sampling” to describe generating random variates distributed according to a ND. We simply refer to a variate as a sample. Press et al. [PTVF07] describe

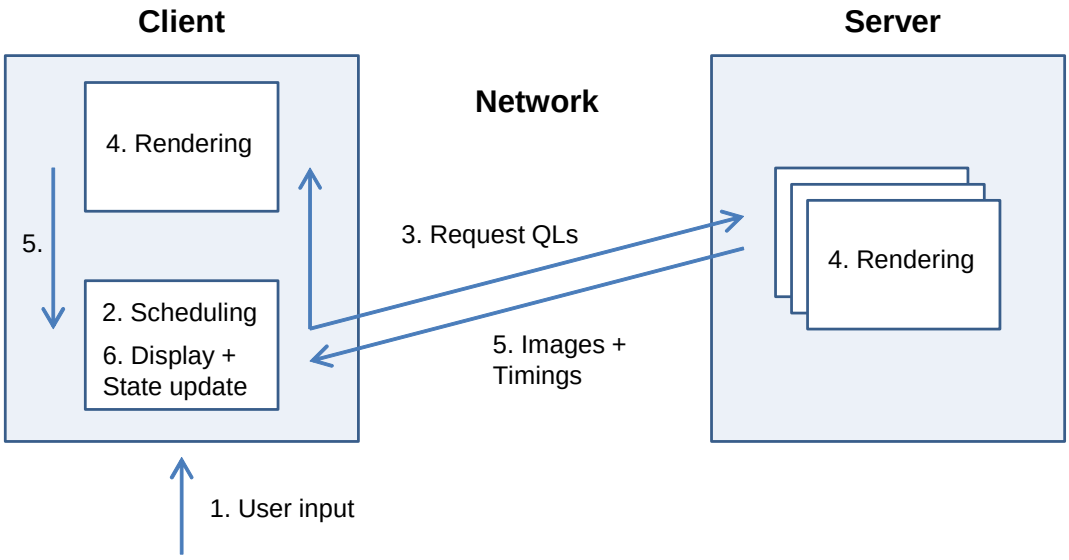


Figure 4.5: Illustration of a frame's rendering. Scheduling runs in the main display thread concurrently to the local rendering thread. The server handles each client in a separate rendering thread. Once user input comes in, the scheduling algorithm determines the QLs to be rendered. The display thread requests the rendering of the server and client QLs. Consequently, generated images and corresponding timings gradually arrive. The scheduler updates its state with the timings. The display thread refines the view with incoming QLs. Should the user issue new input before view refinement concludes, the client can prompt to interrupt the rendering to immediately start the next frame.

several sampling methods. Our system enforces a lower bound for time NDs by falling back to the mean should a sample $<= 0$ occur.

The scheduler bases its decisions on probability distributions due to the uncertain conditions that affect the timings.

4.4.2 Timing Fluctuation

Fluctuation in the timings can occur due to several uncertain conditions, which we classify in the following. The conditions may not only change between rendering sessions but also within a session.

First, server load affects processing timings for server QLs. We make no assumptions about the number of clients that connect to a server, and the load may thus be arbitrary. Network load affects transfer timings. Since we make no assumptions about the link between client and server, bandwidth, latency, and outside traffic are arbitrary. Transfer and processing timings move along with changes in these conditions.

Second, rendering system parameters affect processing timings on both server and client. There

are the view changes but also other parameters like the display resolution or a volume rendering transfer function.

Third, processing time fluctuation can occur due to background factors such as operating system activity and hardware characteristics, even when server load and rendering system parameters are stable. Task scheduling and memory performance variation are persistent examples. There may be temporary factors like a copy or update procedure or a throttling of the GPU due to overheating. The user may perform other tasks alongside the rendering session. Our generic method cannot model the range of external factors, which affect server and client timings independently.

Finally, fluctuation may occur due to outliers. An outlier is a timing that is not representative for subsequent timings under the same conditions. Outliers are difficult to predict, but we can make assumptions about when they are likely to occur for the processing time. Rendering system specific initialization like the allocation of resources and the creation of acceleration structures occurs particularly on the first rendering of each QL. Also, hardware-specific warm-up effects are possible, especially for a GPU-heavy renderer (for example setting up GPU state).

To mitigate the effect of outliers our approach utilizes a weighting system when adding timings to a ND (Section 4.4.5). Section 4.4.4 describes parameters that allow to account for initialization and warm-up related outliers.

4.4.3 Scheduling Algorithm and Rendering Process

This section presents the probabilistic algorithm that allows the scheduler to adapt to changing conditions. We assume there are a number of QLs on server and client and that the corresponding NDs are available. The following sections describe how the scheduler obtains NDs and timings at run-time.

Given a frame to render, the goal of the scheduling algorithm is to update the view with a newly completed QL as fast as possible on the way to reach the highest QL. Completing a QL means its rendering result is available at the client for display.

To set the stage for the following algorithm description, we give a summary first. The scheduler executes once every frame and determines the list of QLs that the client requests for rendering in the frame (as illustrated in Figure 4.5). The scheduler uses the timing information contained in the NDs for its decision. This information tells when a QL is expected to complete. The

scheduler takes into account that client and server can operate in parallel but each side individually works sequentially. Thus, scheduling a QL on one side affects the expected completion time of other QLs on that side. The goal is to minimize the time until the client displays a new QL. The scheduler thus always chooses the QL that is expected to complete earliest next. The exception is the start of the frame, where the scheduler may also choose the highest QL that still adheres to a desired frame rate. As rendering and network transfer are expensive, the scheduler skips QLs lower or equal to the chosen QL, even though no timings can then be obtained for the skipped QLs.

We describe two versions of the algorithm. The pre-sampling strategy reflects the variance in the NDs perfectly, but undesirable slow downs can occur as a side effect. Therefore, we provide the alternative distribution-comparison strategy to mitigate the side effect.

4.4.3.1 Pre-Sampling Strategy

Let list A be a list containing all QLs and their processing and transfer time NDs. First, the scheduler takes a random sample from the processing and the transfer time ND for each QL in A. The accumulated sample time is the sum of processing and transfer time sample. The scheduler stores the processing and accumulated sample of each QL in a new list B and sorts B by the accumulated time in ascending order. Should two QLs match in number and time, the scheduler prioritizes the client QL to keep load off server and network.

The scheduler optionally allows to set up a desired frame rate. The scheduler looks for the highest QL in B that fulfills the frame rate and removes lower or equal QLs from the list. Rendering these QLs would cause unnecessary load as a higher QL already meets the application's requirement. By default, the scheduler considers all QLs in B for rendering.

The main execution loop of the algorithm now processes B. The scheduler generates no further samples from the NDs within the loop, which is why the strategy proceeds deterministically from here on and is called pre-sampling. The scheduler adds the first QL q in B to the end of a list C. C is the output of the algorithm and contains the QLs to render in the frame. QLs lower or equal q are then removed from B. If B is not empty, the scheduler adds the processing time sample of q to the accumulated sample of every QL on the same rendering side as q. The renderer on that side cannot process the next QL before finishing q. The transfer time does not delay the rendering of the next QL and is thus not added. The scheduler sorts B by accumulated time again and iterates until B is empty.

The first QL in C is the interactive QL as it is expected to complete first and thus should provide interactive performance. The remaining QLs in C are the non-interactive QLs. The

scheduler finally splits C into client and server QLs, which the client then requests for rendering. Server and client can process their list of QLs independently and thus render in parallel.

The client uses a small time interval between completing the interactive QL and requesting the non-interactive QLs. This request timer prevents requesting non-interactive QLs prematurely even though the user is still active, which would result in permanent rendering aborts during interaction phases. The overhead to abort the renderer can be noticeable, which depends on how quickly the renderer can exit. In contrast, if a new interaction event arrives within the interval, the client can immediately start the next frame.

Each time the scheduling algorithm runs, different random samples could be chosen to create list B, causing a different scheduling outcome in list C. The order of QLs in B is uncertain and reflects the variance in the NDs. The PDF of a ND approaches zero but never reaches it. A sample can therefore take on any value and any order of QLs is possible. The probability of a specific ordering is arbitrary and may only exist theoretically; in practice, zero probabilities are possible due to limited computer precision.

The probabilistic approach allows the system to adapt to the uncertain and changing conditions. The scheduler will eventually select every QL for rendering. Gathering timings for completed QLs allows the scheduler to notice change. However, the adaptation process is not guaranteed to be immediate. First, how fast a ND shifts in response to new conditions is dependent on the weighting system to adding timings (Section 4.4.5). Second, the scheduling frequency for a QL may be irregular. The probability of a QL to be scheduled for rendering in a frame is the sum of probabilities of the scheduling outcomes that contain the QL. This sum may be very small. If the system rarely renders a QL, for example on an occupied server, it can be slow to notice change in the QL's performance behavior.

Table 4.1 illustrates the possible outcomes of a scheduling scenario. While the chance for the scheduling of QL2 on the client is seemingly low ($\sim$0.52%), the scheduler still produces the case every few seconds in practice considering that this is an interactive environment and the scheduler re-evaluates every frame.

The pre-sampling strategy produces a schedule that takes into account the variance in the observed timings. However, there is an undesirable side effect as the scheduler potentially chooses any QL as the interactive QL. If a QL is chosen that does not fulfill the interactivity requirement, the user may notice a slow down. To uphold responsiveness, the system allows to interrupt the rendering of the interactive QL if the QL does not complete within a target FPS. This is optional since it causes another problem. If no QL can meet the target FPS reliably, the system suffers from constant aborting and gaps in updating the display during interaction.

Table 4.1: Demonstrating the variability of a scheduling scenario. a) Given a fictional state (*mean*, *variance*), we have run the pre-sampling strategy one million times. b) The scheduler produced multiple outcomes (client QLs in *orange*). We derived the probability for each outcome statistically from the runs. For simplification, we used the same transfer time distribution for the server QLs.

Processing time distributions		
	Client	Server
QL1	30 9	20 35
QL2	200 139	105 870
QL3	1100 2227	1000 7828

Transfer time distributions	
QL1-3	15 9

(a)

Scheduling outcomes	
1-2-3	40.3652%
1-2-3	34.9583%
1-2-3	14.9797%
1-2-3	9.084%
1-2-3	0.4896%
2-3	0.0738%
1-2-3	0.0244%
2-3	0.0226%
1-2-3	0.0024%

(b)

4.4.3.2 Distribution-Comparison Strategy

To mitigate the side effect of the pre-sampling strategy, the scheduler provides an alternative strategy: distribution-comparison. The basic structure of the algorithm stays the same. The accumulated time ND is the sum of the independent processing and transfer time ND. The scheduler stores the accumulated time ND of each QL in list B and sorts B by the distribution mean.

The scheduler incorporates the optional desired frame rate by searching B for the highest QL i that can maintain the frame rate with at least a minimum probability. QLs lower or equal to i are removed from B.

The main execution loop then produces C. The scheduler compares the accumulated NDs of the first two QLs in B by taking a random sample from each ND. The QL q with the smaller sample goes into C. If only one QL is left, it is chosen without competition. The scheduler removes QLs lower or equal q from B. If B is not empty, the scheduler adds the processing time ND of q to the accumulated ND of all QLs on the same render side, sorts B again, and proceeds with the next iteration.

Since B contains distributions sorted by the mean, QLs with higher completion time, which are more likely to disrupt an interactive frame rate, are always further up in the list. Random sampling cannot cause these QLs to occasionally end up as the interactive QL anymore.

While the distribution-comparison strategy does not account for all the possible scheduling outcomes like the pre-sampling strategy, it can still provide a smoother user experience. Both strategies are viable, and we recommend an application-specific selection. The system does not automatically switch between the two. Though, we consider to investigate run-time detection of the pre-sampling strategy side effect to automatically decide whether to fall back to distribution-comparison.

4.4.4 Distribution Initialization and Auto-Scheduling

When a rendering session starts, our system has not gathered any timings yet. Before the scheduler includes a QL in the scheduling algorithm, the system has to acquire timings to initialize mean and variance of the QL's NDs. Until this state is reached, the scheduler automatically adds the QL for rendering to facilitate the timing acquisition. We call this process auto-scheduling.

The scheduler provides an option to ignore the first N processing timings for a QL. In addition, the scheduler may ignore all timings for a rendering side until a minimum of processing time occurred on that side. These parameters allow to account for initialization and warm-up related outliers. We assume an initial rendering of a QL to be a likely outlier, and we observed this behavior for *Tuvok*, which is an out-of-core system that needs to page in data from disk. The first accepted timing initializes a ND and is the initial mean.

The scheduler also needs to determine the initial variance. There is limited information on-hand with the first accepted timing, which could still be an outlier. Therefore, the scheduler artificially adds variance to a ND newly initialized from a single timing. Starting with uncertainty in the schedule benefits the acquisition of timings across the QLs to gradually narrow down the NDs and the schedule according to the actual conditions.

The scheduler sets the initial variance of a ND to the maximum valid variance. We define the largest interval that random samples taken from the ND can fall into as $[0, 2 \cdot \text{mean}]$. The goal is to set the initial variance to the maximum value that still reflects this requirement. To determine the value, we solve the cumulative distribution function (CDF) for deviation using $1 - \epsilon$ as the target probability for a very small ϵ (the CDF never becomes one, thus the ϵ):

$$0.5 \cdot \left(1 + \operatorname{erf}\left(\frac{h}{d \cdot \sqrt{2}}\right)\right) = p + \frac{1-p}{2}$$

$$v = d^2 = \left(\frac{0.707107 \cdot h}{\text{erfinv}\,(p)} \right)^2$$

where $erfinv$ is the inverse error function, h the half length of the interval, p the target probability, d the deviation, and v the variance.

4.4.5 Distribution Update and Reset

A newly initialized ND is a sparse representation of the current conditions as only one timing has been acquired so far. The scheduler updates a ND with further timings as they become available. The scheduler provides an option to accumulate more timings before considering a ND meaningful and including it in the scheduling algorithm.

The system passes timings for completed QLs to the scheduler. However, a QL is not guaranteed to complete as the client may issue the abort of the rendering. In this case, estimated timings may still be available as described in Section 4.4.7. Since rendering is expensive, the scheduler skips QLs that are expected to complete later than a higher or equal QL. Concluding, the scheduler is not guaranteed to obtain timings for each QL in frequent intervals.

The scheduler adds timings to a ND using the following time-based weighting system. It is not necessary to retain the timings as the scheduler updates mean and variance incrementally [Fin09]. The weighting method should have two characteristics. First, it should be resistant to outliers. Second, newer timings should have more significance. Our approach takes the time since the last timing into account to determine the weight for a new timing. For each QL, the scheduler allows to define a weighting function that takes the elapsed time in milliseconds as a parameter. The function returns the factor of weight difference between the last and the current timing. The new weight w is calculated as:

$$w = lw \cdot \text{wf}(et)$$

where lw is the weight of the last timing, wf the weighting function, and et the elapsed time since the last timing.

The scheduler uses a function that increases weights linearly by default, but an exponential function or another approach can be used at will. There is a tradeoff between adapting to changing conditions quickly and being outlier resistant.

The more time passes since the last update of a ND, the less representative the ND becomes. Conditions might be different than before, and thus timings loose their significance over time. In terms of the weighting function, the larger the weight of a new timing is, the less significant

the previous timings are. The scheduler allows to set up the maximum weight of a timing. If a timing would reach this weight, the scheduler resets the corresponding ND to the uninitialized state to facilitate the acquisition of up-to-date timings. Previous timings have lost their significance, and we must assume that start-up outliers may happen again.

4.4.6 Obtaining Processing and Transfer Timings

This section describes how the system obtains the timings to generate the NDs. We distinguish between measured timings, which are available if a QL completes, and estimated timings, which may be available after the abort of the rendering. There are two types of timings: processing and transfer.

For the processing time measurement, the rendering time and in case of the server also the image encoding time counts.

For the transfer time measurement, the client starts a timer before requesting the rendering of the server QLs. The client probes the timer when the response for a QL arrives to determine the waiting time. The client immediately restarts the timer for the next QL. The response includes the processing time and for completed QLs also the image data for display. The client determines the transfer time t as follows:

$$t = pt + w - p \tag{4.1}$$

where pt is the transfer time of the previous QL that completed within the request. For the first requested QL, pt is zero. w is the waiting and p the processing time.

Equation 4.1 must include pt for two reasons. First, after having sent the image for a rendered QL, the server immediately proceeds with the next QL. Consequently, rendering continues during the transfer of previously generated images. Second, the latency to send a rendering request from client to server is only included in the measurement for the first QL, but the scheduler must consider the latency for subsequent QLs as well.

4.4.7 Rendering Abort and Timing Estimation

Here we describe the unfavorable effects of incomplete and lost timings caused by rendering abort and how our system can compensate the loss by estimating both processing and transfer time.

Our system allows to abort the rendering to stay responsive. Progressive view refinement terminates if the next frame is about to start due to new user input. However, interrupting the rendering prevents the system from completing the time measurements. There is only a partial measurement for the QL aborted while rendering and no measurement for consecutive QLs that are still scheduled for processing. Missing timings can cause the NDs to stay or become unrepresentative, which prevents the scheduler from adapting. The scheduler could then keep up a warped schedule that does not reflect the current conditions. To mitigate these effects, timing estimation is in place.

4.4.7.1 Processing Time Estimation

To enable the estimation of rendering time, we introduce the concept of work units. For each QL, the rendering system defines a number of work units. QL N has less units than QL $N+1$. We expect each unit to require about the same rendering time under stable conditions. The mapping of actual rendering work to linear work units is up to the rendering system.

Such a mapping may only approximately be possible. For *Tuvok*, we use a direct mapping to the number of bricks. However, there may be significant differences in the rendering time of bricks. While estimations may be off, they at least likely push a ND in the right direction. The advantage of keeping the state up-to-date outweighs the possible inaccuracy. The outlier-resistant weighting system is in place to absorb the impact of largely inaccurate estimations.

When rendering the list of QLs requested for a frame, each rendering side tracks the average work unit completion time. The server also tracks the average encoding time. After an abort, the estimated remaining processing time ep for a QL is calculated as follows:

$$ep = w \cdot wc + e$$

where w is the number of work units still to complete, wc the average work unit completion time, and e the average encoding time.

Since uncertain conditions such as server load affect the rendering and encoding time, the estimation relies on information obtained within the bounds of a frame. Only if the abort hits so early that no measurements for the current frame are available, the estimation falls back on the most recent data from previous frames.

4.4.7.2 Transfer Time Estimation

Transfer time is dependent on latency, bandwidth, and the size of the image.

For each aborted QL, the server estimates the encoded image size based on the last image that was encoded. Since the image size is view-dependent, it is reasonable to base the estimation on the most recent sample that was probably generated within the same frame. But the size may also differ between QLs. Therefore, we consider an extension that keeps information from previous frames persistent. The server could employ a view-matching metric to decide whether a previous image size for the same QL is still representative for the current view. The approach may be feasible as view changes between adjacent frames are likely small in the interactive context.

The client tracks the image transfer rate. Further, the client uses a monitor to approximate the round-trip time. The round-trip time is independent of the image size and thus excluded from the transfer rate measurement.

The client estimates the transfer time et as follows:

$$et = \frac{ei}{tr} + rt$$

where ei is the estimated image size, tr the transfer rate, and rt the round-trip time.

4.5 Results

We tested the hybrid rendering in a number of scenarios to demonstrate characteristics of the method.

In Section 4.5.1, we set out to demonstrate that our method reacts to network latency and limited bandwidth as well as to the performance capability of client and server. We compare against remote rendering and expect the advantages of the hybrid approach to be visible from the results. We used *Dummynet* [CR10] to simulate network conditions.

In Section 4.5.2, we set out to demonstrate that our probabilistic scheduler reacts to run-time change in the performance behavior of a rendering side, even if QLs are barely scheduled for rendering on that side. We make a comparison to a deterministic scheduler, which we expect to behave differently and less accurate.

In Section 4.5.3, we set out to underline the flexibility of the QL concept by adaptively refining a large data set into a variable amount of QLs. We compare this approach to the standard non-adaptive version of the system in a specific scenario and expect the adaptive approach to provide better performance in that case.

Table 4.2 lists the client and server machines that we used for the tests. In Section 4.5.1.1, we used the geometry renderer with the *Thai Statue (TS)* data set (Figure 4.3, 10 million triangles). Otherwise, we used *Tuvok* as the renderer. The data set is the *Visible Human (VH)* (Figure 4.2, 512x512x1884 8-bit voxels). In Section 4.5.3, we used the *Mandelbulb (MB)* data set (8192x8192x8192 8-bit voxels). The rendering resolution is 1280x800. To enable remote rendering, we simply set up a client to not support any QLs.

Table 4.2: Server and client machines used for the results.

Site 1 (Saarbrücken, Germany)		
Name	CPU & Memory	GPU & Network
lab server	Intel i7-2600K @ 3.4GHz	GeForce GTX 680
	16GB	1 GBit/s
fat client	Intel i7-860 @ 2.8GHz	GeForce GTX 560
	8GB	1 GBit/s
thin client	Intel Pentium E5500 @ 2.8GHz	GeForce GT 420
	2GB	100 MBit/s
thinnest client	AMD E-450 @ 1.65GHz	Radeon HD 6320
	3.6GB	1 GBit/s
Site 2 (Salt Lake City, USA)		
orion server	Intel i7-2600 @ 3.4GHz	GeForce GTX 560 Ti
	16GB	1 GBit/s

For equal conditions and reproducibility, we automatized the tests by replaying a four minute set of interaction events that we recorded beforehand. There is a mixture of interactive phases that move the data set in place and idle phases for examination. We repeated the set ten times for each scenario. The results are the averages from the ten runs (thus fractional parts appear).

For each scenario, if not stated otherwise, hybrid rendering is enabled, both sides support all QLs, and no constraints were put on the network link or the rendering performance of either side.

4.5.1 Comparison with Remote Rendering

In the following scenarios, we test the resilience of the the hybrid rendering method towards network latency, limited bandwidth, and server load. We show the synergy the scheduler

creates between client and server. We compare the scenarios to remote rendering to underline the advantages of the hybrid approach.

The upcoming tables and statistics back up our findings. The tables show for each QL how often it was scheduled for rendering on server and client, how often the client actually requested the scheduled QL for rendering, how often the QL was the interactive QL, and how often the QL completed. The request timer described in Section 4.4.3.1 may prevent the client from requesting a QL. A QL may not complete due to rendering abort. In addition to the tables, we state the average time it took to display the interactive QL each frame (iQL DT), the average of the average time it took to refine the view with a non-interactive QL each frame (niQL DT), the total processing time spent on each side (PT client/server), and the total transfer time (TT). For remote rendering, we present no tables as the scheduling is one-sided with only the server being utilized.

4.5.1.1 Heterogeneous Clients

This section examines the scheduler reaction to clients with different capability. We used the lab server. Server and clients are in a LAN, and the latency is therefore negligible.

In *scenario 1* (Table 4.3), we used the thin client.
Frames: 1440.4; iQL DT: 27.4 ms; niQL DT: 325.8 ms; PT client: 30.8 s; PT server: 113.7 s; TT: 11 s

Table 4.3: QL scheduling and rendering with a thin client.

		Scheduled	Requested	Interactive	Completed
QL1	S	507.9	507.9	507.9	505.6
	C	940.4	940.4	940.4	929.5
QL2	S	1371.9	116.6	0.8	105
	C	98.9	6	0	4.7
QL3	S	1371.6	116.1	0	77.6
	C	60.9	5	0	1.9
QL4	S	1146.7	99	0	40.5
	C	57.5	5	0	0
QL5	S	1077	94.8	0	26.5
	C	37.9	3	0	0
QL6	S	1426	122.2	0	25.9
	C	46.9	4.1	0	0.3

The client is able to compete with the server for the simple QL1 (100008 triangles), while the substantially more powerful server is almost exclusively in charge otherwise. The scheduler barely chooses QL2 as the interactive QL, which indicates a stable rendering time gap between QL1 and 2 and consequently low-variance NDs.

The total number of times a QL is scheduled for rendering may exceed or fall below the frame count. Auto-scheduling can cause the scheduler to assign a QL to both sides. On the other hand, the scheduler skips QLs that are expected to complete later than a higher QL. Apart from these two cases, the scheduler assigns a QL to either the client or the server.

In *scenario 2* (Table 4.4), we used the fat client.
Frames: 2019.4; iQL DT: 10.8 ms; niQL DT: 230.8 ms; PT client: 82.9 s; PT server: 119.8 s; TT: 0.4 s

Table 4.4: QL scheduling and rendering with a fat client.

		Scheduled	Requested	Interactive	Completed
QL1	S	52.2	52.2	52.2	46.8
	C	1975	1975	1975	1969.6
QL2	S	1913.1	119.2	0	118.7
	C	324.9	14.1	0	12.9
QL3	S	1294.4	80.2	0	75.8
	C	855.7	52.1	0	28.2
QL4	S	984.5	62.4	0	47.5
	C	1091.5	67.4	0	23.8
QL5	S	1149.8	70.9	0	36
	C	724.9	44.2	0	12.1
QL6	S	1300.5	82.2	0	26.1
	C	779.7	48.1	0	10.7

We repeated the test with remote rendering.
Frames: 1672.2; iQL DT: 21.3 ms; niQL DT: 300.7 ms; PT server: 117.6 s; TT: 2.1 s

We repeated the test with client-side only rendering.
Frames: 2049.9; iQL DT: 10.5 ms; niQL DT: 338 ms; PT client: 121.7 s

This scenario demonstrates a strong synergy of client and server as no side provides a substantial performance lead. Hybrid rendering achieves a faster view refinement compared to both remote and client-side only rendering. For the higher QLs, the focus slightly shifts from client to server.

The about constant encoding and transfer time have less impact on the scheduling decision as the rendering time difference increases in favor of the server.

4.5.1.2 Network Latency and Limited Bandwidth

This section examines the scheduler reaction to latency and limited bandwidth.

In *scenario 1* (Table 4.5), we used the orion server and the thin client. Latency was around 73 ms and bandwidth around 3.2 Mbit/s.
Frames: 564.5; iQL DT: 152.5 ms; niQL DT: 375.4 ms; PT client: 81.2 s; PT server: 28.4 s; TT: 80.1 s

Table 4.5: QL scheduling and rendering in a network setup with moderate bandwidth and latency.

		Scheduled	Requested	Interactive	Completed
QL1	S	48	48	48	43.2
	C	489.8	489.8	489.8	471.2
QL2	S	317.4	59.4	19.1	53.7
	C	181.1	32.3	13.6	28.3
QL3	S	432.2	54.5	0	43
	C	76.8	9.3	0	4.9
QL4	S	558.5	68.2	0	20.1
	C	161.7	16.5	0	0.3

The scheduler reacts to the latency by shifting the interactive QL to the client. The server is still the more powerful machine and therefore in focus to render the remaining QLs. The about constant latency becomes less significant as the difference in rendering time between server and client increases for higher QLs.

Requested QLs do not always complete. The client more likely interrupts the rendering of higher QLs that take long to complete. Further, QL3 and 4 provide detail that is not required for far away views. The renderer thus rejects these QLs for such views, which is a feature described in Section 4.3.2.

We repeated the test with remote rendering.
Frames: 346.9; iQL DT: 285 ms; niQL DT: 440 ms; PT server: 31.4 s; TT: 132.2 s

The iQL display time is greatly increased compared to hybrid rendering. The client cannot bypass the latency by rendering the interactive QL locally. Also, the hybrid approach creates a

schedule in which client and server complement each other. Table 4.5 shows that the server does not have to bother with QL1 most of the time. The server can thus complete the non-interactive QLs for view refinement faster, which results in the lower niQL display time compared to remote rendering.

In *scenario 2* (Table 4.6), we used the orion server and the thin client. We simulated a bandwidth reduction to 500 KBit/s and an additional latency of 150 ms on top of the actual 73 ms.
Frames: 551.4; iQL DT: 138.7 ms; niQL DT: 731.4 ms; PT client: 118.2 s; PT server: 21.7 s; TT: 172.2 s

Table 4.6: QL scheduling and rendering in a network setup with low bandwidth and high latency.

		Scheduled	Requested	Interactive	Completed
QL1	S	7	7	7	1.8
	C	490.6	490.6	490.6	475.7
QL2	S	55.1	9.3	0	7.2
	C	489.7	113.8	59.8	86.7
QL3	S	208.2	30.8	0	18.7
	C	352.9	39.5	0	17.1
QL4	S	542.7	65.9	0	18.3
	C	152	16	0	0.1

The additional network constraints significantly increase the transfer time. The scheduler thus shifts the focus further to the client. While QL4 is still almost exclusively scheduled on the server, the client now concentrates on QL1, 2 and 3.

We repeated the test with remote rendering.
Frames: 133.2; iQL DT: 760.5 ms; niQL DT: 1400.8 ms; PT server: 16 s; TT: 241.2 s

The iQL display time shows that the remote rendering can barely maintain an interactive frame rate. We set up the client to abort the rendering of the interactive QL if the QL does not complete within one second. The client indeed regularly discards the interactive QL for close views that show a lot of detail. For QL1, the abort-to-request ratio is 0.5. Aborting the interactive QL, which is the first QL of a frame, ultimately means display updates are lost, and the user input becomes decoupled from the visible result. The situation is even worse than the iQL display time indicates as the measurement only includes QLs that could complete within a second. Even though the client regularly issues a rendering abort for the interactive QL, the

server likely already finished the QL and sent it to the client as the delay on the client is caused by the network and not the rendering.

The hybrid approach maintains interactive performance regardless of the network and server conditions as long as the client renders at least the first QL at interactive frame rates. Therefore, when mapping a renderer's data sets to QLs, it is reasonable to make the first QL render interactively on all target hardware even if this means reducing the QL's visual quality considerably. The scheduler is able to automatically choose the highest QL adhering to a target frame rate, and as a consequence the low quality QL only comes into play if there is no alternative.

In *scenario 3* (Table 4.7), we used the lab server and the thin client. We simulated a latency of 200 ms. Bandwidth is a negligible factor for the high-speed local area link.
Frames: 585.4; iQL DT: 138.2 ms; niQL DT: 558.7 ms; PT client: 111.1 s; PT server: 15.1 s; TT: 126.4 s

Table 4.7: QL scheduling and rendering in a network setup with high bandwidth and latency.

		Scheduled	Requested	Interactive	Completed
QL1	S	8.5	8.5	8.5	3.1
	C	523	523	523	499.3
QL2	S	113	18.8	1.3	18.2
	C	435.7	102.8	57.6	81.5
QL3	S	335.9	42.2	0	35.3
	C	238.9	25.5	0	12.3
QL4	S	582.4	69.6	0	29.1
	C	159.1	16.3	0	0.1

The scheduler mostly assigns QL1 and 2 to the client to bypass the latency. For QL3 and 4, which take substantially longer to render on the client than on the server, the server remains primarily in charge. The client unburdens the server and vice versa.

We repeated the test with remote rendering.
Frames: 202.4; iQL DT: 531 ms; niQL DT: 842.1 ms; PT server: 16.6 s; TT: 200.1 s

The client cannot bypass the latency, and neither side can take load off the other. Thus, iQL and niQL display time increase compared to hybrid rendering.

4.5.1.3 Server Load

This section examines the scheduler reaction to server load.

We used the lab server and both thin and fat client. The thin client captured the measurements. We ran 19 remote rendering sessions on the fat client to generate server load. The fat client sessions kept running until the thin client finished. Table 4.8 shows the scheduling result.
Frames: 529.3; iQL DT: 150.4 ms; niQL DT: 921.8 ms; PT client: 136.7 s; PT server: 148.6 s; TT: 0.9 s

Table 4.8: QL scheduling and rendering with a server under high load.

		Scheduled	Requested	Interactive	Completed
QL1	S	20.2	20.2	20.2	16.4
	C	453.1	453.1	453.1	445.6
QL2	S	110.3	19.3	3.8	16.2
	C	407.7	102.2	57.8	84
QL3	S	222.7	30.3	0.3	20.1
	C	323.3	39	0	16.5
QL4	S	386.8	48.3	0	5.8
	C	296.3	36.8	0	2.8

The scheduler responds to the server load by shifting QLs to the client. Except for QL4, the focus is on the client. We observe a low completion rate of QL4 since both loaded server and thin client can hardly deliver in time before the next interaction event interrupts the progressive refinement.

We repeated the test with remote rendering. The performance loss compared to hybrid rendering is substantial.
Frames: 303.7; iQL DT: 310.6 ms; niQL DT: 1594.4 ms; PT server: 226.5 s; TT: 4.3 s

To conclude, the previous three sections demonstrated that the hybrid rendering method can overcome limited client capability, latency, limited bandwidth, and server load by shifting rendering work between client and server. The method is able to create a synergy of the two sides to improve the user experience. We demonstrated the advantages over a remote rendering system. The findings confirm the expectations we originally formulated.

4.5.2 Comparison with Deterministic Scheduling

In the following scenarios, we compare our probabilistic scheduling method to a deterministic approach that always produces the same schedule from a given input. We examine the reaction to run-time condition changes. We identify similarities and differences and outline the

advantages of the probabilistic scheduler.

To enable deterministic scheduling, we set up the scheduler to always use the mean of a ND.

We modified the system to simulate a run-time change in the rendering performance. Client and server can delay the rendering of a QL. In each run, the condition change hits after two minutes. We performed the tests with a new four minute interaction set that represents a continuous rotation in a fixed distance. We used QL1 only. The usage of the simplified interaction set with just a single QL demonstrates the behavior of the scheduler over time most clearly.

In *scenario 1*, we used the thin client and the lab server. The condition change is a slow down of the server performance by a factor of 7.5. Figure 4.6 shows the schedule amount of QL1 on each rendering side over time in milliseconds. The schedule amount is the number of times a QL was scheduled for rendering. The histogram scale of client and server is not equivalent. The server without delay is substantially faster, and thus a higher frame rate is present if the server is the active side. The active side is the rendering side the scheduler focuses on. In contrast, the scheduler only infrequently assigns the QL to the idle side.

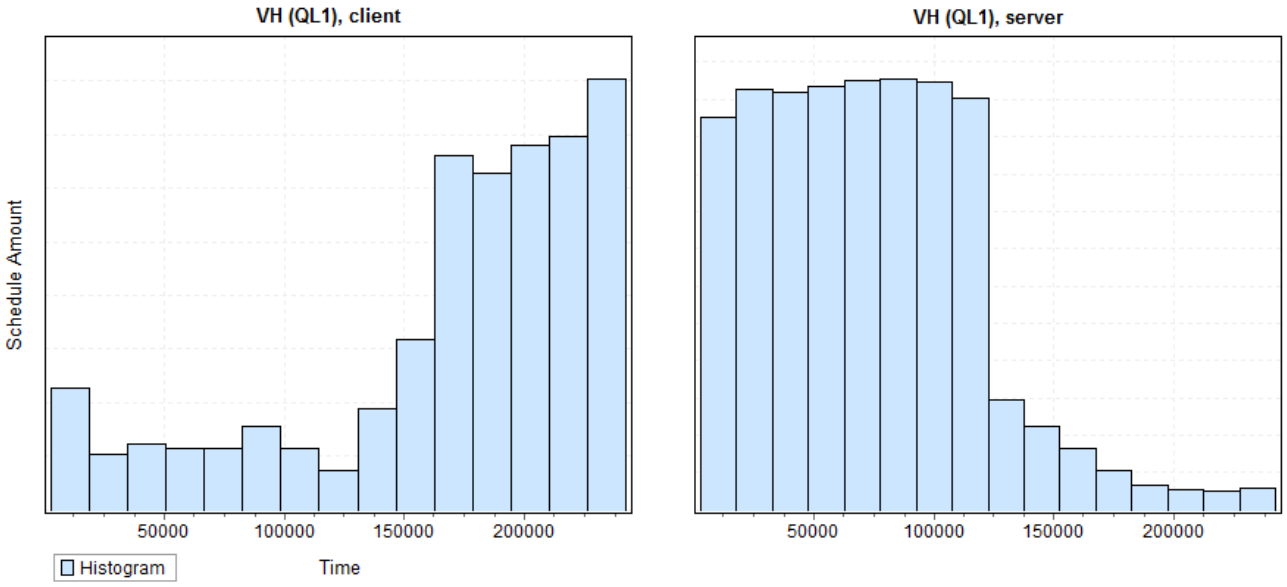


Figure 4.6: Schedule amounts determined by the probabilistic scheduler in response to a server slow down.

As expected, the server is the active side until the slow down occurs. Due to the probabilistic nature of the scheduler, the client side is still sampled from time to time. The distribution initialization process (Section 4.4.4), which facilitates timing acquisition to adapt to the unknown conditions, is reflected in the initially higher schedule amount on the client. Once the slow down hits, the client schedule amount first decreases before starting to climb. The transition to the client as the active side is not immediate. The focus stays briefly on the slowed server,

which results in a lower frame rate and thus fewer chances for the scheduler to assign the QL to the client. The reason for the incremental shift is the proximity in rendering performance between the slowed server and the client. The advantage of the client is not substantial. The weighting function also affects how quickly new timings cause the overtaking. The scheduler's behavior is thus valid given its parameters and the conditions.

We repeated the scenario with the deterministic scheduler. We provide no histograms as the server is the only active side.

Rendering on the client stops after the initialization phase until the slow down occurs. Since the slow down occurs on the active server, the deterministic scheduler notices the change. However, unlike our expectation, the scheduler does not switch rendering persistently to the client. First, the server side distribution does indeed fall behind the client, causing the rendering of the QL on the client. However, the client timing is an outlier as there was no client-side rendering since the beginning. The client's distribution was thus also not updated since the beginning, and the outlier has a huge impact. As a result, the server's distribution, even though it reflects the slowed down state, stays in front in the end. The problem is the abrupt transition with only a single guaranteed rendering sample on the client, which is prone to outliers.

In contrast, the probabilistic approach enables a smooth transition when one distribution approaches another. Each timing affects the subsequent scheduling decisions, accelerating the shift in the direction of change. In a chain-reaction, the chances for one side gradually increase while they decrease for the other. This causes an outlier-resistant transition, with multiple samples taken from both sides until a stable state representing the changed conditions has been reached. The approach is flexible due to the weighting function that can either favor fast reaction to change or outlier-resistance. The deterministic approach is not able to absorb client-side outliers, which results in permanently keeping up a bad scheduling decision.

We repeated the scenario with the deterministic scheduler, replacing the linear with an exponential weighting function (Figure 4.7).

Now the scheduler performs the switch to the client similarly to the probabilistic approach. The exponential weighting function causes a distribution reset (Section 4.4.5) on the idle client about every 45 seconds. The reset allows the scheduler to obtain several client-side timings depending on the initialization parameters, which here is enough to absorb initial outliers.

Consequently, a deterministic scheduler that probes the idle side with a certain interval can achieve a similar behavior than the probabilistic approach. This also applies to the next scenario, which simulates change on the idle side. However, the probing frequency is arbitrary and does not reflect the actual conditions. There is no chain-reaction to smooth and accelerate

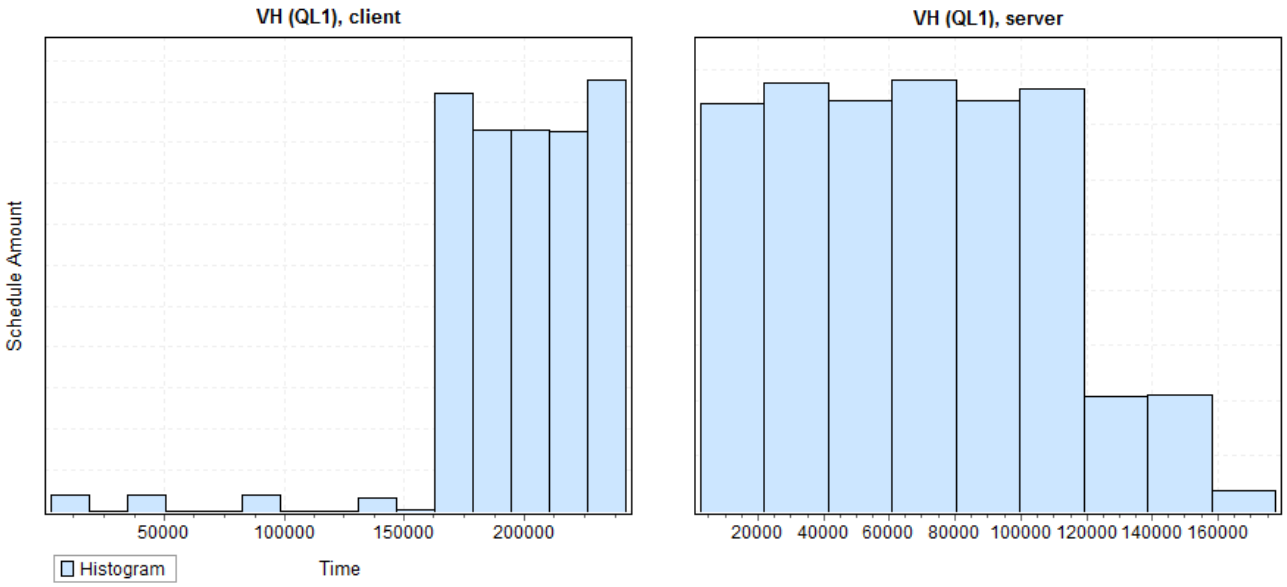


Figure 4.7: Schedule amounts determined by the deterministic scheduler with an exponential weighting function in response to a server slow down.

the adaptation process as timings do not affect the scheduling decision until one distribution finally overtakes the other, which causes an abrupt transition. Probing is no generally applicable concept that can adapt to any situation. A low frequency may result in reacting to change late. In our scenario, the distribution reset that triggers the switch to the client would be about 40 seconds late if the slow down occurred at the 95 second mark. The timings obtained after a reset might not be enough to complete the shift of a distribution. Outliers might heavily influence these timings. A high probing frequency might result in generating unnecessary load if conditions are stable and the two sides far apart performance-wise. If the performance is similar, the probabilistic scheduler is able to react by equalizing the schedule, which is especially relevant to balance multiple-client scenarios. The deterministic approach would instead focus on one side while probing the other with an arbitrary interval that does not take the performance proximity into account.

The results underline the benefit of the generic concept behind our probabilistic scheduling method, which should adapt to any unknown situation without requiring specific parameters or workarounds.

In *scenario 2* (Figure 4.8), we used the thin client and the lab server. The condition change is a speed up of the server performance by a factor of 7.5. The server slows its performance down by this factor initially and returns to normal performance to simulate the speed up.

As expected, the client is the active side until the speed up occurs on the server. The probabilistic scheduler notices the change as it continues to sample the server from time to time. In contrast to *scenario 1*, the transition concludes quickly. Once the server regains full perfor-

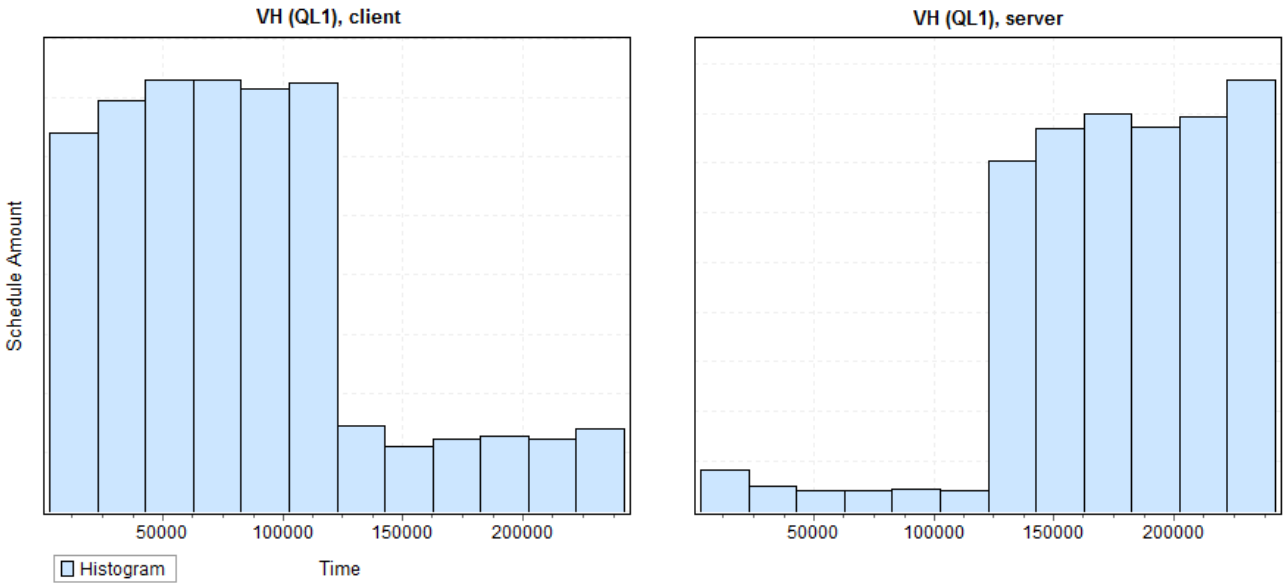


Figure 4.8: Schedule amounts determined by the probabilistic scheduler in response to a server speed up.

mance, the difference to the client is substantial, which allows the server-side timings to quickly put the server in front.

We repeated the scenario with the deterministic scheduler. We provide no histograms as the client is the only active side.

The scheduler does not notice the condition change. The server is left alone entirely after the initialization phase. Without a workaround, such as forcing to reactivate the idle side every N time steps, the deterministic scheduler is not able to react to change that occurs only on the idle side.

We repeated the scenario with the deterministic scheduler, replacing the linear with an exponential weighting function (Figure 4.9).

Now the scheduler performs the switch to the server similarly to the probabilistic approach. The distribution reset caused by the exponential weighting function triggers auto-scheduling of the QL on the server. The scheduler can thus obtain several timings, which are enough to trigger the transition, though at a later point. However, the result is again specific to the situation and the scheduling parameters chosen and does not reflect a generically applicable concept. The small schedule amount peak on the client in the end is also induced by a distribution reset.

Summarizing, the probabilistic scheduler has an advantage in absorbing outliers, especially in the transition phase when one side overtakes the other, and in reacting to change on the idle rendering side. Still, a problem with the latter can occur if the probability for a QL to get scheduled for rendering on the idle side gets very low or even towards zero. Such a scenario is

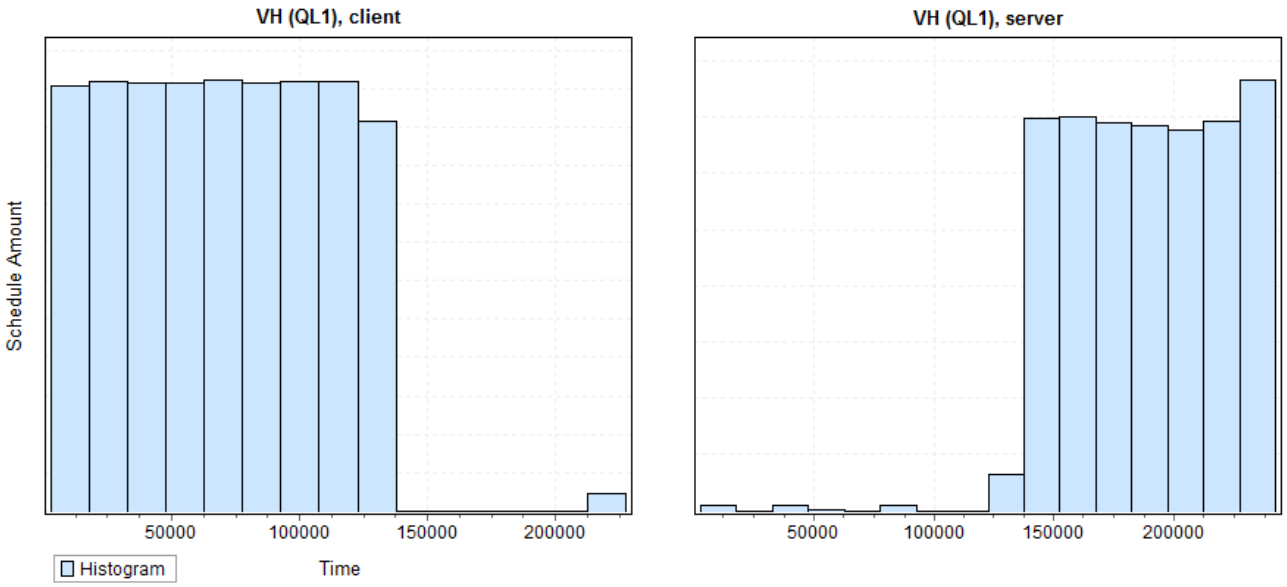


Figure 4.9: Schedule amounts determined by the deterministic scheduler with an exponential weighting function in response to a server speed up.

possible if the two sides are far apart performance-wise and timing variance is low. We assume condition changes can affect the situation any time. The scheduler cannot predict the changes given the current state represented by the distributions. Therefore, a mechanism independent of the state is required to guarantee that the scheduler notices change in a timely manner. The customizable weighting function allows to address the problem. The function could trigger a reset faster or in predefined intervals. It is up to the developer what kind of logic is put into the function. We provide predefined options including the linear and exponential versions used in this chapter.

In addition to modifying the weighting function, the scheduler supports another more generic solution that is called probabilistic auto-scheduling. The more a QL's distribution approaches the reset threshold in terms of the weighting function, hence the longer the distribution has not been updated, the higher is the probability for the QL to bypass the scheduling algorithm and get automatically selected for rendering. The scheduler evaluates the decision every frame for every QL. Parametrization is possible to control how fast and to what maximum the probability rises. The mechanism is an optional component and recommended in environments where substantial condition changes are expected. Using auto-scheduling extensively can cause unnecessary load and thus be counter-productive.

Concluding, our probabilistic scheduler is able to react to changes in the performance behavior of active and idle side. While a deterministic approach can achieve similar results in some situations, such a method especially fails to adapt to change on the idle side reliably. Our system provides parameters and optional features to tune the scheduling if desirable. Probabilistic

auto-scheduling and an exponential weighting function have proven to be viable options to accelerate the adaptation process.

4.5.3 Adaptive QL Mapping

This section demonstrates how our system enables the interactive rendering of a large data set on a client with limited resources. We used the thinnest client and the fat client as the server. The thinnest client is restricted in disk space, with only 60GB available in total. This is not enough to store the MB data set in its uncompressed form. In its compressed form, the MB still occupies more than 23GB, which makes it infeasible to store the data set on such a machine (considering an additional 15GB for the operating system alone). We therefore only enable a subset of the QLs on the client.

Further, the client is restricted in rendering performance. The MB has seven LODs. When using a direct LOD-to-QL mapping, even the first QL renders barely interactively. We tested the interaction set from Section 4.5.1 and measured an iQL display time of 620.2 ms with client-side only rendering of QL1. To improve interactivity and enable a more fine-grained view refinement, we extended the QL mapping for *Tuvok* with an adaptive approach that allows to split each LOD into a variable amount of QLs. These QLs are distinguished by the sampling frequency and resolution at which they are rendered. Frequency and resolution are run-time parameters of the renderer. Figure 4.10 shows example QLs for a LOD of the MB.

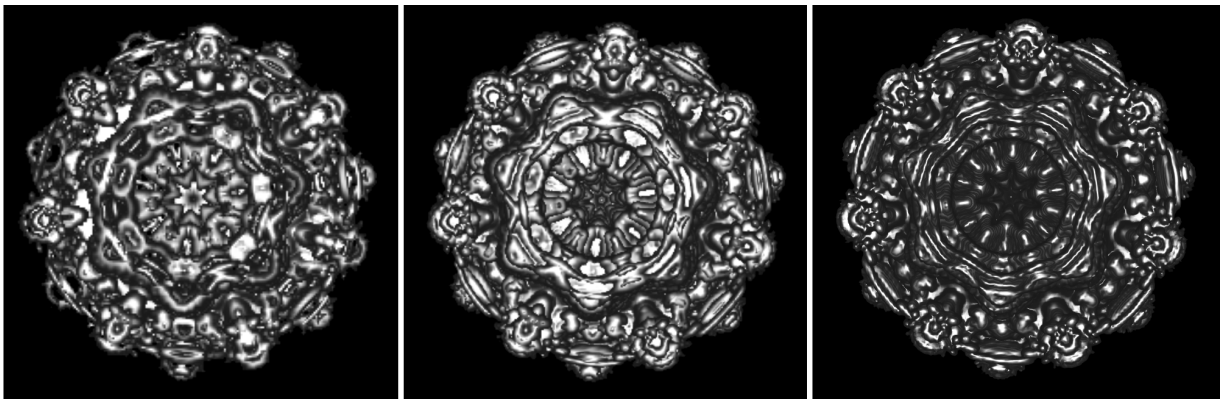


Figure 4.10: LOD1 of the MB split into three QLs. QL3 (right) renders at full resolution and with the default sampling frequency. Resolution and frequency decrease for the lower QLs to enable faster rendering.

For the test, we split each LOD into two QLs, resulting in 14 QLs overall. We introduced 28 ms of network latency to facilitate iQL rendering on the client. The client supports QL1 to 6. Frames: 734.5; iQL DT: 82.7 ms; niQL DT: 286.8 ms; PT client: 42.4 s; PT server: 101.7 s; TT: 35.8 s

The client mostly renders the iQL (62.7% share). Though, for close views that demand more rendering time, the scheduler regularly switches the iQL to the server as the impact of the latency fades. The server almost exclusively handles the niQLs.

We repeated the test without the adaptive QL mapping, thus ending up with seven QLs. QLN is equivalent to QL$2 * N$ in the adaptive mapping. The client supports QL1 to 3.
Frames: 514.1; iQL DT: 146 ms; niQL DT: 487.1 ms; PT client: 14.8 s; PT server: 114.1 s; TT: 37.8 s

The server performs almost all rendering including the iQL. The latency has no impact on the schedule as the performance difference between client and server is too substantial. Both iQL and niQL display time are increased compared to the adaptive approach.

The results underline the flexibility of the abstract QL concept, which allows an arbitrary, application-specific mapping of a data set to QLs. The adaptive mapping substantially improved the interactive rendering of the demanding MB data set.

4.6 Normal Distribution Usage

Our system obtains discrete timings to gradually approach the underlying continuous distribution, which we assume to be normal. In general, the distribution characteristics are unforeseeable. They are dependent on unknown factors especially attributed to renderer, operating system, hardware, and network. The distribution type may differ between data sets, QLs, and even views. Along with changing conditions, distribution characteristics may also change. Insight could be obtained for a specific hard- and software setup with stable conditions. However, such an offline assessment does not apply to our generic approach that should adapt to variable run-time conditions. We support arbitrary server and client machines and make no assumptions about the possibly changing environment during a rendering session.

We performed a number of tests using the machines and data sets presented in Section 4.5 to confirm the ND is a reasonable choice in the majority of cases. In each run, the test machine rendered a single QL repeatedly for four minutes. There was no additional load on the machine. We performed the runs with a static view as well as with the data set slightly and continuously moving (rotating or zooming in and out). Figure 4.11 and 4.12 show millisecond timing histograms with NDs fitted to the data for a selection of runs. A ND has proven to be a good fit in most cases.

When enabling the movement, we could still fit NDs of increased variance to much of the data. In some cases, the different views caused by the movement resulted in several peaks in the

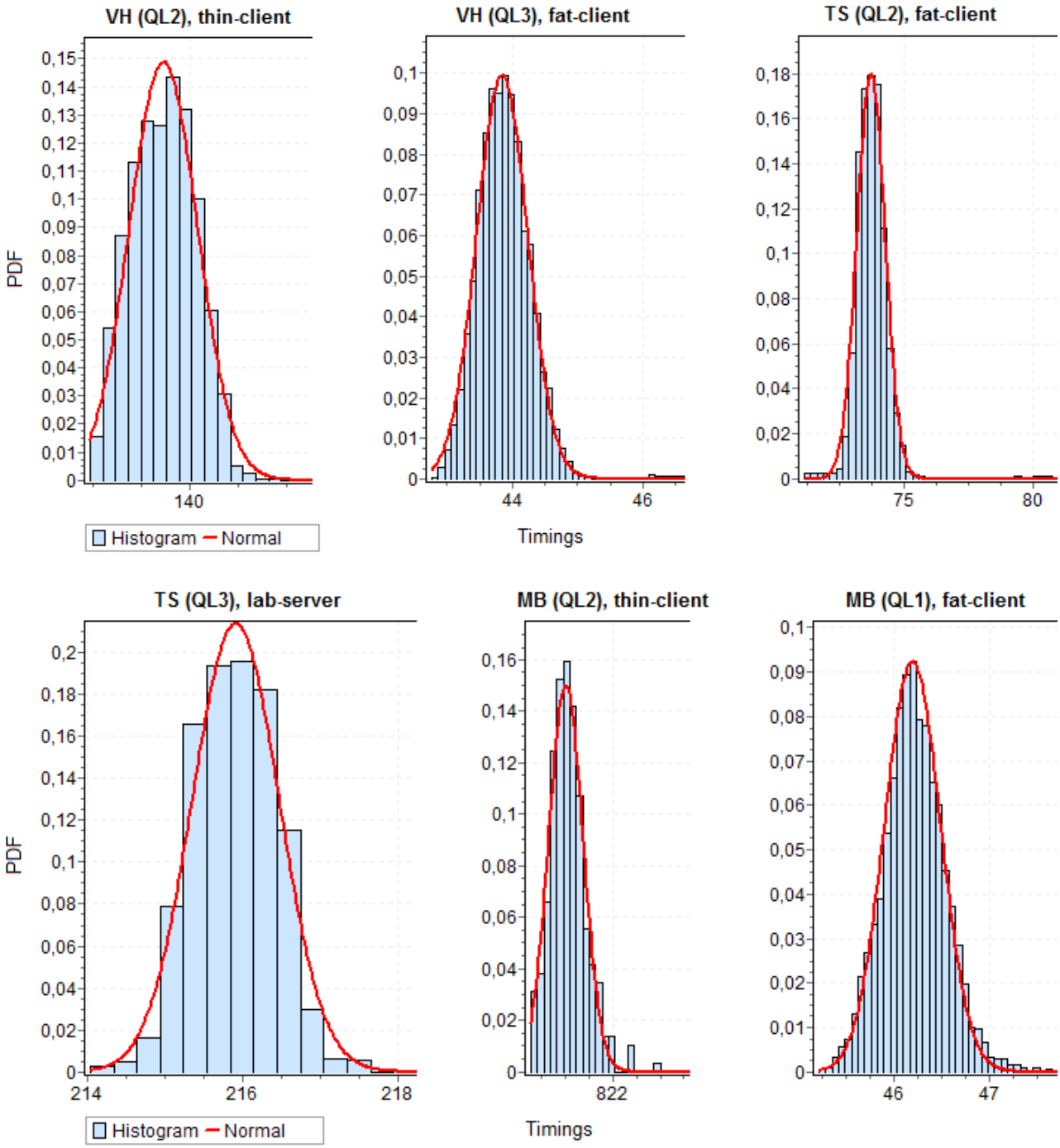


Figure 4.11: A ND fitted to the histogram of processing timings in six scenarios.

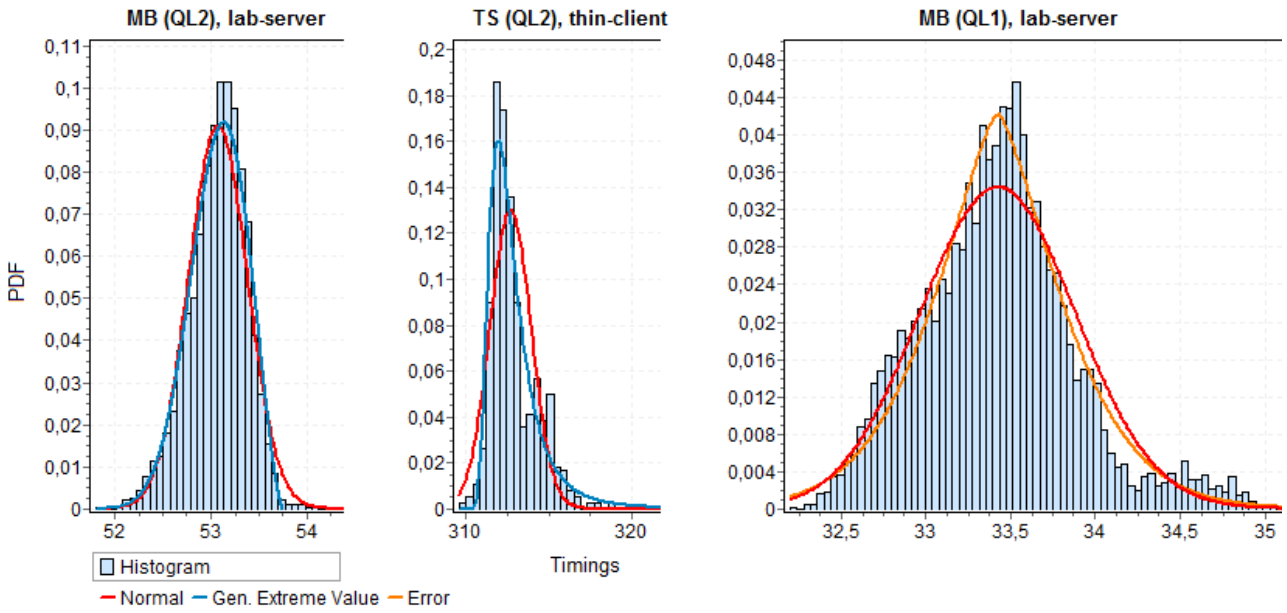


Figure 4.12: A ND fitted to the histogram of processing timings in comparison to a generalized extreme value distribution and an error distribution.

histogram and thus more than one distribution being present. This is expected and we assume a ND can be fit to the timings for a specific view.

We further performed tests using the fat client and the orion server to obtain transfer timings. Generally, we could fit a ND with very low variance to the data. However, major outliers were regularly present in some test runs. The client reaches the orion server via the Internet, and there are possible external influences on the link.

The results show that the ND decently represents the timing data in our system. The choice to approximate the performance state with NDs is thus feasible. We ultimately decided to use the ND as the default distribution type in our generic method.

However, isolated scenarios with a stable setup of client, server, and network components as well as renderers and data sets are possible. In such a case, tests like the ones described in this section could be performed to gain an understanding about the distribution characteristics. Figure 4.12 also fits a generalized extreme value distribution (EVD) and an error distribution to the timing data. Those distributions are more suitable in the depicted cases. A ND is still a reasonable fit. Though, the ND deviates more clearly from the histogram in the middle scenario of Figure 4.12.

While our system uses NDs by default for the deployment in a heterogeneous and uncertain setup, the system is not bound to NDs. The pre-sampling strategy is independent of the distribution type as the strategy only relies on random sampling. Thus, the system is config-

urable and allows the replacement of the distribution type. There are other parameters like the weighting function that can be used for adjustment. In the future, we will investigate the incorporation of additional distribution types. Especially the EVD has shown to fit some timing data well. We will also investigate setting the distribution type for each QL individually to account for the possible difference between QLs.

4.7 Conclusion and Discussion

We presented a generic hybrid rendering method that distributes workload to server and client in terms of QLs of a data set. We use a probabilistic scheduling algorithm to account for the various uncertain factors when determining which QLs are to be rendered on which side. The system obtains and updates timing NDs for the QLs at run-time to adapt the schedule to initially unknown and potentially changing conditions. The weighting system to add timings absorbs outliers. The method balances the utilization of server and client resources. Client-side rendering capabilities reduce the dependency on server and network. Utilizing the client puts less load on server and network and thus improves the scalability of the system. We demonstrated the usability of our approach for renderers of multi-resolution data sets, in particular for a LOD-based volume renderer and a progressive geometry renderer.

In a steady situation, the schedule converges to a state that reflects the current conditions. However, the adaptation may not be immediate due to the possible irregularity in the scheduling of QLs, which we addressed in Section 4.4.3.1. Given the scheduler's goals and probabilistic view, the irregularity is expected behavior. But this implies two problems.

First, if the probability for QLs to be rendered on a side is low, the scheduler may only slowly react to a condition change on that side. This especially applies to server QLs, where substantial timing fluctuation is more likely due to arbitrary server and network load. The user does not benefit from an improved server performance until the scheduler incorporates this change. The speed of the adaptation process is also dependent on whether and in what direction conditions change on the other rendering side. If the client performance decreases, the probability for server-side rendering indirectly increases. Further, the weighting function determines how much a timing impacts a ND. But increasing weights fast makes a ND less resistant to outliers. Also, the scheduler must first acquire a timing. With arbitrarily low probabilities being possible, resetting a ND and probabilistic auto-scheduling ultimately enable to update the state.

Second, if conditions on a side change rapidly, the scheduler may constantly be behind with its predictions, even if timings come in regularly. An incremental shift of a ND may be counter-

productive if the conditions quickly change again in the opposite direction. This is more likely for server QLs, where a sudden increase or drop in load is possible. A weighting function that increases the impact of the next timing fast helps dealing with such volatile conditions.

The scheduler may schedule non-interactive QLs, but the client may not request these QLs for rendering due to the request timer described in Section 4.4.3.1. The scheduler may thus only sporadically receive measured or estimated timings for non-interactive QLs. Consequently, setting up QLs that have a high expected completion time to reach initialized and meaningful NDs with fewer timings along with giving new timings more weight can be reasonable.

Finally, we briefly outline a possible approach to let the scheduler adapt faster in response to server load changes. For each QL, the scheduler stores the server load that was present when the QL's processing time ND was last updated. The scheduler can then determine whether a ND still approximately reflects the current server load. If not, the scheduler could decide to switch to auto-scheduling. For the concept to be beneficial, the server load metric must be accurate. If the scheduler draws conclusions due to incomplete or inaccurate assumptions, the correctness of the method is undermined, even if improved scheduling results occur in some scenarios.

4.8 Future Work

We plan to deploy the system more widespread by incorporating additional devices, especially mobile ones, which are placed at different locations. We want to incorporate the interleaved data transfer in future tests. We are also considering a cloud computing environment for deployment. The cloud becomes increasingly popular to provide services to heterogeneous clients. Hybrid rendering could be such a service.

We described two scheduling strategies. In both, the goal is to provide the user with the next QL as soon as possible. We want to investigate the design of an alternative strategy that gives more priority to the scalability of the system. The strategy would not necessarily select the schedule optimal for the user but find a compromise that is still acceptable in terms of user perception but puts less load on server and network.

Further, when determining the next non-interactive QL for rendering, the scheduler always selects the one expected to complete earliest. However, the time difference to following QLs may be minimal in terms of user perception. It could be beneficial to prefer a higher QL or the client instance if the user would barely notice a delay or missing refinement step. The strategy would enable to put less load on the system while possibly reaching the highest QL faster.

Chapter 5

The Browser as the Platform for Remote Visualization

5.1 Introduction

Graphics-intense applications like scientific visualization and games require computing and storage resources that may not be available on all display devices. Especially mobile devices may lack the capabilities to handle large scenes and data sets at interactive frame rates or at all. The browser as the execution platform imposes additional challenges as applications run in a secured environment that restricts access to persistent storage and native libraries. WebGL is a limited subset of OpenGL and thus impairs the options of rendering algorithms. Exascale visualization may make it infeasible to even download simulation data from a supercomputer. Further, the data may be confidential and must not be sent to a client.

Remote rendering tackles these restrictions, most importantly by providing visualization to devices with limited resources. A rendering server or a cluster of servers generates images and transfers them to the client for display. Client-side requirements are minimal. However, network latency, bandwidth, and reliability can impact responsiveness and quality of the application. Therefore, incorporating both client- and server-side rendering is a feasible approach, which we have demonstrated in Chapter 4.

The browser has established itself as a ubiquitous application across operating systems and platforms, step by step closing in on functionality otherwise provided by native applications. Persistent network connections via WebSockets (WS), client-side graphics via WebGL, as well as audio and video media support are now widespread available as built-in features that require no plugin. Efforts to reflect these developments in HTML-conform standards and thus increase

the accessibility for web developers emerge. Examples are declarative 3D [SKR+10, BEJZ09] and real-time communication [LR12]. Further, the cloud concept becomes increasingly popular to provide data and services to users anywhere. These developments enable building new types of browser applications like games, virtual worlds, visualization, and videoconferencing.

In this chapter, we provide a classification and description of the technologies that enable plugin free remote rendering in the browser. We present an interactive remote visualization system that unifies the technologies. Supporting several technologies enables widespread support across desktop and mobile browsers as well as adaptivity to network conditions and application requirements.

While developing a browser plugin or using an existing one like Flash for the client functionality is one solution, this has several disadvantages. Plugins with full privileges on the client system (NPAPI, ActiveX) are a stability and security risk. The installation thus requires a user dialog. There is no standardized plugin mechanism across browsers, which complicates development. The provider therefore has to maintain a tailored plugin version for each supported browser. Existing plugins may not be available on all platforms (for example Apple does not support Flash in their mobile products). Browser developers move away from plugins [Sch13, Sme15, Mic15] and instead continue to extend the browser's functionality to support more use cases. Consequently, we do not consider plugins future proof to develop web applications.

Our solution therefore stays close to HTML5 and within the functionality browsers provide today. Adhering to W3C standards simplifies the possible integration into other compatible technologies and improves the accessibility for web developers. In Chapter 6, we integrated the remote rendering techniques presented here into XML3D [SKR+10]. If browsers widely adopt standardized functionality, a single, portable client application with no or minor cross-browser tweaks can be maintained. This ultimately enables users to access the application from any device that runs a capable browser.

Our system supports several methods to transport images to the client. The server can send images encoded with JPEG, Motion JPEG (MJPEG), and S3 texture compression (S3TC) [HIN99].

Furthermore, the server utilizes the WebRTC [LR12] technology to stream video directly into the web page's *video* element. WebRTC is primarily intended for browser to browser real-time communication via webcam and microphone. The video stream is thus optimized for low latency, which is a major requirement in our interactive context. Consequently, we have adapted the WebRTC framework and plugged in our rendering component.

Finally, we support the Native Client (NaCl) [YSD+09] technology available in Chrome to

receive a video stream. NaCl can run native code safely within the browser's sandbox, which allows to close in on the performance of a native application. However, as of 2017, NaCl has been declared deprecated in favor of WebAssembly [Nel17], which is a cross-browser solution for high performance code in the web. WebAssembly already runs in multiple browsers and is in the process of standardization. However, it still lacks several of the API features that NaCl offers [Goo17]. We expect the gap to close until NaCl support is removed from the open web in early 2018.

The remainder of the chapter is structured as follows: In the next section, we describe related work in the area of remote rendering in the browser, with a focus on real-time video streaming. We then provide a classification of the technologies that match our requirements. The implementation section describes the visualization system that unifies the technologies. The results section provides measurements and comparisons. The chapter finishes with a conclusion and future directions.

5.2 Related Work

One application area of remote rendering is visualization. EnVision [JMWJ09] enables remote rendering in Java-enabled browsers and uses Virtual Network Computing [RSFWH98] as the strategy to deliver rendered images from the server to a Java applet. Yoon and Neumann [YN00] describe an early server-backed system that combines ray-casting with image-based rendering and also enables browser access via a Java applet.

ParaViewWeb [JJAM11] is a visualization framework for the web, which allows to receive remotely rendered images within a Java or Flash plugin. In addition, a plugin free client using HTTP long-polling [Lor11] is available. Other plugin free approaches that rely on HTTP exist [JKKM+03, SBEF14]. McLane et al. [MCY+10] use Ajax communication to receive *base64* encoded JPEG images in an XML response. Dyken et al. [DLS+12] employ a browser-based client for their hybrid geometry rendering system. The client draws a coarse version of the data set interactively while the server generates more detailed views on demand. To receive server-generated images, the client issues HTTP requests including support for long-polling.

HTTP requests represent no persistent connection. After the client has received an image, the browser drops the connection and the client needs to reconnect to send updates and request the next image. This is a considerable overhead in our context where the server performs rendering at interactive frame rates. In contrast, the overhead is negligible in systems that use the server only for static view refinement [DLS+12, SBEF14].

An approach that induces less overhead and provides more flexibility at least for the image transfer is MJPEG over HTTP. Here, the browser keeps a connection open to receive consecutive images as a multipart message. Kaspar et al. [KPS10] use MJPEG in their remote volume rendering system.

While techniques that enable bidirectional communication with HTTP exist, HTTP is ultimately not designed for such usage, and multiple issues can arise from using these techniques [Lor11]. Hence, relying only on HTTP complicates the implementation of an asynchronous pipeline that enables the client to request several frames in advance, thus facilitating server utilization, and then receive resulting images on a separate channel.

With the advent of WS, persistent connections are now possible and provide a viable alternative to communicate with the server. Wessels et al. [WPJR11] use the JS WS API to receive JPEG images, which is similar to our image-based implementation. ParaViewWeb has been extended to support WS [MPJ+13].

Behr et al. [BMP+15] describe a service infrastructure for visualization applications in the browser. The framework includes client-side rendering using the *hare3d* [SLTB15] library in addition to a server-side rendering component that supports WS image transport.

While most of the above visualization systems focus on individual image encoding, video encoding can provide superior compression, which makes it especially viable in situations where bandwidth is a bottleneck. Multiple clients may be active at the same time and generate network load. Video streaming can also absorb packet loss to some degree. The user perceived quality might still be acceptable even if artifacts appear. Video is thus especially suited for unreliable networks like the Internet or wireless links.

Cloud gaming platforms [CCT+11], which need to serve a possibly large number of clients concurrently, have widely adopted video streaming. There is research to optimize the video stream for such platforms [HJNS+13, SHNC11, SSB09]. Games@Large [JFE+09, FE10] optimizes the stream using rendering context information, which is particularly related to our adaptive video streaming outlined in Section 2.4.4.

Chen et al. [CCT+11] list a number of cloud gaming platforms. We only consider the ones that provide a browser client. The now discontinued service OnLive [OnL16] provided demos within the browser while the user needed to download a native client for full games. The game streaming service Gaikai, which has now been integrated into PlayStation Now [Sar14], could deliver games as a video stream to Java or Flash enabled browsers. Gaikai also developed a plugin free client using NaCl [YSD+09].

The HTML5 *video* element does currently not support real-time video streaming even with

standard protocols such as RTSP [Moz16]. The fall back to WS transport and JS video decoders is an option to implement the functionality plugin free in the browser. However, even with ongoing improvements in JS performance and features, JS video decoders like Broadway.js still lack behind native counterparts that may even utilize hardware-acceleration.

Otoy presented the ORBX codec [Oto15a, Oto13], which is supported by ORBX.js for efficient video decoding purely with JS and WebGL. The approach does thus not rely on browser-specific video codecs. In cooperation with Mozilla, Autodesk, and Amazon, the technology has been deployed on Amazon Web Services to provide remote gaming and desktop applications. However, ORBX is not publicly available.

A way to transmit live video to the browser without a plugin is segmented streaming over HTTP. Bringuier [Bri11] discusses and compares existing methods. While there are a number of proprietary solutions (Apple HTTP Live Streaming, Adobe HTTP Dynamic Streaming, Microsoft Smooth Streaming), DASH [Sod11] is an ISO standard that is already widely used by service providers such as Netflix or Google. But on the browser side, there is no native DASH adaptation and interoperability yet. However, the upcoming Media Source Extension standard (MSE) [W3C16a] allows to construct media streams for the HTML5 media elements in JS. All major platforms except iOS already support MSE. MSE enables to implement support for DASH with JS and the *video* element, and portable implementations such as dash.js or Bitmovin Player exist.

Live streaming methods like DASH require the segmentation into small video files, which the client continuously downloads via HTTP. This introduces overhead and buffering delay, which our interactive remote rendering application is very sensitive to (in contrast to a sports event or similar, where even several seconds delay are usually acceptable). Zorrilla et al. [ZMS+14] use DASH for their remote rendering but also demonstrate the substantially increased latency of segmented streaming compared to RTSP [ZMM+12]. The proprietary MSE-based solutions Unreal Media Server [Unr16] and EvoStream [Evo16] reach sub-second latency. Unreal Media Server uses WS transport and supports video segments of very small size, but the minimum delay is still around 200 ms.

We require instant reaction to user input. WebRTC [LR12] has been designed for real-time purposes and is an upcoming standard with a JS API. All major browser vendors already adopted the standard. To our knowledge, the system presented in this chapter is the first to utilize WebRTC for remote rendering. Later systems [QLB+16] also employ the technology, which underlines the feasibility of the approach.

While HTTP streaming can avoid traversal issues with Firewalls and Network Address Translation (NAT) in contrast to UDP/RTP based solutions, this is no major requirement for us.

We assume that a dedicated server is available, which the client can connect to. Also, WebRTC has mechanisms in place to overcome connection establishment issues.

5.3 Classification

In this section we identify five methods that are suitable for plugin free remote rendering in the browser with minimal response time. Table 5.1 gives an overview of the methods, which we describe and compare in the following.

Table 5.1: Technologies that enable plugin free, interactive remote rendering in the browser.

Method	Transport	Display
Image-based methods		
JPEG, PNG	WebSocket	*img* element
Motion JPEG	HTTP	*img* element
S3TC	WebSocket	WebGL with S3TC
Video-based methods		
WebRTC	RTCPeerConnection	*video* element
NaCl	WebSocket, TCP, UDP	OpenGL ES

5.3.1 Image-based Methods

The first approach transfers JPEG or PNG images to the client via a a WS connection. To reach legacy browsers that do not support WS, a HTTP fallback using *XMLHttpRequest* and long-polling is possible, but we do not consider this option here. Today, the WS API is widespread available in browsers, and HTTP requests have significant disadvantages as described in the previous section. We therefore set WS support as our minimum requirement.

Any image format natively supported by the browser is suitable. The client translates a received image to a JS image object, either by using the *createObjectURL* function or by *base64* encoding the binary data. We do not recommend to perform the *base64* encoding on the server as it increases the network traffic. The browser takes care of the decoding, thus avoiding the usage of slower JS image decoders. The client displays the image with an *img* element or draws the raw pixels into a HTML5 *canvas* element.

The main advantage of the method is its simplicity and almost ubiquitous availability. Even if a browser supports no other method, the client can almost certainly fall back to this approach. However, in best-effort networks, we want to be able to switch to a video-based solution that is more bandwidth-efficient and may also absorb packet loss.

The second approach utilizes MJPEG, which has plugin free support in Safari, Chrome, and Firefox, including Chrome and Firefox for Android and Safari on iOS. The web page contains an *img* element with the *src* attribute pointing to the rendering server. The browser then takes care of establishing a persistent HTTP connection and receiving and displaying the images. The server sends each JPEG as part of a HTTP multipart message.

While not as ubiquitously available as WS, the advantage of MJPEG is the bypassing of WS and JS to receive and display the images. The browser performs these steps natively and can apply any optimization it sees fit. Also, no *base64* encoding is required.

The third approach utilizes WebGL capable browsers that support the S3TC [HIN99] extension. S3TC is a lossy block-compression technique that achieves a fixed compression ratio of 6:1 for 24-bit RGB images. S3TC enables fast parallel en- and decoding. Via the WebGL extension, decoding on the GPU is supported. The client can thus upload compressed images as is to the GPU for display.

S3TC cannot achieve the compression ratio of the other methods. However, in scenarios where the necessary bandwidth is known to be available and possible visual artifacts attributed to S3TC (mostly visible when encoding sharp edges and gradients) are acceptable, S3TC's fast en- and especially client-side decoding performance make it a feasible approach to achieve high frame rates and minimal latency. There are similar texture compression methods like PVRTC [Fen03] and ETC [Str08]. We choose S3TC as it is the only method with broad support across browsers, including Chrome and Firefox for Android.

5.3.2 WebRTC

The goal of WebRTC is to enable real-time video and audio applications to run within the browser, only using HTML5 and JS. The WebRTC specification [W3C16c] is in the process of standardization. While the central use case is browser to browser communication in a peer-to-peer fashion, the native WebRTC framework is open source and allows the integration of the technology into other applications. Here, this means enabling our native rendering server to stream WebRTC video. The client uses the *RTCPeerConnection* JS API to setup the connection. The browser takes care of displaying the video in the HTML5 *video* element.

5.3.2.1 Architecture

Figure 5.1 shows the components of WebRTC.

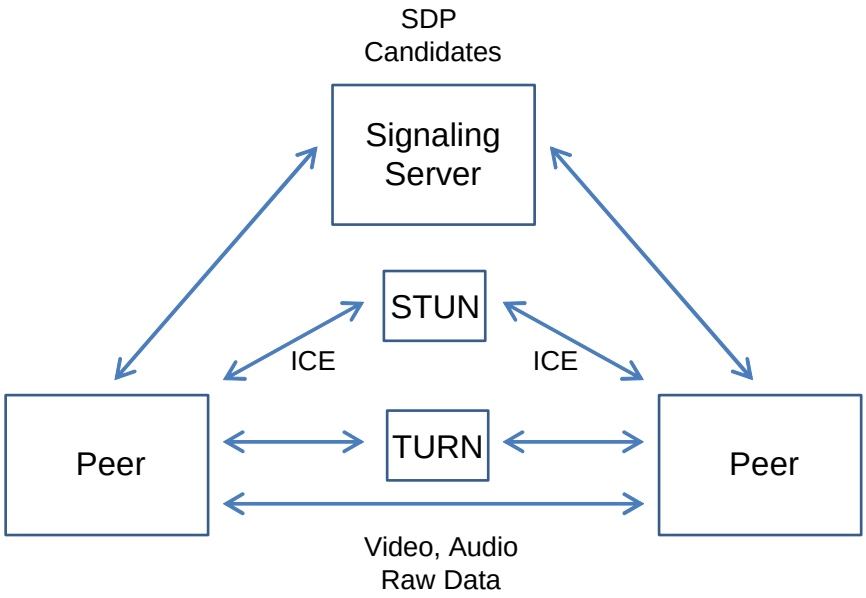


Figure 5.1: The WebRTC architecture.

To enable the exchange of video and audio data between two endpoints, first a session has to be established. The setup involves negotiating session parameters using the Session Description Protocol (SDP) [Han06] and an offer/answer scheme (for example finding video settings supported on both sides). The initiating peer communicates its desired parameters to the other side, which determines an answer according to its own capabilities. The setup includes finding reachable address and port candidates for the endpoints, which may be behind a NAT router or a firewall. Each peer gathers candidates with the Interactive Connectivity Establishment (ICE) [Ros10] technique using STUN [Ros08] servers. If the candidate exchange does yield a direct connection, ICE falls back to TURN [Mah10] servers that relay the traffic.

How session description and candidates are signaled to the other side is not specified. Applications may use any suitable channel of communication. We consider *XMLHttpRequest* support as the minimum requirement to implement the signaling. Usually, the service provider deploys a publicly reachable server to forward signaling messages between the endpoints.

After the setup, the media transport can begin. In addition, the endpoints can exchange arbitrary messages using a data channel. Media and data channels exist independently. WebRTC enforces the encryption of all traffic.

5.3.2.2 Discussion

We decided to use WebRTC video streaming for our remote visualization for the following reasons: WebRTC is an upcoming standard that allows plugin free browser clients and thus the non-invasive integration into other HTML5-only technologies. Second, the video streaming has been specifically designed for real-time purposes. Third, there is already wide browser support with Chrome, Firefox, Opera, Edge, and Safari, including Chrome and Firefox for Android and Opera Mobile. In addition, WebRTC is the technology of choice if confidential transfer of the generated images is required.

While Internet Explorer does not offer native WebRTC support, plugins exist that enable the functionality. Apple finally supports WebRTC with Safari 11. While there was a competing proposal from Microsoft with CU-RTC-Web, Microsoft moved on and now supports the ORTC API in their main browser Edge. ORTC [W3C16b] removes the dependency on the SDP-based offer/answer session negotiation, which generated a lot of skepticism in the community [Ray13], and instead encapsulates the RTC functionality in configurable JS objects.

The JS objects provide more control over the real-time communication than WebRTC. But ORTC also allows to layer a WebRTC compatible API on top of the ORTC API, which benefits developers that are familiar with WebRTC. Edge supports the WebRTC API. Extensions of the WebRTC specification already incorporate many of the ORTC JS objects, and thus ORTC can be seen as the future direction of WebRTC.

The WebRTC API has been designed for easy access by web developers. As of now, the API only provides a high-level access to the underlying video streaming and encoding parameters, which complicates tweaking towards a special use case. This chapter demonstrates that applications beyond communication exist.

There is a debate whether to prefer H.264 over VP8 as the main WebRTC video codec [Bom13]. Levent-Levi [Lev16] lists several arguments for H.264. One argument is the widespread hardware support of H.264, especially on mobile devices where CPU decoding of high-resolution video can be a bottleneck and drain battery life quickly. Google pushes VP8 and also the successor VP9 as the codec of choice. However, the interoperability between the major browsers is currently only given with H.264. While Chrome, Firefox, Opera, and Edge support H.264 and VP8, Safari so far only offers H.264.

This chapter demonstrates the feasibility of WebRTC for remote rendering in the browser. WebRTC development is ongoing, and we expect improvements in the future, such as extended hardware acceleration support for VP8, additional API features, or the upgrade to modern video formats like H.265 and VP9.

5.3.3 NaCl

While JS video decoders exist, they still offer less flexibility, features, and performance than native implementations. We therefore instead focused on the NaCl technology built into Chrome to implement the video receiver. While NaCl is now deprecated and to be replaced with WebAssembly, it provides a video decoding API that has no equivalent in WebAssembly yet.

NaCl enables application developers to run native code safely within the sandbox environment of the Chrome browser. NaCl automatically validates code to adhere to the security requirements (for example direct OS system calls are prohibited) and thus requires no user dialog to obtain permission. Similar to a plugin, a NaCl module can be embedded into a web page, optionally covering a visible area. Web page and module can communicate using a simple API. Within the module, the Pepper Plugin API (PPAPI) provides standard functionality, including networking and restricted access to local storage. In addition, many third-party libraries have been ported to run under NaCl and are made available via *webports* [Goo16]. NaCl has been developed as the replacement for the legacy NPAPI plugins, which have full access to the OS and require user installation due to their potential impact on the security and stability of the host system.

NaCl enables to implement a client that can receive and display video streams in various formats. The rendering server can transfer the stream over WS or over raw TCP and UDP sockets. *FFmpeg* [aut17a] and *Libav* [aut17b] are available in *webports*, and the client can use these libraries to decode the video. Further, the PPAPI includes its own video decoder. The client can display frames with the PPAPI *Graphics2D* class or with an *Graphics3D* context and OpenGL ES 2.0.

The main advantage of NaCl over WebRTC is the low-level control of the video stream. Within the proprietary NaCl solution, fine-tuning for a specific use case or a closed scenario with known network conditions is possible. This includes flexibility in the video codec to use (as long as it is real-time capable and can be ported to NaCl) and the option to avoid encryption, which is mandatory in WebRTC. With the high-level WebRTC API, adaptability is limited, especially on the client side.

NaCl has two types of modules. The traditional NaCl modules are portable across operating systems but are architecture-dependent. The browser must decide at run-time which executable to load for the given architecture. Applications that contain NaCl modules must be distributed over the Chrome Web Store (CWS) and thus require user installation.

With the advent of PNaCl [Don10], applications can also be published on the open web without requiring installation. PNaCl modules are OS- and architecture-independent. The browser translates a PNaCl module at run-time to a hardware-specific executable. While PNaCl is

truly portable, the traditional NaCl provides some additional features like architecture-specific inline assembly and SIMD instructions. However, the goal was to eventually provide portable replacements for most of these features in PNaCl.

NaCl can close in on a native application's performance, but there is still a feature gap to NPAPI. The PPAPI does not expose the rich functionality that native libraries offer. A NPAPI plugin can utilize OS-specific libraries and optimizations that do not run in NaCl due to the sandbox's restrictions. However, we expect the browser platform in general to become more feature-rich and close the gap to native applications.

NaCl is only available in the desktop version of the Chrome browser. It is no HTML5 compatible standard technology with widespread support like WebRTC. However, given the large user base of Chrome and the flexibility that NaCl provides on the development side, we decided to incorporate the technology in addition to WebRTC to enable video-based remote rendering. We could also implement the image-based methods in NaCl. However, we deem this unnecessary. Standardized browser functionality, which is more widespread available than NaCl and close to HTML5, is adequate to implement the image-based approaches.

5.4 Implementation

We have presented a classification and description of the technologies that enable a browser client to receive images and video generated by a rendering server in real-time and without a plugin. In this section, we present an interactive remote visualization system that unifies the technologies.

By employing several technologies, we expand the support across browsers and thus devices. Our client is able to run in a wide range of browsers including mobile ones. Further, the ability to select from several methods increases the adaptability to different conditions and requirements. The switch to a video streaming solution is feasible and may even be required to maintain interactive frame rates if limited bandwidth and packet loss become the bottleneck. Bandwidth efficiency is especially relevant if multiple clients connect to a server. We are able to deploy the system under real-world conditions where network restrictions can occur, like in the Internet or in wireless networks. But we can also tune the system in closed scenarios, for example by switching to fast S3TC en- and decoding in dedicated networks to keep up with a desired high frame rate. Using WebRTC, we can provide a secured connection.

5.4.1 Rendering System

The server uses the rendering library introduced in Section 2.4.4 to implement its rendering functionality. The API is therefore generic and enables to integrate different renderers.

Like in Chapter 4, the core concept is a multi-resolution representation of data sets. A data set consists of one or more quality levels (QLs), which the server can render independently from each other. The QLs are totally ordered, and detail increases with the level number. At least the first level should render fast enough to provide interactive frame rates. The server progressively refines the view with consecutive QLs once user interaction stops.

Any renderer that can map its data sets to QLs can be plugged in. A renderer that does not require QLs, for example a common rasterizer, can still plug in by advertising only a single QL to the API. The renderer must be interruptible during view refinement to maintain responsiveness. We make no assumptions about the rendering time of QLs apart from the first QL.

We have tested the system with the two renderers described in Section 4.3.2. Tuvok [FK10] provides a hierarchical, LOD-based volume renderer, which maps directly to QLs. Secondly, we integrated a proprietary geometry renderer based on progressive meshes. Figure 5.2 and 5.3 show the renderers in action.

Figure 5.2: The *Visible Human* volume data set [U.S12] with four QLs, rendered with Tuvok (QL1 (left), QL4 with 512x512x1884 8-bit voxels).

Figure 5.3: The *Thai Statue* mesh data set [Sta11] with six QLs (QL6 with 10 million triangles shown).

5.4.2 Handshake

Before the visualization session begins, the client needs to determine which of the remote rendering methods depicted in Table 5.1 it supports and then provide the user with an option to select the method for the session. Checking for support includes verifying the existence of

the JS objects *WebSocket* and *RTCPeerConnection*, WebGL with the S3TC extension, and the NaCl mime types *application/x-nacl* and *application/x-pnacl*. Chrome and Firefox still prefix some parts of the WebRTC JS API until the standardization progresses. Therefore, we use the polyfill wrapper adapter.js [Web17] to access the WebRTC interface. We call the connection to transfer images or video from the server to the client the display channel.

Further, the client establishes another connection, which we call the synchronization (sync) channel. The sync channel is a reliable connection used to perform the handshake that initializes the rendering session. The client sends information such as the selected remote rendering method, the resolution and data set for rendering, and also the preferred streaming frequency and bit rate for the video. The server sets up its end of the connection according to what the client chooses. Connection setup with WebRTC is a special case as it involves a signaling stage before establishing the sync channel, which we detail in Section 5.4.5.

After the handshake, the client sends user interaction events to the server over the sync channel. Most importantly, the user can modify the data set pose via transformations, but other parameters like the resolution, the selected data set, and a transfer function in case of the volume renderer also fit in here.

Figure 5.4 gives an overview of the sync and display channel types our system supports for the different remote rendering methods. Even though raw TCP in NaCl avoids the overhead attributed to the WS protocol, we prefer WS by default. The WS overhead is not crucial. More importantly, raw sockets require the user to install a NaCl application from the CWS, which we want to avoid. The WebRTC data channel is currently the only way to access UDP from JS in the open web. Since the data channel also supports reliable transport, it is suitable to implement the sync channel. But we allow switching to a WS sync channel to bypass the data channel encryption. In the future, we might add encryption support to the image-based methods through the data channel. However, for MJPEG, we can only control the sync channel as the browser establishes the HTTP display channel internally.

To conclude the session setup, the client creates the HTML element for display according to the selected remote rendering method and establishes the display channel to receive the image or video data. For example, the client embeds a PNaCl module in the page where the H.264 video is to be shown.

5.4.3 Synchronization

The client captures mouse, keyboard, and touch events with JS and requests the rendering of the data set for a new transformation over the sync channel. After the server rendered the first QL of

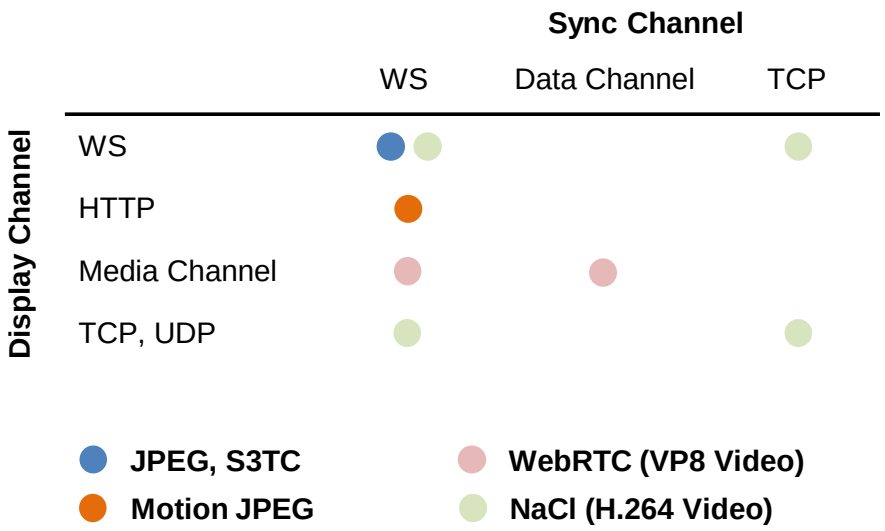


Figure 5.4: Supported synchronization and display channel combinations for the remote rendering methods.

a frame, it waits a small time interval before rendering the consequent QLs for view refinement. If the client issues a new transformation within the interval, the server can immediately start the next frame without having to interrupt ongoing rendering. Rendering abort can cause noticeable overhead depending on how fine-grained the renderer-specific interruption mechanism is.

The synchronization of user input can operate in two modes. First, the client does not send the next transformation before having displayed the first QL for the previous transformation. This sequential mode enables to measure the achievable frame rate and response time under consideration of all factors, which includes the rendering but also the en-/decoding as well as the network transfer and latency. However, the server is idle waiting for the next transformation after sending a completed QL to the client. The idle time is at least the sum of network round-trip, transfer, client-side decoding, and display time. Vice versa, the client is idle while network and renderer are busy.

In contrast, the non-sequential mode decouples sync and display channel and enables parallel utilization of server, client, and network. The client sends transformations with a fixed frequency (30 per second by default) while rendering results arrive asynchronously on the display channel. The user perceives a constant response delay, but the frame rate can stay smooth independent of the network latency. This is true if server, network, and client are capable to reflect the frame rate that the update frequency dictates.

However, if the update frequency is higher than the achievable frame rate, transformations are either lost or have to be queued at the point of the bottleneck. If rendering is the bottleneck (which may be view-dependent, for example falling short for close data set views in a pixel

shading heavy volume renderer), the server unnecessarily receives transformations while still rendering the first QL for a previous transformation. The server must consequently either drop transformations except the most recent one or queue the transformations. Though, losing transformations sporadically is not crucial and barely noticeable. We prefer this approach over queuing. Queuing results in the display becoming decoupled from the user input if the server is a noticeable amount of frames behind, which is the worse side effect than gaps occurring after skipping updates.

The network link is the bottleneck if the transfer of the images is not fast enough due to limited bandwidth or packet loss. The client might not be able to receive, decode, and display frames at the rate the server sends them. Network and client-side bottlenecks are more crucial as they result in the server wastefully rendering images that are dropped further down the pipeline. In contrast, sending a few bytes of unnecessary transformations is no major overhead.

The update frequency dictates but also limits the frame rate. The renderer is done early if it processes transformations faster than new ones arrive. The capping is reasonable to free resources for other clients.

5.4.4 Remote Visualization using Images

Figure 5.5 shows the architecture of the image-based remote visualization.

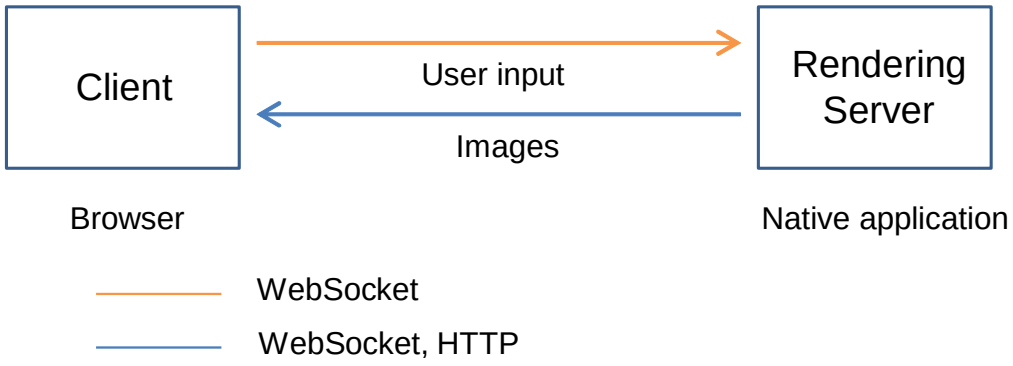


Figure 5.5: Image-based remote visualization architecture.

For JPEG and S3TC, the client establishes both sync and display channel as WS connections. The client receives encoded images, similar to the system outlined by Wessels et al. [WPJR11]. However, our server sends images directly as WS binary frames and thus bypasses the *base64* encoding. Browsers now widely support WS binary data.

The client uses the browser's *createObjectURL* function to load the JPEG images into a JS image object for decoding and then draws the decoded pixels into a HTML5 *canvas* element.

Using WebGL, the client directly uploads S3TC compressed images to the GPU for decoding and display.

To save en-/decoding time and bandwidth, the server only encodes the frame buffer area that the data set covers. To determine the area, the renderer calculates the data set's screen space bounding box using the Sutherland-Hodgman algorithm [SH74]. The client shows the image inside the canvas with the corresponding offset.

For MJPEG, the display channel is a persistent HTTP connection between the browser internally and the rendering server. The server advertises a multipart message to the browser in a HTTP response of content type *multipart/x-mixed-replace* and then sends the rendered JPEGs as message parts delimited by a boundary. The browser decodes and displays the images automatically.

Firefox triggers the *img* element's *onload* event for each JPEG decoded as part of the multipart stream. The client can thus implement the sequential synchronization mode by waiting for the *onload* event to fire before sending the next transformation. However, Chrome behaves differently and fires the event only once for the first decoded image. Therefore, the client cannot directly support the sequential synchronization mode for MJPEG in Chrome. The client still provides indirect support through a workaround that we describe in Section 5.4.5.1 for WebRTC.

5.4.5 Remote Visualization using WebRTC

Figure 5.6 shows the architecture of the WebRTC-based remote visualization. The server delivers rendering results to the client with low-latency VP8 video streaming. At the point of the implementation, there was no H.264 support for WebRTC in Chrome.

Using the WebRTC JS API, the client creates an offer that contains the session description. The client sends the offer and the ICE candidates to the signaling server via a WS connection. We have implemented the signaling server as a native application with proprietary WS support and as a node.js [TV10] application using the WebSocket-Node module. The signaling server forwards the client messages to the rendering server using a TCP connection. We could also merge signaling and rendering server. The system is not dependent on the presence of a separate signaling server, but the separation reflects the design goals of the WebRTC architecture shown in Figure 5.1. The rendering server utilizes the native WebRTC framework and responds with candidates and an answer according to the client's offer. Since WebRTC is a peer-to-peer approach, the role of rendering server and client is interchangeable during connection setup.

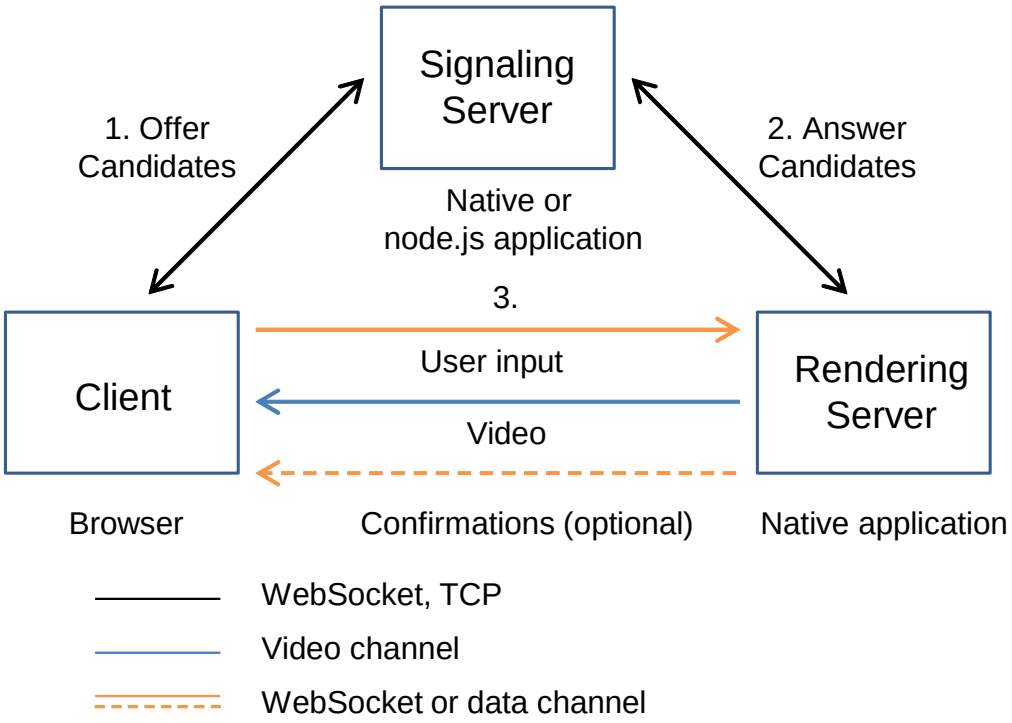


Figure 5.6: WebRTC-based remote visualization architecture.

The server could therefore also send the initial offer. Once the client has received the answer and found a candidate pair to receive the video stream, the display channel is established.

The rendering server should be reachable with a public address. Existing STUN and TURN solutions can be integrated to improve the connectivity in the Internet. We use one of Google's public STUN servers by default.

The client displays the video stream in a HTML5 *video* element. The browser takes care of streaming and decoding internally. WebRTC tries to use UDP for the media transport but can switch to TCP if a UDP connection is not possible, for example due to a firewall that blocks the UDP ports.

5.4.5.1 Synchronization

For remote rendering with WebRTC, the system supports two ways to synchronize user input with the rendering server. First, the client sends transformations to the server with a WS sync channel. Alternatively, the client establishes a reliable WebRTC data channel.

There is no API available that allows the client to determine when a newly rendered image encoded at the server side is reflected in the video. Thus, the client cannot wait for the display of a QL before sending the next transformation. This prevents the direct implementation of

the sequential synchronization mode. In Section 5.4.4, we described the same issue for MJPEG in Chrome.

Therefore, when using the sequential synchronization mode, the server sends a confirmation over the sync channel after the first QL of a frame has been rendered. The client does not send the next transformation before having received the confirmation for the current frame. However, the confirmation messages are not synchronized with the video stream and the display of the QL. As outlined in Section 5.3.2.2, WebRTC currently only provides a high-level API, which can complicate fine grained adaptation to special use cases.

5.4.5.2 Video Streaming

The video streaming runs concurrently to the rendering. The server encodes each image immediately after its generation, which results in a dynamic encoding frequency that is aligned with the rendering output. For static views, the server keeps encoding the last generated image repeatedly for at least a certain amount of time to allow the decoded image quality on the client to progressively improve.

We prefer the dynamic approach to an encoder that operates at a fixed frequency. The immediate encoding enables the client to reflect each image as fast as possible. Further, the rendering frame rate may be initially unknown and subject to fluctuation, which can result in misalignment with a fixed encoding frequency. If rendering is fast, the renderer may wastefully produce several images between encoding passes. Vice versa, if rendering is slow, the server may encode the last image repeatedly while the renderer is busy with the next image. Performing encoding passes during rendering can affect the image completion time, especially if both renderer and encoder heavily rely on the same resources. Therefore, the server only encodes images once during interactive rendering.

5.4.6 Remote Visualization using NaCl

Figure 5.7 shows the architecture of the NaCl-based remote visualization. The server delivers rendering results to the client with low-latency H.264 video streaming. Our client-side implementation compiles with both traditional NaCl and PNaCl.

The client places a NaCl or PNaCl module in its HTML code. The module implements the display channel to receive the video stream. We support WS, TCP, and UDP for the transport. However, Chrome only permits raw socket communication if the web page is packaged as a CWS app with special permissions. Such an app requires a user dialog before execution like a

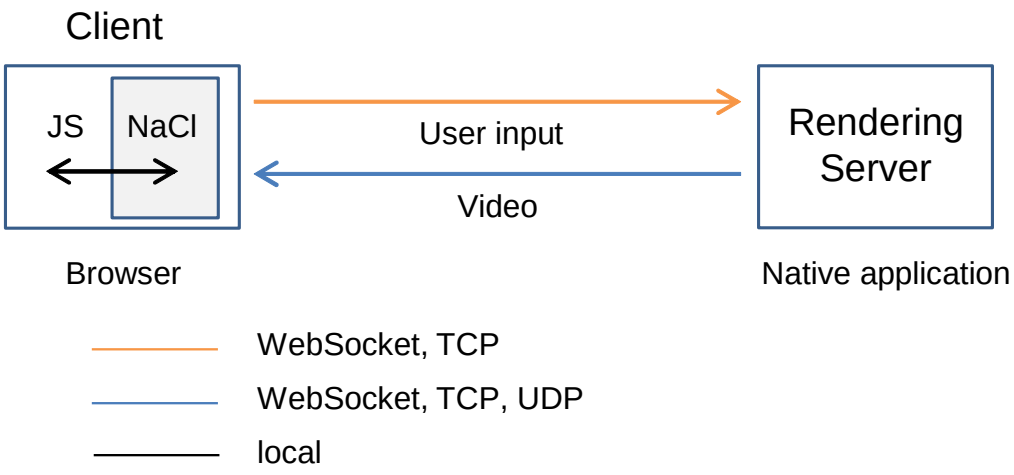


Figure 5.7: NaCl-based remote visualization architecture.

plugin. We primarily target PNaCl and the open web as the execution environment. Therefore, WS is the default transport mode.

The module displays decoded frames using OpenGL ES 2.0. The module also supports display with the PPAPI *Graphics2D* context and the *ReplaceContents* function, which enables to replace the context's back buffer with the decoded image without a copy.

The module's default sync channel implementation uses WS. The web page captures user input and passes handshake information and transformations to NaCl. The module may also handle the user input internally, which avoids the overhead of sending messages from JS to NaCl. Alternatively, the client may implement the sync channel in JS outside of the module. As the communication overhead is minor, we have not added either option yet.

We use *Libav* with enabled *x264* [Vid17] support to en- and decode the video. The client also supports the PPAPI video decoder, which is the preferred option as this decoder utilizes the browser's native capabilities. In contrast, the *webports* version of *Libav* must adhere to NaCl's security requirements, and we were only able to compile the library with disabled assembly optimizations.

The server uses the *fast* preset and the *zerolatency* and *fastdecode* tune options of the *x264* library. Consequently, server and client use multi-threaded, slice-based en- and decoding. The server defaults to a selectable constant quality with an optional bit rate cap but also supports a selectable constant bit rate that optionally tolerates some fluctuation. The server aligns the video streaming frequency with the rendering frame rate as outlined in Section 5.4.5.2. The client can select bit rate and quality in the handshake to account for the expected connection characteristics.

On the dedicated server, we are free to include any hardware-accelerated video encoder to increase the scalability and facilitate the streaming of high resolution content. On the client side, the PPAPI video decoder enables hardware-acceleration.

Using NaCl, we can avoid the encryption that is mandatory with WebRTC. In our use case, confidential transfer of the encoded video frames is less likely a concern than in a communication scenario where audio data is also transmitted.

5.5 Results

This section demonstrates the usage of the remote visualization methods. Table 5.2 shows the test machines. The machines were connected in a LAN, and the network latency is thus negligible. We used Dummynet [CR10] to simulate limited bandwidth. We ran the tests using the sequential synchronization mode, which enabled us to measure the overall latency for each frame independently. The latency is the time from requesting the first QL of a frame for rendering until the display of the QL minus the rendering time. Thus, the latency includes encoding, decoding, and display as well as network transfer and round-trip time (RTT).

Table 5.2: Server and client machines used for the results.

	CPU & Memory	GPU & Network
Server	Intel i7-4770K @ 3.5GHz & 16GB	GeForce GTX 760 & 1 GBit/s
Client	Intel i7-2600K @ 3.4GHz & 16GB	GeForce GTX 680 & 1 GBit/s

We used the Chrome browser on the client machine. Only for MJPEG, we switched to Firefox to enable direct support for the sequential synchronization mode. The rendering resolution is 1280x720. We used Tuvok and the *Visible Human* data set shown in Figure 5.2 for rendering. The video encoder used *x264's* default constant quality mode.

To perform comparable and reproducible runs, the client automatically played back a predefined one minute set of interaction events in each run. The focus is on continuous movement. The latency is crucial when the user interacts, which is the context we compare our methods in. In contrast, for view refinement in static phases, where QLs may take an arbitrary amount of rendering time, a latency difference of even several 100 milliseconds has no considerable impact on the user experience. Therefore, using images encoded with high quality for view refinement is the best option as long as bandwidth is not heavily restricted. We are considering a hybrid

approach that uses video streaming during interactive rendering and image-based encoding during view refinement.

Table 5.3 shows the average measurements without constrained bandwidth.

Table 5.3: Statistics for the remote visualization methods in a network setup with high bandwidth.

	Images			Video	
	JPEG	MJPEG	S3TC	WebRTC	NaCl
Frames per second	103.1	109.8	121.6	40.1	30.5
KBytes/s sent	1469.5	1564.5	13650.1	229.4	99.2
Latency (ms) per frame	3.7	2.9	1.9	15.5	22.6

The image-based methods provide the superior performance. With network limitations being of negligible concern, encoding, decoding, and rendering time dictate the frame rate. Especially the S3TC en- and decoding is extremely fast. We attribute the slight advantage of MJPEG over JPEG to the bypassing of JS and WS to receive and display the images. However, the results also show that the video-based methods are far more bandwidth-efficient.

The interaction set contains several sections where the data set covers only part of the screen with the rest being a regular black background. These sections facilitate fast execution of the pixel-shading bound volume renderer and also fast encoding with small-sized output images. The average frame rate is consequently high. The particularly low latency for the image-based methods is attributed to an encoding optimization that we discuss further below.

At the time we produced the results in Table 5.3 and 5.4, NaCl did not provide an internal video decoder, and we thus used the *Libav* decoder. Due to NaCl's restrictions, we could only use *Libav* with disabled assembly optimizations.

Since a PPAPI video decoder is now available, we performed another set of tests to compare the decoding options. We tested three different decoding setups with a new interaction set and the rasterizer and city scene from Chapter 6. The setups are the PPAPI decoder with and without hardware-acceleration and the *Libav* decoder. Results show that the hardware-accelerated PPAPI decoder is the fastest option. It provided a performance gain of around 144% over PPAPI without hardware-acceleration and 385% over *Libav*.

On the encoding side, our server does currently not support hardware-acceleration for both video streaming methods. Incorporating hardware-acceleration could further reduce the la-

tency of these methods. However, for WebRTC's VP8 codec, hardware-acceleration is not as widespread available as for H.264.

To demonstrate the advantage of the video-based methods in a bandwidth-limited scenario, we simulated a bandwidth reduction to 2 MBit/s. Table 5.4 shows the average measurements.

Table 5.4: Statistics for the remote visualization methods in a network setup with low bandwidth.

	Images			Video	
	JPEG	MJPEG	S3TC	WebRTC	NaCl
Frames per second	12.9	13.7	4	23	20.5
KBytes/s sent	151	157.7	215.4	142.1	65.5
Latency (ms) per frame	53.7	50.2	231.5	33.9	36.2

The image-based methods begin to break, especially S3TC, which requires the most bandwidth. The performance loss is substantial compared to Table 5.3. In contrast, the video-based methods can uphold a frame rate that is much closer to what we observed under unrestricted conditions. NaCl catches up to the performance of WebRTC as WebRTC uses more bandwidth per frame, and the bandwidth is the bottleneck. The decoding performance in NaCl is thus less of a factor.

For the image-based methods, the server optimizes the encoding based on the screen space bounding box of the data set as described in Section 5.4.4. The optimization is of great benefit for the interaction set, where the data set often covers only a fraction of the screen. To illustrate the effect, we repeated the run for MJPEG without bandwidth limitation but disabled the optimization this time. We measured 59.3 frames per second, 1547.9 KBytes/s sent, and a latency of 8.3 ms per frame, which is substantially worse than what Table 5.3 shows. However, given a different data set or sequence of view changes, the optimization might be of less or no benefit. In that case, the advantage of the image- over the video-based methods in Table 5.3 would be less significant while the video-based methods would have a stronger lead in Table 5.4.

5.6 Conclusion and Future Work

This chapter presented a two-fold contribution. We described and compared the state-of-the art of the technologies that enable plugin free, interactive remote rendering in the browser. We then described a visualization system built on top of the technologies, which supports

multi-resolution data sets and allows different renderers to be plugged in. The system unifies the technologies to achieve widespread browser support and adaptivity to different connection conditions and application requirements. We have demonstrated the application of WebRTC beyond communication. To our knowledge, this is the first remote visualization system that utilizes WebRTC.

Figure 5.8 and 5.9 show the remote visualization deployed on desktop and mobile browsers.

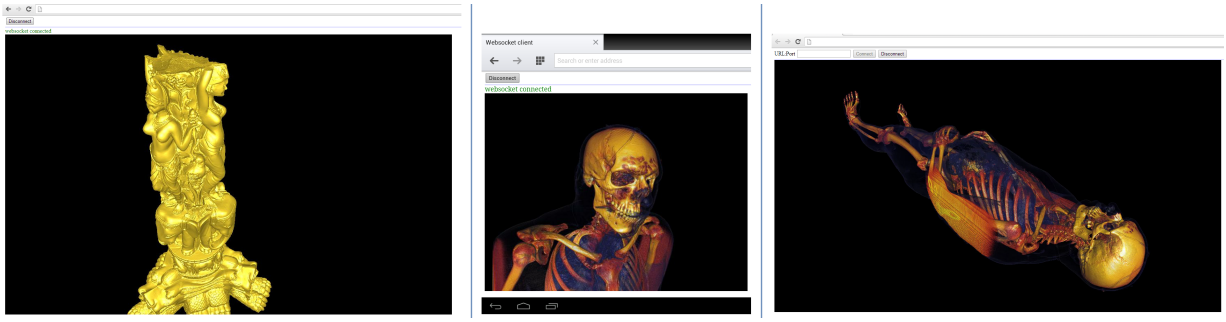


Figure 5.8: The remote visualization system running with MJPEG in Chrome (left, the geometry renderer), JPEG in Opera Mobile (middle, volume rendering), and NaCl in Chrome.

Figure 5.9: The remote visualization system running with S3TC in Firefox for Android (left), WebRTC in Chrome (middle), and WebRTC in Chrome for Android.

While this chapter focused on the visualization of multi-resolution data sets, the remote rendering methods are also interesting for other applications like simulations, games, and virtual worlds. We particularly want to investigate collaborative applications. Since collaboration may require shared views, video streaming with UDP multicast could be an interesting option. However, there is a lack of multicast support in WebRTC, which makes the method less suitable for the use case. NaCl supports UDP multicast, but Chrome restricts network communication to WS in the open web. Also, since NaCl development has been discontinued, we will instead investigate WebAssembly for future implementations.

The conditions of the network affect remote rendering. Bandwidth, latency, and reliability can impact the user experience, especially in best-effort networks like the Internet. Further, providing the server scalability for multiple clients may require substantial investments in hardware.

Today, even thin mobile devices may have a considerable amount of computing capability. A purely server-based approach leaves the client-side rendering resources idle. Therefore, we consider the incorporation of hybrid rendering to allow server and client to cooperate. Client-side rendering can bypass network issues and reduce server load. Our hybrid rendering method from Chapter 4 is particularly suitable as it also builds on QLs and does not necessarily require persistent storage at the client side.

Currently, the user selects a fixed remote rendering method at the beginning of a visualization session. To enable automatic switching to the best suitable method, we consider to monitor the connection characteristics at run-time. If bandwidth is the bottleneck, switching from image to video transfer is the most prominent example. Though, the establishment of a new display channel and encoding pipeline might cause a noticeable slow down. A solution could be to already setup several methods in the beginning, which would increase the initial start-up time but facilitate seamless switching between methods.

In the non-sequential synchronization mode, the client could support automatically adapting the transformation update frequency. The client could decrease the frequency if server or network cannot keep up but also increase it if the system can maintain higher frame rates.

Chapter 6

Distributed Real-time Ray-Tracing for Declarative 3D in the Browser

6.1 Introduction

Modern browsers continuously expand the functionality they provide and thus establish themselves as a platform for a wide range of applications. The tendency is further reflected in the restriction and ultimately removal of plugin-based approaches in recent and upcoming browser versions. These plugins are a stability risk as they run with full privileges on the client system and may contain platform- and OS-specific code. They therefore require a user dialog for installation. In contrast, an application within the browser's bounds enables cross-platform development and user access via a standard web page on any capable device.

One application area in the browser is interactive graphics. The widely adopted WebGL enables the development of GPU-accelerated 3D applications within a web page. However, WebGL is a low-level API. While higher-level libraries like three.js [thr17] exist, they are separate from HTML5 and the Document Object Model (DOM), apart from the integration with the HTML5 *canvas* element for display. Therefore, graphics- or library-specific programming knowledge is required to develop proprietary WebGL applications.

To make graphics content creation more accessible for the web developer, approaches for the declarative description of 3D scenes, tightly coupled with the web page, have been developed [BEJZ09, SKR+10, A-F17]. Especially XML3D has been designed as a HTML5 extension and utilizes the DOM directly for the scene hierarchy and manipulation.

An aspect not addressed by above WebGL-based libraries is server-based rendering. Not every device may have the capabilities to store and interactively render a scene at the desired frame

rate and resolution. Specific to the browser environment, persistent storage for large binary data is restricted and only a subset of the OpenGL features is available in WebGL. JavaScript (JS) provides reduced features and performance compared to native code. Moving the rendering workload to a dedicated server back-end can overcome these limitations.

In this chapter, we present the extension of XML3D to support server-based rendering. We decided to use XML3D due to its HTML5-embedded, generic approach for 3D content creation accessible for the common web developer. Also, XML3D is an established framework, and there is recent work to further improve declarative 3D and XML3D [LSSS16]. This allows us to make the server back-end available to a range of upcoming and already existing applications.

The server back-end provides a rasterizer as well as a real-time ray-tracer, which supports additional features and material properties. Real-time ray-tracing has been a topic of research for over a decade [PMS+99, WS01]. Being an embarrassingly parallel problem, the key for high performance is the careful utilization of parallelism on modern computing architectures. Still, rendering diffuse effects like ambient occlusion and area lights in real-time at high resolution is barely possible on a single commodity machine. For this, running a ray-tracer distributed on multiple machines is required.

Our server back-end can operate in a distributed fashion, supporting an arbitrary hierarchy of servers in a standard or InfiniBand network. The load balancing process determines how well a ray-tracer scales on such an architecture. Ray-tracing workload can be highly heterogeneous. Some areas in an image may be more expensive to compute than others, which depends on the number of intersection tests required to find hit points, the properties of the hit materials, and the amount of secondary rays being cast. To achieve linear scalability with more workers, load balancing aims to keep all workers occupied until the image generation concludes.

We present a load balancing approach that exploits frame-to-frame coherence in a real-time scenario. Based on cost measurements for the previous frame, the load balancer can achieve an accurate balance for the next frame with negligible overhead. During the rendering of a frame, there is no coordination between the master node, which accumulates the final image, and the rendering nodes and also no communication among the rendering nodes. The approach is thus especially suitable if the connection between some nodes is a bottleneck or not possible.

The following section outlines related work for declarative 3D and server-based rendering in the browser as well as for real-time ray-tracing and load balancing. The next section focuses on the client-side extension of XML3D to support server-based rendering. We then describe the server back-end and the load balancing method for distributed ray-tracing. The results section provides measurements and an analysis of the distributed ray-tracer.

6.2 Related Work

6.2.1 Declarative 3D in the Web and Server-based Rendering

There are three initiatives to embed declarative 3D content in a web page and thus make 3D applications accessible for the web developer without requiring domain-specific or graphics programming knowledge. X3DOM [BEJZ09, JRS+13] utilizes the XML-based X3D format to describe 3D content within a web page. In contrast, XML3D [SKR+10, KSSS14] is an extension of HTML5. A XML3D scene is part of the DOM and can be manipulated using the existing JS API that developers are accustomed to. A-Frame [A-F17] builds on top of three.js to allow creating virtual reality applications in the browser using a HTML-embedded scene description. All approaches are currently implemented as a polyfill JS library with an internal WebGL renderer. The original XML3D was implemented as a browser modification and provided a client-side ray-tracer [GS08] for high-quality rendering.

Chapter 5 describes the state-of-the-art methods for plugin free server-based rendering in the browser and presents a visualization system that consolidates the methods. The existing server-based rendering solutions are domain-specific and require proprietary libraries to operate from a web page. Further, most solutions have no support for a distributed server back-end to facilitate high quality and performance rendering. In contrast, we enable server-based rendering in the declarative 3D library XML3D, which allows to specify generic 3D content in HTML5. Custom application logic can be built on top of XML3D with JS. Using this approach, we make the distributed rendering back-end available to a wide array of potential applications.

6.2.2 Real-time Ray-Tracing and Load Balancing

With the advances in parallel computing architectures, real-time ray-tracing is a topic of increasing interest. Today, even commodity multi-processor machines provide a high level of parallelism. Since the focus in this chapter is on the load balancing, we give only a brief overview of real-time ray-tracing.

Parker et al. [PMS+99] describe an early interactive ray-tracer. Wald et al. [WS01] thoroughly outline the research area and its challenges and present their own distributed ray-tracer. Georgiev et al. [GS08] describe a generic, template-based interactive ray-tracing framework. OptiX [PBD+10], a ray-tracing framework running on the GPU, enables a range of applications including real-time usage. On the CPU side, a collection of optimized kernels has been made available with Embree [WWB+14].

Load balancing is the process to distribute the potentially heterogeneous ray-tracing tasks to the processing units, with the goal to achieve maximum utilization. We distinguish between methods that divide the image space and methods that divide the scene among the workers [NFLC12, SY17]. In this chapter, the focus is on image space decomposition. We distinguish between dynamic and static load balancing [CDR02].

6.2.2.1 Dynamic Load Balancing

A dynamic load balancer assigns initial tasks of potentially varying cost to the workers. When a worker becomes idle, it is assigned still outstanding tasks on demand. The first approach is to manage a central task queue. Workers request new tasks from the queue as they finish their current work [Pla02]. Ize et al. [IBH11] describe an out-of-core ray-tracing system that uses a queue both locally to schedule tasks on threads and globally for the nodes in a cluster. The queue manager described by Wald et al. [WS01] attempts to assign the nodes tasks they have previously rendered, which facilitates good cache locality assuming temporal coherence in interactive ray-tracing.

The second approach is work stealing [BL99]. Workers attempt to steal tasks from others instead of relying on a central queue. This effectively removes the queue manager as a possible communication bottleneck as different node pairs can communicate in parallel. Tzeng et al. [TPO10] use work stealing to assign irregular workload to the GPU, giving ray-tracing as one application. Their variant called "task donation" additionally allows workers to offload tasks to others in case memory is not sufficient to hold the data for the locally outstanding tasks. DeMarle et al. [DGP04] initially assign previously rendered tasks to exploit temporal coherence in their distributed shared memory ray-tracing system. This is crucial to minimize fetching missing data from another node. After the initial assignment, work stealing is used.

Dynamic load balancers are generically applicable to parallel problems and naturally handle heterogeneous computing resources. Freisleben et al. [FHK97] demonstrate the advantage of a queue-based approach over traditional static load balancing variants in a setup with heterogeneous workers. However, dynamic approaches can suffer from communication overhead, which increases with the number of workers. A low-latency connection between the master and the workers and in case of work stealing between all workers is essential.

While a way to hide network latency is to assign several tasks to the workers in advance, a process that has been called task prefetching [WBDS03], this has several drawbacks. When a worker finishes a task and sends the result, it should already have another task available, otherwise the worker would be idle until more tasks arrive from the queue manager or another

worker. However, the amount of work that needs to be prefetched to hide the latency depends on the latency, computing power of each worker, and the time it actually takes to process the tasks, for which the cost is unknown. Further, assigning tasks in advance reduces the granularity of the load balancer. The more work is assigned to workers in advance, the higher the potential for load imbalance, which results in reduced scalability. The results presented by Wald et al. [WBDS03] show that a linear speedup could not be achieved. Therefore, task prefetching is no generic solution, and its benefits are configuration dependent.

6.2.2.2 Static Load Balancing

A static load balancer assigns fixed tasks to the workers before rendering a frame and thus avoids task management and communication overhead during rendering. Our distributed rendering back-end supports any hierarchy of nodes, and a dedicated network setup is not mandatory. We do not assume a fast or any link between the rendering nodes. The node hierarchy can span multiple levels, and thus not even the master is necessarily directly connected to a rendering node. Therefore, we employ static load balancing. However, a static approach cannot react to imbalance by shifting tasks to workers that become idle. Thus, determining a task distribution that accurately equalizes the rendering cost on the workers decides about the effectiveness.

Heirich et al. [HA98] discuss several load balancing strategies for ray-tracing, including a randomized static assignment of pixels among workers. While such highly granular scattering can achieve an even cost distribution, it facilitates bad cache and data locality as each worker operates across the whole image [WPSB03]. Scattering and image space decomposition in general have shown to perform poorly in large-scale out-of-core and especially distributed memory systems if memory access dictates the performance [NFLC12]. Scattering individual pixels does also not fare well with a modern ray-tracer that traces packets of coherent rays.

More recent approaches attempt an estimation of the cost distribution for the next frame. From the cost predicate, the load balancer can derive a partitioning into tasks of equal cost. Rincón-Nigro and Deng [RND13] perform a reduced ray traversal through the bounding volume hierarchy to estimate the cost of ray-tracing tasks. They use the estimate to balance the task distribution in a multi-GPU setup. On each GPU, they use a queue-based scheduling mechanism similar to Aila and Laine [AL09].

Moloney et al. [MWMS07] calculate a per-pixel cost estimate for their direct volume rendering system. Gillibrand et al. [GDC05] propose to time profiling rays at a lower image resolution and then apply the resulting cost map to the full resolution. They tested the approach only with primary rays. Though, producing a representative cost map by profiling can cause major

overhead, especially when considering secondary rays.

Similar to our approach, Cosenza et al. [CCDC+08] assume temporal coherence in a real-time ray-tracing system. They consider timings obtained for the previous frame representative for the next frame. However, along with Gillibrand et al. [GDC05], their approach suffers from inaccuracy as timings are obtained in a lower resolution than the one of the renderer. Each node only measures the rendering cost of each task assigned to it. However, the cost of rays or ray packets within a task may vary, which can lead to unbalanced scheduling decisions. Also, the cost map and thus the precision of the load balancer is irregular as the task sizes may change every frame. Consequently, the system additionally incorporates a task queue to account for possible imbalance.

Cosenza et al. [CDE13] render a rasterized preview on the GPU to approximate the cost map for the next frame. The load balancer then uses a summed area table based tiling algorithm to derive tasks of equal cost from the map. Though, the approximation suffers from substantial inaccuracy, which prevents a scaling similar to a dynamic approach. Therefore, they ultimately propose a work stealer that is optimized through sorting of the initial tasks by the approximated cost.

Our renderer uses packet-tracing [WWB+14] and thus divides the image space into packets of adjacent pixels. Each node obtains a cost map for its task in packet space using high-resolution timings. The load balancer can thus achieve a strong accuracy while still only generating one static task per node and frame, effectively minimizing communication and tiling overhead. In contrast, Cosenza et al. [CCDC+08, CDE13] end up with variants of existing dynamic approaches due to the limitation of their static attempts, eliminating the advantage of not requiring communication during rendering. They do not consider non power of two node counts and heterogeneous nodes. We extend the tiling of Cosenza et al. [CDE13] to support any amount of heterogeneous workers.

6.3 System Architecture

Our goals are to support server-based rendering in XML3D and to provide a distributed rendering back-end suitable for real-time ray-tracing. Figure 6.1 outlines the architecture of our system. From the user's perspective, accessing an application is as simple as opening a standard web page. Embedded into the page is a XML3D scene as well as the application logic on top of XML3D. If feasible and desirable, the user can select the client-side WebGL renderer.

In addition, XML3D may offload rendering to a native server back-end. There are two indepen-

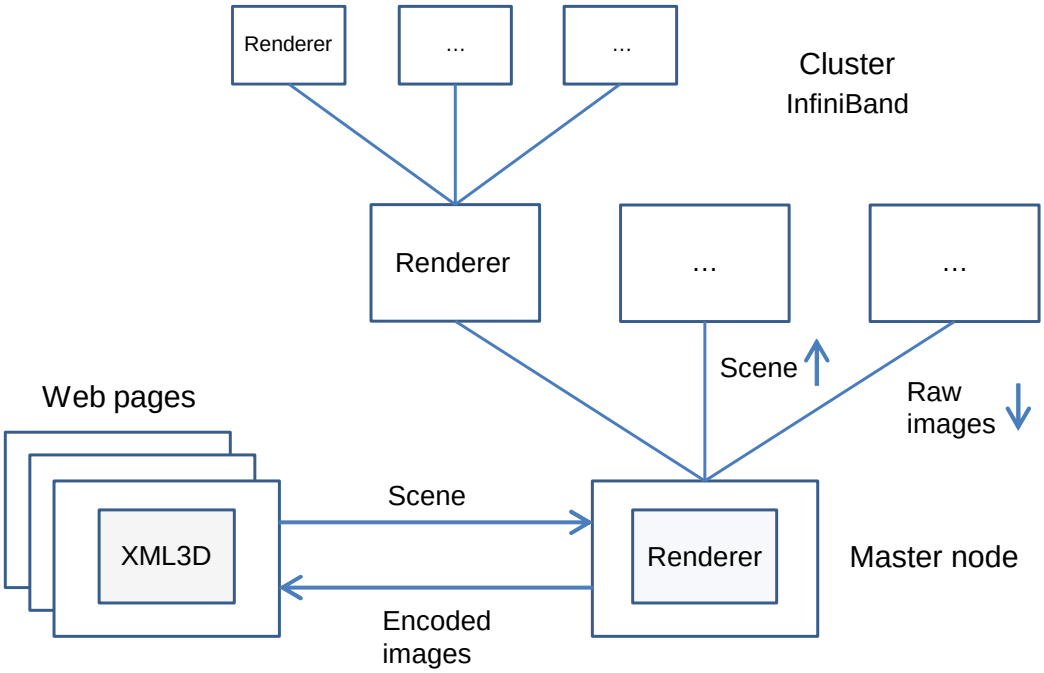


Figure 6.1: Exemplary architecture of the distributed rendering system. XML3D clients connect to a cluster with a hierarchy of rendering nodes connected via InfiniBand.

dent connections between the client and the master node and between each pair of connected nodes in the cluster. The first transfers the scene updates from the client to the master. Each node forwards the data to its child nodes. The second transfers the output from the resident renderer and the child nodes down the pipeline. As there may be bandwidth restrictions between client and master, the master performs an image encoding step. Within the cluster, we support a standard network and also InfiniBand, which can offer higher bandwidth and is thus ideally suitable to transport high-resolution image data. The separate connections enable an asynchronous pipeline where the client already prepares and synchronizes the updates for the next frame while the current frame is still being rendered or transferred to the client.

The cluster network sends raw pixel data to avoid issues with encoding multiple parts of an image separately and later joining them. This can result in an overall reduced compression ratio and decoding performance and most importantly produce visual artifacts in the merged image. Though, these drawbacks do not apply to S3TC, which operates on independent pixel blocks with a fixed compression ratio. We therefore utilize distributed S3TC encoding in Chapter 7.

6.4 Client Side

A web page may contain a statically embedded XML3D scene. The web developer can also build arbitrary application logic on top of XML3D to manipulate the existing scene and add new

elements or the entire scene dynamically. With the server-based rendering extension, we want to keep and utilize this flexibility. We therefore followed a minimally invasive approach that exposes the server-side functionality with a small set of attributes described in Section 6.4.4. The attributes are an optional addition to existing XML3D elements. There are no new elements the developer needs to get accustomed to.

Consequently, existing applications can immediately be used and new ones created as before. The approach enables a hybrid architecture where the client executes the application logic in parallel to the server-side rendering. The server does not need to adopt XML3D-specific features, which avoids maintaining redundant functionality and makes the server easily portable to other clients. In Chapter 7, we describe a native client application that is compatible with the server back-end.

The disadvantage is that the client holds the scene and needs to synchronize resources like buffers and textures with the server. Though, the synchronization procedure is progressive, so rendering can already commence and provide the user with intermediate results quickly while part of the scene is still loading. Server-side caching is a strategy to avoid the synchronization overhead for static resources. However, in XML3D, every resource is potentially dynamic and could require continuous updates. Section 6.9 therefore outlines a different approach for a future version of the system.

6.4.1 Connection Setup

To enable server-based rendering, the *xml3d* HTML element must contain a *server* attribute that points to the address and port of a rendering server. Otherwise, the client-side WebGL renderer processes the scene. The client first establishes a WS connection to synchronize the scene data. This synchronization channel is also used to send an initial handshake to the server. Via the handshake, the client can select the server-side renderer to use.

The handshake also tells the method to encode and transfer the image data. According to the selection, XML3D creates the display channel, which establishes the connection for incoming images and provides the HTML element to display the images in the web page. The client places the display element behind the transparent *canvas* element otherwise used for local rendering. The canvas is still required to capture user input, for example to select objects or move the camera. The architecture is modular and enables to integrate several display channel types. Section 6.4.3 outlines the supported types.

6.4.2 Synchronization

The data to be sent over the synchronization channel includes the resolution, camera, and lights. The data also includes a collection of meshes. A mesh does not store data other than a transformation but references buffers, textures, texture samplers, and material properties. Meshes may share references, enabling the reuse of data (for example for geometry instancing). This separation offers a lot of flexibility to compile meshes. The client can compress buffers with the *Deflate* algorithm and send JPEG and PNG compressed textures.

XML3D loads resources asynchronously and progressively. An event notifying the initialization, change, or deletion of a data entry can be generated anytime. Instead of synchronizing the update immediately with the event, the client schedules the update on the main run loop. This loop runs at a selectable rate to pass outstanding updates to the synchronization channel. If at least one update was sent during an iteration, the client requests the rendering of a new frame. The application logic may trigger update events at a high rate. If multiple updates of the same resource or parameter occur between loop iterations, the client only keeps the most recent version for the synchronization. The scheduling prevents either excessive rendering requests or the sending of redundant updates between requests.

Especially the initial loading of the scene may trigger heavy traffic for geometry and textures. The client therefore implements a rate control by postponing updates if the amount of data in the WS send buffer exceeds a threshold, thus preventing a potential overflow of the buffer.

6.4.3 Display

The client receives rendering results asynchronously to the sending of scene updates. We support several of the plugin free image transfer and display methods from Chapter 5, which allows the application to choose the most appropriate method given the conditions and requirements. In a dedicated network, the focus may be on maximum en- and decoding performance. In a best-effort network, bandwidth efficiency may be of more concern.

The server can transfer JPEG images over WS and MJPEG over HTTP. To trade compression-ratio for more en- and decoding speed, the server also supports S3TC. S3TC enables fast parallel encoding. Using a commonly supported WebGL extension, the client decodes S3TC images in hardware on the GPU. For more bandwidth efficiency, the server supports H.264 video, which the client receives and decodes using NaCl [YSD+09]. Chapter 5 presents measurements regarding latency and bandwidth efficiency for the methods.

In addition to the methods supported in the web client, the server can also interface with DaaS [LPHS12] to enable streaming rendered content to virtually any display.

6.4.4 HTML Integration

We expose server-based rendering to the developer with a set of attributes that can be added to the *xml3d* HTML element. With the exception of the *server* attribute, all attributes are optional.

- **server**: Address and port of a rendering server.
- **renderer**: The renderer to use. The server currently supports a rasterizer and a real-time ray-tracer (defaults to the rasterizer).
- **display**: Method to transfer and display images. The system currently supports JPEG and S3TC via WS, MJPEG via HTTP, and H.264 via NaCl (defaults to JPEG).
- **naclTransfer**: NaCl-specific option that specifies the connection to receive the video, supporting WS, TCP, and UDP (defaults to WS, which is the only connection permitted in the open web).
- **nodes**: The maximum number of nodes to use for distributed rendering (defaults to all nodes). It may be desirable to only use a subset or single node to increase the server back-end's client capacity. A renderer may not require or benefit from several nodes.

Further, we extended XML3D with a set of new material properties to reflect the capability of the server-side ray-tracer. Refraction and reflection coefficients and the refraction index can now be specified for any material.

Figure 6.2 demonstrates the simple changes to port a scene to server-based rendering. By simply removing or renaming the *server* attribute, XML3D falls back to the WebGL renderer, which silently ignores unsupported features.

6.5 Server Side

Figure 6.3 illustrates the main components of the server back-end. Analogous to the client, the server manages a synchronization and a display channel. The display channel is responsible to send the image data to the client and thus interfaces with the renderer. The display channel

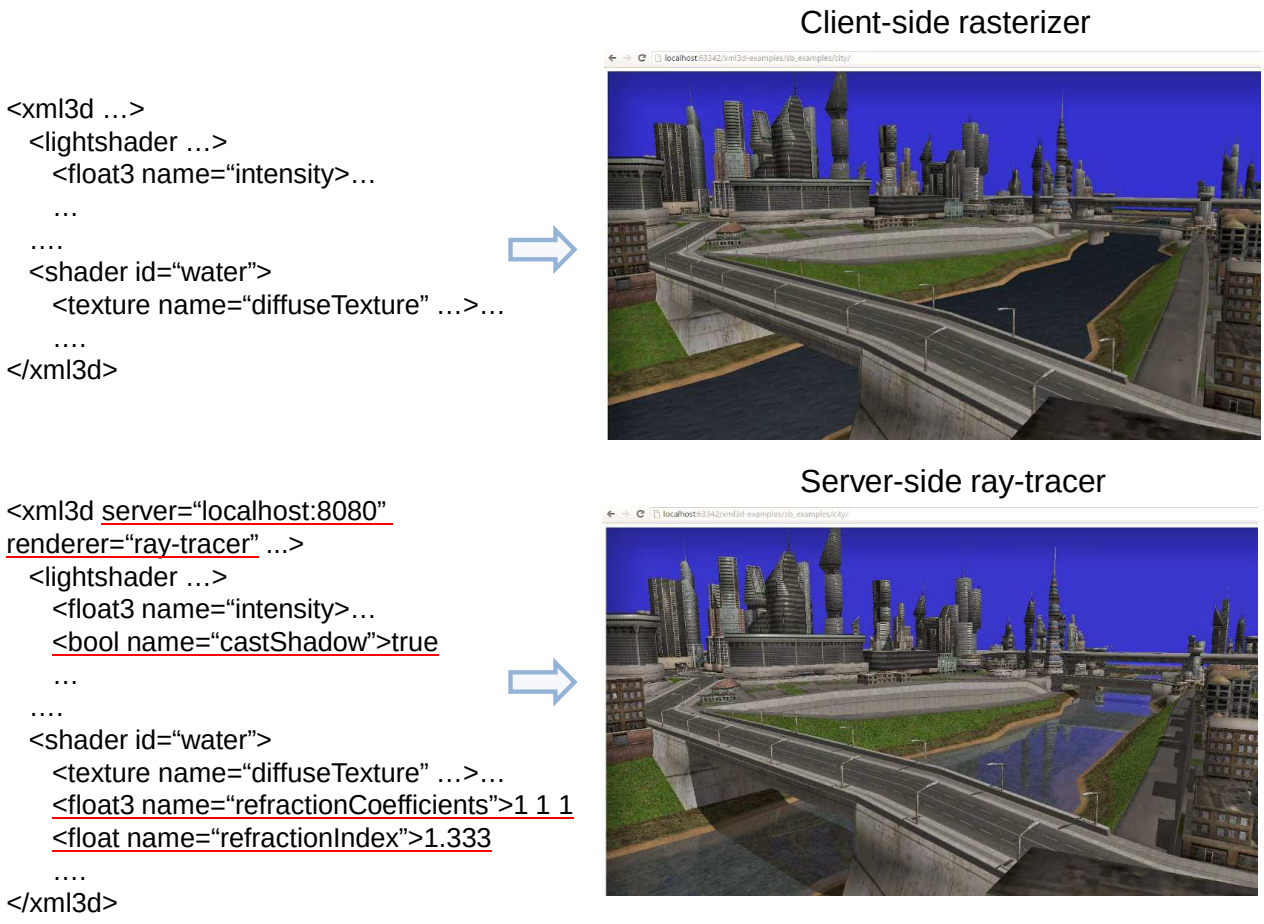


Figure 6.2: The simplified original version of a XML3D scene (top) and the adapted declaration for the server-side ray-tracer (changes highlighted in red).

passes scene updates from the synchronization channel to the renderer or caches the updates if the renderer is still busy with a previous frame. We call the last image receiver the display client. The master encodes the final image to be sent to the display client.

For distributed rendering, synchronization and display channel also act as a client by establishing a connection to each of the participating child nodes. The corresponding handshake requests a dedicated display channel for raw pixel transport in the cluster network. The child nodes receive updates from and send their rendering output to their parents. The master runs a renderer-specific load balancer to determine the rendering tasks for the next frame, which we address in Section 6.5.2. Every node has the same capabilities and can assume the role of the master. This results in a flexible architecture that allows to chain an arbitrary hierarchy of nodes.

The server facilitates asynchronous execution. The components in Figure 6.3 run in separate threads. While the local renderer executes, rendering results from the child nodes may arrive and be forwarded to the parent. Concurrently, scene updates for the next frame may arrive to

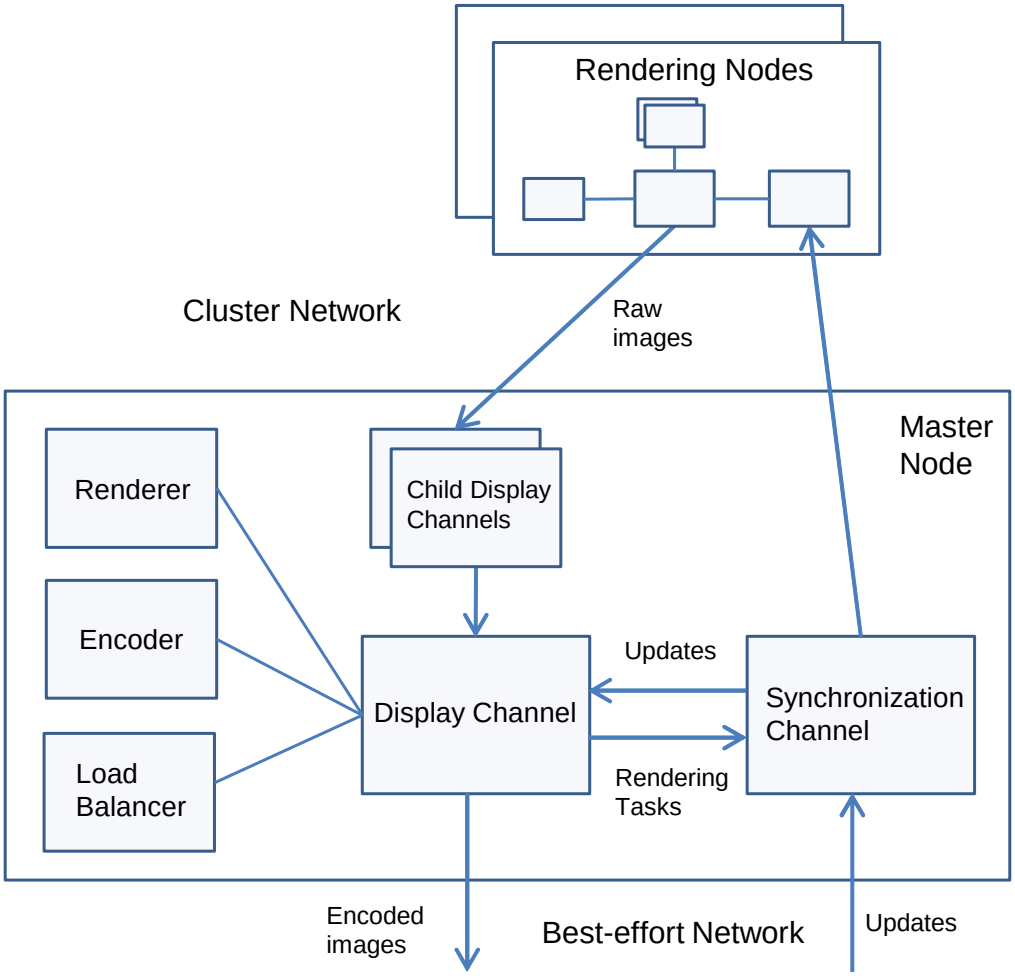


Figure 6.3: The components that run a distributed rendering session in the server back-end.

be forwarded to the child nodes and cached for the local renderer. The server can immediately start the next frame from cached updates, keeping the renderer occupied. The encoder can run in parallel to the rendering of the next frame. The display client prepares the upcoming application state while the rendering back-end is busy. Therefore, we achieve strong parallel utilization of the computing and network resources.

6.5.1 Renderers

The server provides an abstract API that developers can implement to plug in their renderers. So far, we have integrated two renderers. The first is the reference rasterizer, which mimics the functionality of XML3D's WebGL renderer. The second is a custom CPU ray-tracer that we implemented on top of the Embree [WWB+14] ray-tracing kernels. The ray-tracer is optimized for real-time performance. It operates on packets of rays for both tracing and shading, capable to utilize the SSE, AVX, and AVX2 instruction sets. Also supporting 512-bit SIMD instructions

(AVX-512) is straight-forward once the required hardware is avalable to us. To run the renderer locally on multiple cores, we utilize the CilkPlus multi-threading language, which originates from Blumofe et al. [BJK+95], and its internal work stealer.

The ray-tracer supports ambient occlusion, which is a Monte Carlo technique that requires a good amount of sample rays to avoid noisy results. The rendering cost can snowball quickly with more samples, especially considering materials that trigger secondary rays. The feature is barely adequate for real-time performance on a single commodity machine, which motivates the usage of a rendering cluster.

Our ambient occlusion implementation exploits the fact that the hit points for a packet of coherent rays are likely close together on the same mesh. The normals are thus also likely similar. Given a number of random hemispherical rotations, for each rotation, the renderer transforms the normals to generate a set of ambient occlusion rays and then uses packet-tracing to sample the set. The rotations can be pre-computed.

As shown in Chapter 7, we also integrated a global illumination ray-tracer as a third renderer in a new version of the distributed rendering framework.

6.5.2 Distributed Rendering

Nodes that participate in rendering, which may include the master, are called rendering nodes. For each renderer, a rendering node stores a coefficient that indicates the node's performance relative to the other nodes in the cluster. In a cluster of homogenous machines, all nodes have the same coefficient. It is up to the operator to determine coefficients that reflect the heterogeneous nodes in the cluster, for example with benchmark tests. Such tests could be integrated into the framework, which is a topic for future work. Section 6.6.3 describes the coefficient calculation for our ray-tracer.

When the display client connects, the master requests the coefficients of the renderers available in each child's sub-hierarchy. A child node adds its own renderer to the list and further traverses the node tree by requesting the coefficients from its children. Effectively, this process flattens the node hierarchy and the master ends up with the complete list of coefficients without requiring a direct connection to each rendering node. The master then selects the rendering nodes to use in this session, prioritizing stronger nodes if only a subset of the available nodes should take part.

For the communication between two nodes, the server supports TCP over Ethernet and InfiniBand. Ethernet enables the deployment in a commodity setup but may be limited in band-

width. Since the cluster network transports raw pixels, we recommend a bandwidth of at least 10 GBit/s.

To determine the screen space rendering task for each node in the upcoming frame, the master asks a renderer-specific load balancer. The server provides an abstract API for static load balancing, which a renderer implements to benefit from distributed execution. A renderer may generate custom cost data for the current frame, which the node transports to its parent along with the pixel data. The master gathers the cost data from the nodes and passes it to the load balancer to generate the task distribution. The load balancer may consider the renderer coefficients to weigh nodes. To facilitate integrating new renderers quickly, we provide a default load balancer that keeps returning a fixed set of evenly sized tasks. We use the default implementation to distribute the rasterizer.

Concluding, our architecture enables a flexible setup of possibly heterogeneous nodes with different roles. Nodes not suitable for rendering may still contribute as a master that is dedicated for encoding or as a network hub that gives access to a set of rendering nodes otherwise not reachable. Since each node can be the master, the service provider can define sub-clusters that provide different access levels for varying clients. A ray-tracing client may access the whole hierarchy while it may be enough to restrict rasterization to a small branch. Due to the static nature of the load balancing, the rendering nodes do not need to connect with low latency to each other.

6.6 Load Balancing

This section describes the static load balancing for the distributed real-time ray-tracer. The foundation is the observation that in a real-time scenario, view changes between consecutive frames are likely small. Thus, in most cases timings for one frame are representative for the following frame. The concept fits our server back-end and the ray-tracer, which are explicitly designed for real-time operation.

The ray-tracer processes the image space in packets of neighboring pixels. During rendering, the ray-tracer measures the cost to determine the colors for each packet and thus effectively produces a cost map in packet space. The renderer transforms the cost map into a summed area table (SAT), which the display channel transfers to the master in addition to the pixel space image produced for the current task. The SAT allows to determine the cost of any rectangular region in constant time. Once the master has accumulated the SATs from the rendering nodes, the load balancer uses the SATs to determine a tiling into tasks of balanced rendering cost. Figure 6.4 outlines the steps to acquire the task distribution for the upcoming frame.

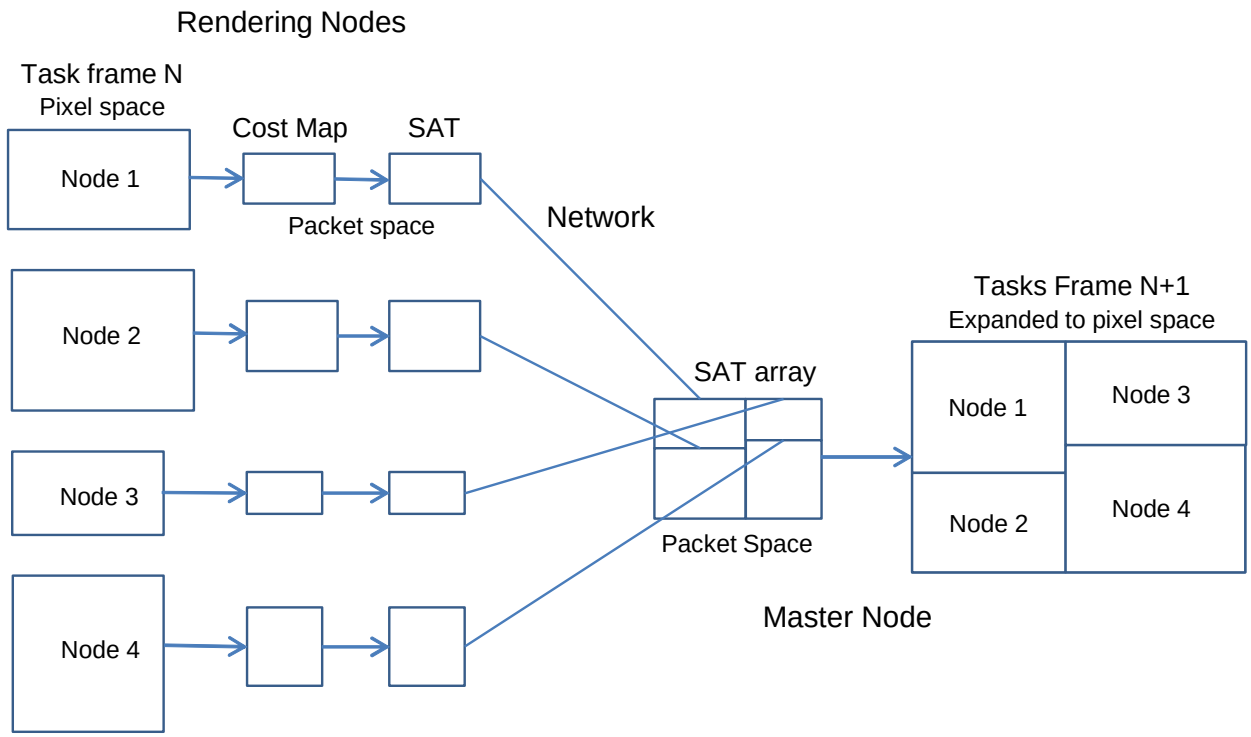


Figure 6.4: The static load balancing for real-time ray-tracing. In this example, the ray-tracer uses SSE with packets of 2x2 pixels. Since the timings are per-packet, the cost map is four times smaller than the rendered image. Each node outputs a SAT to the master, which runs the tiling algorithm on the array of SATs to generate tasks of balanced cost for the next frame.

Due to the high timing granularity in the packet space of the ray-tracer and the frame-to-frame coherence present in the real-time system, the load balancer can achieve a strong accuracy and thus scalability as we demonstrate in Section 6.8. There is no communication during rendering and no communication between the rendering nodes at all, which makes a basic deployment with any setup of machines and network possible.

The load balancer requires each node to generate a cost map during rendering, convert the map to a SAT, and send the SAT to the master. The overhead is constant and depends on the resolution of the image. More nodes effectively reduce the overhead as they process continuously smaller parts of the image in parallel. More cores on a node reduce the cost map generation overhead as the rendering threads acquire timings concurrently. In contrast, the communication overhead of a dynamic load balancer increases with the number of nodes and tasks. While the tiling cost on the master also increases with more nodes, Section 6.8.2 demonstrates that the cost stays insignificant even for many tasks.

As the rendering cost increases, the constant overhead becomes increasingly negligible. In contrast, a dynamic approach may require a finer task granularity in response to a high rendering time of individual tasks to avoid a single worker and task to stall the completion in the end. More tasks in return increase the communication overhead.

6.6.1 Cost Map

The mechanism to measure the cost for the pixel packets must be fine-grained and induce little overhead. We support two techniques that reflect the requirements. The processor time stamp counter (TSC) [Int98] stores the number of clock cycles since its last reset. Since the TSC is a 64-bit value, the chance that a reset disrupts a measurement is extremely low. To acquire values that are consistent across heterogeneous nodes, the ray-tracer divides the cycle count by the maximum core frequency in Kilohertz, assuming that all cores on a node run at this frequency under the ray-tracing load. This essentially yields time in milliseconds.

In our tests, the TSC produced reliable results. Still, the approach may suffer from issues that can reduce the timing accuracy. The TSCs on different cores may not be tightly synchronized. A thread that switches core between two measurements can thus result in distorted values. Also, processors with out-of-order execution support may shift the execution order of instructions, which can cause a slightly misplaced read of the TSC via the *rdtsc* [Int16a] intrinsic. The *rdtscp* intrinsic takes care of serialization but is not as widespread supported in hardware and performed substantially worse. The processor switching its frequency can cause further inconsistencies.

To account for the potential issues with the TSC, the ray-tracer alternatively supports the performance counter provided by the Windows OS. The performance counter usually relies on the TSC internally and thus also provides high precision and speed. It adds logic to handle the TSC issues and can be considered as portable and reliable across recent systems. Mostly on older systems, the performance counter may use a slower and possibly less accurate timing mechanism than the TSC internally.

6.6.2 Summed Area Table Generation

Hensley et al. [HSC+05] describe fast SAT generation on the GPU. However, since our cost map resides on the CPU, we implemented a multi-threaded CPU version that utilizes SIMD instructions and induces minimal overhead. The network already transfers the image data while the node generates its SAT. The master can thus start the encoding of the accumulated image while the SAT array is still incomplete.

Cost map and SAT store a 32-bit value per pixel packet. In a bandwidth setup of 10 GBit/s, the SAT transfer therefore only takes around 0.369 ms for a 720p image divided into packets of eight pixels. Since the SAT sending is distributed among the nodes, the transfer overhead is further mitigated. Also, the master may participate in the rendering, which reduces the

network load.

6.6.3 Tiling

The master stores the SATs from the rendering nodes in the SAT array data structure. The array is the input to the tiling algorithm, which determines the tasks for the next frame. The tiling executes asynchronously to the image encoding, mitigating the already low overhead of the algorithm. The array behaves like a single SAT in the overall packet space resolution and provides the cost for a rectangular image region from the origin to any packet. Several SATs may contribute to the cost, which Figure 6.5 illustrates. The load balancer initially sorts the SATs by their offset on the x-axis, which then allows to quickly reject SATs that start beyond a region of interest using binary search.

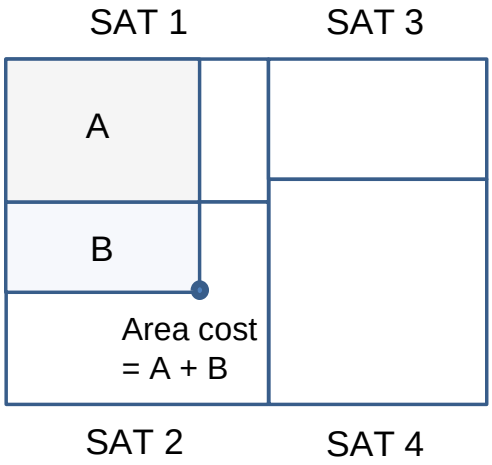


Figure 6.5: Sampling two SATs in the SAT array to obtain the rendering cost for the pixel packets in an area.

The tiling algorithm starts with a packet space resolution tile that all the nodes belong to. Consulting the SAT array, the algorithm uses a binary search to split the tile into two child sides with the cost balanced according to the nodes attributed to each side. The algorithm recursively splits the child tiles, switching the axis on each level, until a leaf tile representing the task for a single node has been reached. For an even count of homogeneous nodes, balancing means finding the split that evens out the cost on both sides. However, the algorithm also considers tiles with an uneven node count and the presence of heterogeneous nodes. The algorithm weighs nodes according to their renderer coefficient.

A timing represents the cost to render a packet of pixels on a single core. But the core performance may vary between heterogeneous nodes. We therefore statically assign a performance coefficient to each node, which indicates the performance increase for a single core of the node

relative to the node with the weakest cores. The load balancer normalizes the cost values retrieved from a SAT by multiplying with the performance coefficient of the originating node. The coefficient is an empirical factor that the cluster operator must choose. If all nodes share the same processor family, we set the coefficients proportional to the nodes' core frequencies.

Each node locally distributes the ray-tracer across the logical cores with a work stealing load balancer. The static load balancer in the cluster thus assumes a linear scaling of the ray-tracing performance to the number of cores on each node. The load balancer therefore calculates a node's renderer coefficient as the product of the number of logical cores and the single-core performance coefficient.

When splitting a tile, the algorithm balances the normalized cost based on the ratio between the sum of renderer coefficients attributed to the first and the second child, thus accounting for any node count and heterogeneous nodes. For an uneven node count, the load balancer assigns the additional node to the child that brings the sums on both sides closest together. This facilitates producing child tiles of similar cost.

The ordering of the nodes in the tile tree is fixed, which causes each node to stick to roughly the same image area. This facilitates good cache locality and thus can improve the rendering performance.

6.7 Applications

Here we introduce several existing applications that utilize the distributed rendering framework and its XML3D client.

Within the SINNDODIUM [Sof15] project, we integrated the technology into two demonstrator applications. The first demonstrator supports the layout planning and stocking of the shelves in the retail industry. The demonstrator provides a web interface that visualizes a real sales floor. The sales floor is an instrumented, sensor-equipped environment that enables to automatically translate changes to products or product information to the virtual world. The virtual camera can control cameras in the real setup to allow a side by side display of both worlds. Since the sales floor may contain thousands of objects, the offloading of the rendering work to the server back-end improved the user experience especially on mobile devices.

The second demonstrator supports the collaborative analysis and maintenance of production plants. A web interface visualizes the production machinery and the associated sensor data. The system integrates video streaming to enable the collaboration of multiple staff via webcam.

The video streaming incorporates the PRRT [GSH12, GGH13] protocol, which enables adaptive error control under specific time constraints and is suitable for real-time applications. The demonstrator currently still uses a NPAPI browser plugin to implement the video receiver. To enable the PRRT-based video streaming also for our distributed rendering solution, we created a new display channel in XML3D that incorporates the plugin. The integration is experimental until a portable version of the receiver exists. The video streaming server is a separate component that runs on the master node. The rendering server passes images to the video server via shared memory.

A third example application allows multiple users to collaboratively roam a virtual city and potentially any other scenery with an avatar. The client interfaces with the FiVES [FIW15] synchronization server to keep the application state synchronized across the connected participants. Figure 6.6 shows the views for two avatars as they explore the city side by side, with one view being generated by the server-side ray-tracer on four nodes and the other by XML3D's client-side renderer.

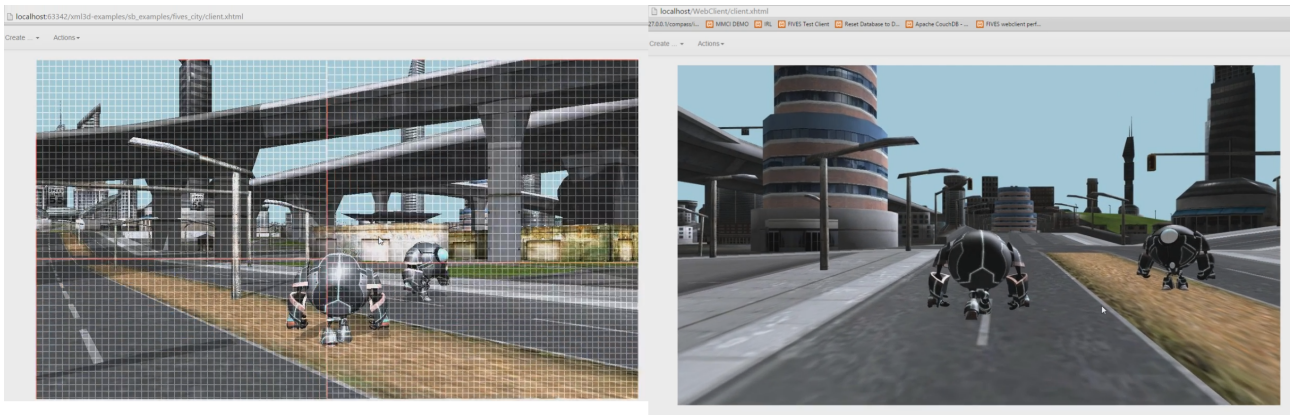


Figure 6.6: Two users explore a shared world in an example application running on top of XML3D and the distributed rendering framework. The server-side ray-tracer generates the left view, while the client-side WebGL rasterizer produces the right one. The tiling visualizes the server-side task distribution. The red lines show the distribution among four rendering nodes as determined by the cluster-level static load balancer. The white lines show the small tasks that the renderer uses on each node for the thread-level work stealing load balancer.

In Chapter 7, we present a novel version of the rendering back-end and its deployment to generate on-set feedback for virtual production.

6.8 Results

This section demonstrates the performance of the server back-end and the distributed real-time ray-tracer. The cluster consists of 20 rendering nodes. Each node is equipped with two Intel

Xeon X5650 six-core processors running at 2.66 GHz. The processors do not support AVX instructions. Consequently, the ray-tracer falls back to SSE and packets of 2x2 pixels. We compared the performance of the ray-tracer in SSE and AVX mode on a modern machine and measured an average performance increase of 88.8% with AVX.

The nodes are connected with 1 GBit/s Ethernet. Moreover, there is a 10 GBit/s InfiniBand link between ten of the nodes. The rendering nodes send RGBA output with 32 bits per pixel. The master uses the S3TC encoder. The image resolution is 1280x720.

We used the example scenes shown in Figure 6.7 to produce the results. The scenes are textured with diffuse and specular maps. The city has 66 thousand, the tavern 1.38 million, and the hacienda 7.7 million triangles. All scenes contain parts where there is heterogeneity in the rendering cost. The background is the cheapest area. The city contains a river that causes secondary rays due to refraction. The tavern contains a wet reflective table. The most demanding scene is the hacienda with refraction for the glasses and the fountain as well as a large amount of leafs that are rendered via alpha mapping. Each scene has a single light. There are 16 ambient occlusion rays per hit for the city and eight for the tavern and the hacienda scene.

Figure 6.7: The city (top left), the tavern (top right), and the San Miguel hacienda scene. The city rendering visualizes the tiling into rendering node tasks.

For reproducible results, the master automatically replayed a recorded set of camera interaction events for each scene. The camera movement is continuous, and the view changes between

consecutive frames are small as expected in a real-time scenario. The camera creates different viewing angles, effectively shifting the rendering cost distribution. The results build on the following per-frame measurements.

On each Rendering Node

- **Kernel**: The total cost spent in the ray-tracing kernel across all threads to determine the colors for the pixel packets. The load balancer aims to equalize the kernel cost on the nodes. Therefore, this is the core measurement to show the scalability.
- **Rendering**: The time to render the frame. The kernel executes on the logical cores using a work stealing scheduler. This value thus includes the thread management and also the kernel timing overhead.
- **SAT Generation**: The time to generate the SAT from the cost map.

Master Node

- **Tiling**: The time to determine the tasks for the next frame.
- **Pipeline**: The time in addition to the rendering spent to send the final image on its way to the display client and generate the rendering tasks for the next frame. The value includes the encoding and in case of distributed execution SAT generation, image and SAT transfer, and the tiling.

6.8.1 Scalability

Table 6.1 states the single node performance for each scene to set the benchmark. For the kernel and rendering measurements, the table states the strong scaling efficiency in the cluster. The values are the averages across all frames. Figure 6.8 illustrates the performance increase as nodes are added.

The kernel exhibits a super linear scalability. With more nodes, the load balancer assigns increasingly smaller tasks to the nodes in a fixed order. This can result in an increased cache locality, which we attribute the super linear effect to. Also, the foundation for the strong result is the accurate rendering cost balance that the algorithm can derive from the SAT array generated for the previous frame.

Along with the kernel, we observe a strong scalability for the rendering time. The rendering includes the thread management overhead, which stays about constant with more nodes. There

Table 6.1: The scaling efficiency and pipeline time for different node counts with reference to the single node (SN) performance.

City (**SN Rendering**: 384.1 ms, **Pipeline**: 0.884 ms)

	2	3	4	5	6
Kernel	100.8%	100.9%	100.5%	100.7%	100.4%
Rendering	100%	99.6%	99.1%	98.7%	98.6%
Pipeline	2.645	2.275	2.393	2.801	3.358
	7	8	9	10	20
Kernel	100.3%	100.4%	100.4%	100.6%	100.8%
Rendering	98.1%	97.7%	97.4%	97.3%	95.1%
Pipeline	3.714	4.022	4.29	4.495	36.98

Tavern (**SN Rendering**: 823.5 ms, **Pipeline**: 0.862 ms)

	2	3	4	5	6
Kernel	100.9%	101.3%	100.9%	101%	100.7%
Rendering	97.9%	98.2%	97.8%	97.9%	97.9%
Pipeline	1.376	1.979	1.615	2.033	2.046
	7	8	9	10	20
Kernel	100.8%	100.9%	101%	101.2%	101.5%
Rendering	98%	98%	98.1%	98%	95.9%
Pipeline	2.315	2.637	2.363	2.892	35.23

Hacienda (**SN Rendering**: 818.7 ms, **Pipeline**: 0.861 ms)

	2	3	4	5	6
Kernel	101.9%	102%	101.7%	101.5%	101.5%
Rendering	100.7%	99.1%	99.4%	99.2%	96.1%
Pipeline	2.4	2.259	2.098	2.307	2.187
	7	8	9	10	20
Kernel	101.2%	101.1%	101.3%	101.5%	101.5%
Rendering	96.3%	94.8%	95.3%	95.5%	92.9%
Pipeline	2.266	2.112	2.322	2.331	33.5

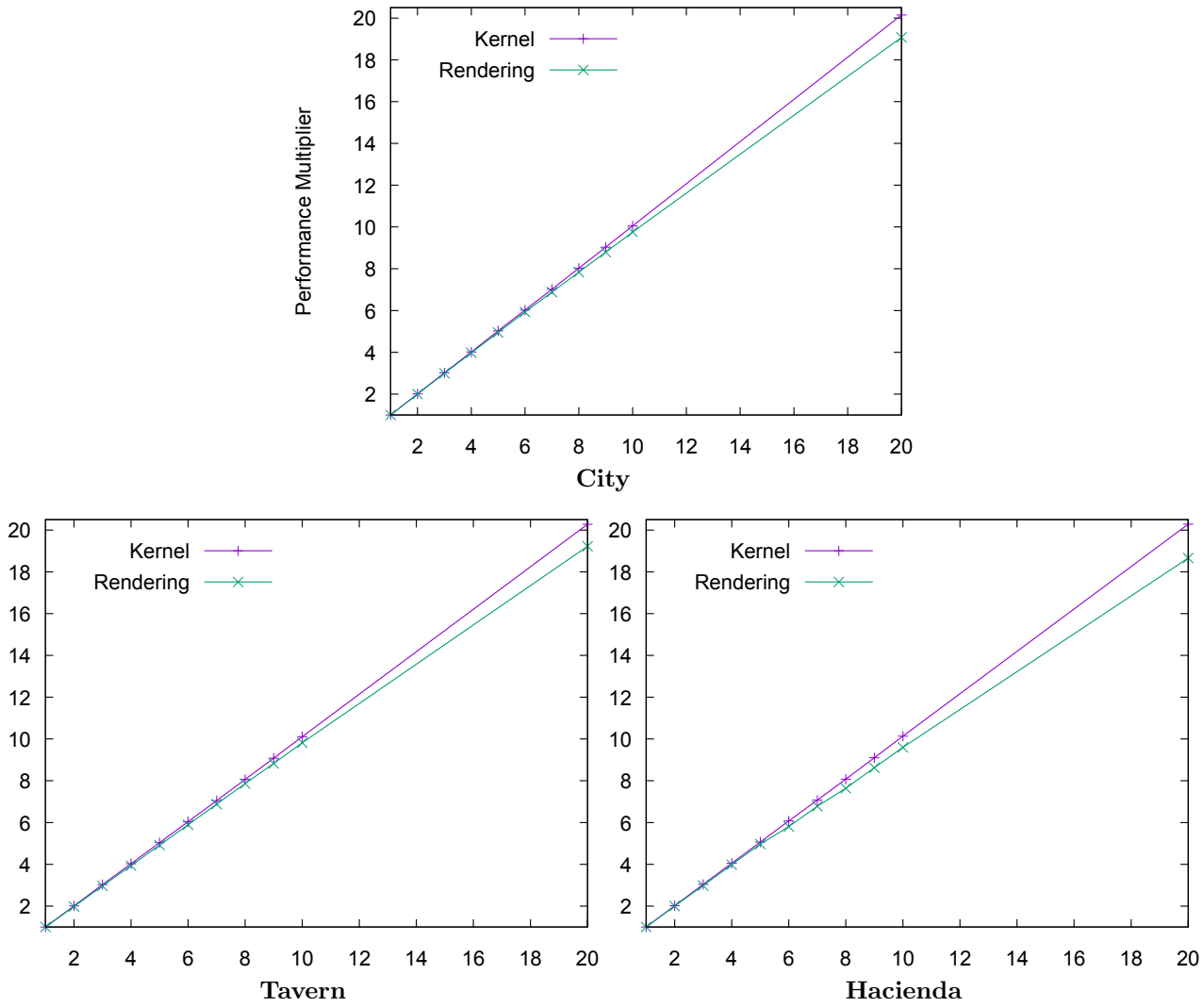


Figure 6.8: The kernel and the rendering performance increase as a function of the node count.

is also the per-thread overhead to iterate over and time the assigned pixel packets. This overhead depends on the task size and does thus not necessarily decrease comparatively to the kernel cost with more nodes. Therefore, the rendering time scaling efficiency is naturally below the kernel equivalent. As the per-node ray-tracing cost decreases with more nodes, the constant overhead accounts for a larger portion of the rendering time, which can cause the efficiency to gradually drop as nodes are added.

We observed an almost identical single node rendering time with en- and disabled cost map generation, which demonstrates that the timing overhead is minimal and becomes negligible with increasing per-packet cost. This is true for both the TSC and the performance counter as the timing mechanism. We only measured a marginally increased rendering time with the performance counter compared to the TSC.

Due to the execution on multiple cores, the rendering time is especially susceptible to fluctuation and outliers caused by outside interference, like the OS occupying a core for a different task.

Such occurrences can temporary reduce the scaling efficiency of the work stealer. The more nodes, the more likely a disruption on any node occurs. Also, the efficiency of the work stealer can fluctuate by itself. Imbalance of the thread-level scheduling reflects negatively on our cluster-level load balancer, which assumes a consistent scaling to the number of cores on each node. To mitigate the effect, we increased the process and rendering thread priority, which substantially reduced the appearance of outliers.

The scalability for both kernel and rendering remains stable over time with occasional minor fluctuation and outliers as Figure 6.9 illustrates. Only the second half of the hacienda run shows a more substantial fluctuation of the rendering time scaling efficiency. In contrast, the kernel measurements stay stable. The fluctuation is therefore not caused by our static cluster-level load balancer but by the thread-level work stealing, for which we employ the third-party solution CilkPlus. Figure 6.10 shows a consistent result with single-threaded execution.

6.8.2 Pipeline Time

The tendency is the increase of the pipeline time with more nodes. The master participates in the rendering. As the task size assigned to the master shrinks with more nodes, the network load increases proportionally. For 20 nodes, we switched from 10GBit/s InfiniBand to 1GBit/s Ethernet, and the transfer speed therefore drops substantially.

We observe a higher pipeline time for the city than for the other scenes in Table 6.1. Even though all scenes show a strong scaling efficiency of the renderer, the nodes still do not finish their tasks exactly at the same time and thus send their results to the master with some offset to each other. The tavern's and hacienda's higher rendering cost causes the offset to be higher in absolute time. Therefore, when the last node finishes its rendering, a larger part of the overall transfer already happened, resulting in a larger reduction of the pipeline time. Also, the sending with offset relieves the network interface on the master since there is less overlap of the incoming results.

The hacienda shows a slightly reduced rendering time scaling efficiency compared to the tavern from node count six onwards. Once more, the consequence is a higher offset between transfer operations, which prevents a pipeline time increase with more nodes as is the case with the other scenes. Vice versa, the scaling efficiency for the hacienda is initially higher than for the tavern, which results in the pipeline time being initially higher as well.

Some of our tests even show that minor load imbalance can be beneficial as the reduction of the network bottleneck outweighs a slightly improved rendering scalability.

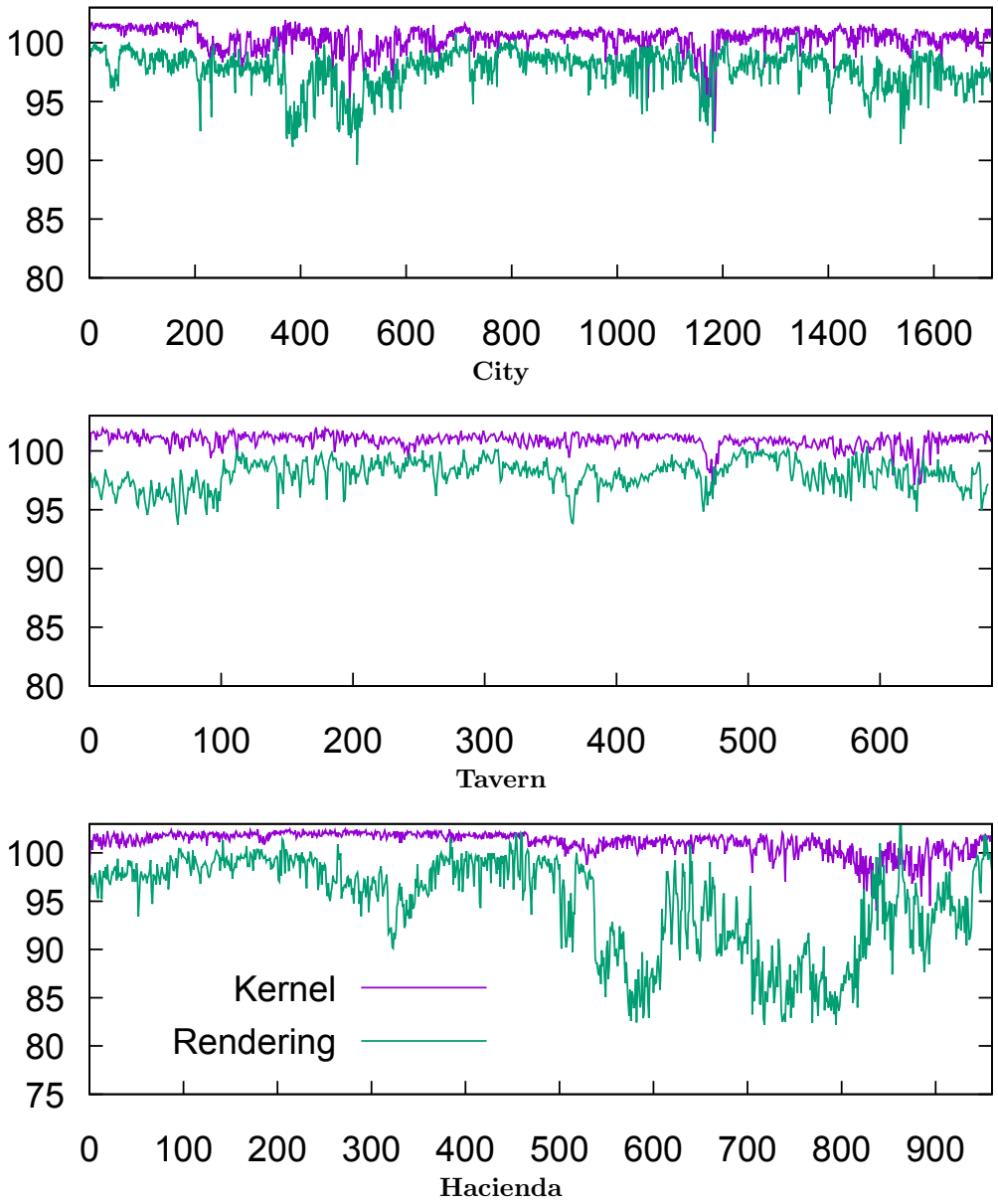


Figure 6.9: The kernel cost and rendering time scaling efficiency for eight nodes as a function of the frame number.

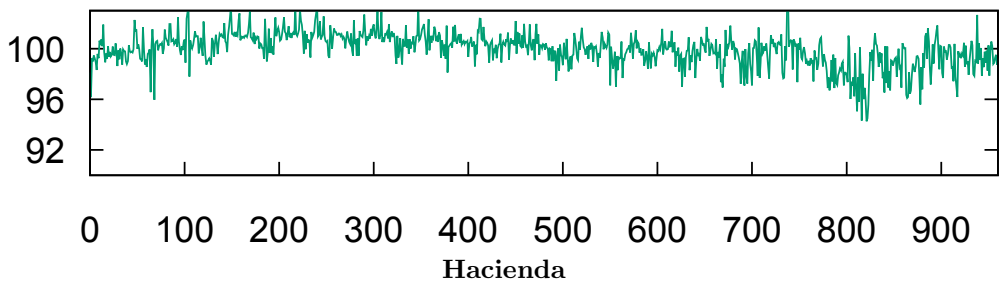


Figure 6.10: The rendering time scaling efficiency for eight nodes as a function of the frame number. For this run, each node disabled the local work stealing scheduler and only used a single rendering thread.

The SAT generation overhead is low even on a single node with around 0.244 ms. Due to the distributed SAT array generation, the overhead drops with more nodes.

The tiling time increases with more nodes as the master must split more tiles to find a task for each node. Most crucially, the sampling of the SAT array becomes increasingly expensive. However, the cost stays low with around 0.236 ms across the 20 node runs. For much higher node counts, we plan to extend the framework so that the master can also accumulate cost maps to generate one overall SAT on its own. For test purposes, we extended the tiling with multi-threading and ran it several times with a single SAT and 50000 tasks. The algorithm concluded in only 0.67 ms on average. For high node counts, the heavily accelerated tiling outweighs the distributed SAT generation benefit.

6.8.3 Comparison

We observe a similar scaling efficiency compared to Cosenza et al. [CDE13]. They utilize a cluster-level work stealing scheduler, which makes a low-latency network between the rendering nodes mandatory. In contrast, we achieve the results with a static load balancer that allows a flexible network setup. Their system is not interactive even for the highest presented node count of 16.

Further, we repeated the run for the city and eight nodes but this time only measured the overall cost of a task like Cosenza et al. [CCDC+08]. In that case, the scaling efficiency of the kernel drops substantially to 49.8%. To achieve competitive scalability, Cosenza et al. [CCDC+08] incorporate a task queue to compensate the inaccuracy. In contrast, our method can solely rely on the fine-grained timing mechanism.

We further performed a comparison with prevalent dynamic approaches on the thread level. For this, we repeated the runs on a single node with disabled ambient occlusion and used three different load balancing methods to distribute the tasks among the threads: our static load balancer, a task queue, and work stealing. Table 6.2 shows the rendering performance of the static method in competition with the dynamic methods.

The static approach performs almost on a par with the dynamic schedulers. Dynamic load balancers are ideally suitable locally due to the direct link between a moderate amount of cores. However, within a cluster, network communication and the coordination of many nodes can decrease the efficiency of these approaches. The task queue on the master can become a synchronization bottleneck if there are many simultaneous requests. A low-latency network is essential and must be available between all nodes in case of work stealing. In contrast, our method scales independent of the latency and can utilize nodes that are not connected to each

Table 6.2: The rendering performance of the static load balancer on the thread level relative to prevalent dynamic approaches: a task queue using OpenMP's dynamic scheduler and work stealing using CilkPlus.

	City	Tavern	Hacienda
Task Queue	-3.6%	-3.4%	-1.4%
Work Stealing	-0.9%	-2.1%	-1.4%

other. Only the tiling overhead increases with the number of nodes but stays at a negligible level.

6.8.4 Reduced Frame-to-Frame Coherence

The load balancer relies on a strong coherence between consecutive frames in real-time rendering. To test the method under restricted conditions, we repeated the runs with the city scene but used a new set of interaction events with coarser view changes this time. The new set contains only every fourth view of the original set. As expected, the accuracy of the load balancer drops with the larger discrepancy between frames, which Figure 6.11 illustrates. Though, the scaling efficiency is still strong. While the load balancer breaks if view changes become arbitrary, the results demonstrate that the method is well-suited for an interactive environment with continuous camera movement.

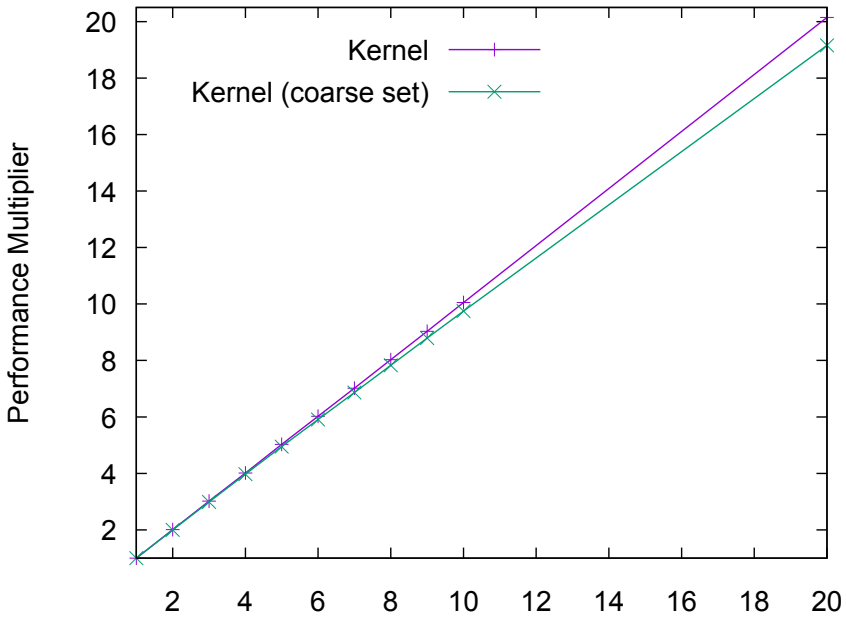


Figure 6.11: The city scene kernel performance increase as a function of the node count for the original and the coarser interaction set.

6.8.5 Multiple Clients

To test the system under more pressure, we repeated the run with the city and eight rendering nodes. This time, we connected three clients simultaneously. For each client, we measured a rendering time comparable to a single client that uses three nodes. Due to a minor offset between the connections, there is a short span in the beginning and end where not all renderers are active. This explains the higher than expected average performance per client.

For each client, the load balancer still achieves a strong kernel scalability, which only drops by around 2.6% compared to a single client. Due to the fine-grained timing mechanism in packet space, the OS unlikely switches to another thread during a measurement. Therefore, the measurements within a rendering session are mostly unaffected by the other clients and remain stable. The load balancer can consequently operate each session accurately, which ultimately results in an equally smooth execution for all clients.

6.9 Conclusion and Future Work

The contribution of this chapter is twofold. We presented the extension of the XML3D framework, which enables declarative 3D content in the web, with server-based rendering. The minimally invasive integration keeps the application logic untouched in the XML3D front-end, enabling arbitrary existing and upcoming applications to harness the back-end's power. The back-end is capable to run different renderers in a cluster. We presented a static load balancing method to distribute a real-time ray-tracer in this architecture. The load balancer exploits temporal coherence between adjacent frames in the real-time scenario. Based on high-resolution timings gathered for the previous frame, the load balancer derives rendering tasks of balanced cost for the potentially heterogeneous nodes in the cluster. We demonstrated the strong scalability and low overhead the approach can achieve.

The combination of XML3D, which enables generic and portable graphics applications in the browser, and the dedicated server back-end, which gives these applications access to a selection of high-performance and possibly distributed renderers, makes our architecture accessible to both the common web developer and the expert user.

The main limitation of the current architecture is the necessity that the client holds and synchronizes the scene data, which the application logic may change at any time. We therefore plan to investigate the execution of the XML3D page in a headless, server-side browser environment. The client only runs a reduced XML3D version that captures user input and displays rendering results. The approach would enable XML3D to interface with the rendering back-end directly

and also remove potentially expensive XML3D features like data processing and animations from a less capable client. However, the original architecture remains a viable option as it utilizes the client-side resources for application state management in parallel to the server-side rendering and reduces server load.

Chapter 7

The Dreamspace Distributed Rendering Architecture for Virtual Production

7.1 Introduction

Virtual production encompasses the creation of movie and TV experiences using a composition of filmed and computer-generated elements. In the traditional approach, the production pipeline begins with the filming in the on-set environment including actors, scenery, and props and ends with the visual effects creation in post-production. However, postponing the embedding of visual effects till post-production severely limits the creativity and experimental freedom of the on-set professionals. It also puts more pressure on the post-production process to adjust for effects that have not at least preliminary been tested before.

The European Commission funded the Dreamspace project to develop a platform that allows film professionals to combine the real and the virtual world on-set and in real-time [GHJ+16]. The platform enables experimentation by giving interactive control over the on-set visualization.

Incorporating the CG elements already during filming not only increases the flexibility in the collaboration of director, actors, and the CGI experts but also improves the cost-effectiveness. Overlapping the two traditional pipeline steps can avoid lengthy adjustments in post-production as the evaluation already happened during filming. Another major goal of the project is to make the technology available and affordable for a wide range of production contexts. Usability plays an important role to allow the non-technical staff on the set to operate the platform without overhead.

This chapter details a major component of Dreamspace: the distributed rendering framework that provides high-quality and interactive renderings of the virtual scene in the on-set environment.

There is a close connection between the rendering framework and several other components developed within Dreamspace. The main display client is the LiveView application, which performs the compositing of the filmed and the rendered content. LiveView also acts as a hub to receive scene updates from other components, in particular from the light capturing system, the camera tracking and depth capture, and the on-set editing tools. Our rendering client plugs into LiveView to synchronize the scene with the server back-end and hand received images back.

The rendering server enables different levels of service to account for the possibly varying demands on set. For the use cases where a simple preview of the lighting or the alignment of virtual and real objects is adequate, we provide a rasterizer. We also support a direct illumination ray-tracer that enables additional material properties such as reflection and refraction.

To allow the professionals to judge the impact of virtual elements realistically, we require another renderer that can simulate lighting conditions with physical correctness. The framework's main renderer is thus a global illumination ray-tracer. Global illumination uses a computing intense Monte Carlo simulation that converges slowly to the correct image. However, to enable interactive experimentation, we require a high-performance system that translates scene changes into meaningful results immediately and ideally maintains real-time frame rates. We therefore present an architecture that can distribute the expensive rendering procedure to an arbitrary number of nodes in a standard or InfiniBand network.

In addition to high performance, a key design goal of the framework is accessibility and usability. The architecture allows flexibility in the hardware and operating system it can be deployed on and provides several mechanisms to simplify and automate the installation and usage. While a dedicated high-performance rendering machine and network setup is recommended to achieve the best experience, the system is already functional in a commodity environment. In addition to the stationary LiveView client, we provide a simple portable web client for display only, which can be accessed from any device via a standard browser. We also integrated a client into the popular 3D modeling solution Blender [Ble17], which makes our system available to a large user base.

The remainder of the chapter is structured as follows: The next section discusses related work in the area of interactive distributed ray-tracing solutions. The following sections then detail the distributed rendering architecture, first the client and then the server side. The results section provides performance measurements and also an evaluation of the system during a multi-day test production at a film studio. We conclude with a summary and future directions.

7.2 Related Work

7.2.1 Global Illumination and Real-time Ray-Tracing

Global illumination algorithms attempt to produce realistic lighting conditions by simulating advanced material properties such as diffuse reflections, caustics, and subsurface scattering. Integrating these methods into practical applications like film production renderers is a topic of much interest [TL04, KFC+10, CHS+12, LGXT17].

While rasterization-based approximate methods exist, which can produce adequate results for domains like games, we ultimately require realistic and artifact-free image quality at high resolution. This is important as critical decisions affecting the production cost and time as well as the quality of the final product depend on how well visual effects can be judged and prepared on-set. Recently, there is a research focus on progressive Monte Carlo techniques [GKDS12, DKHS14] that produce noisy results initially but eventually converge to the correct image.

In Dreamspace we do not only require high-quality images with low noise but also need to produce these images at interactive frame rates. However, ray-tracing and in particular global illumination is an expensive task requiring a substantial amount of computing power. There are several low-level frameworks that are highly optimized for performance. An early attempt is OpenRT [WPSB03], which provides an OpenGL like API for CPU ray-tracing. The current state-of-the-art on the CPU is the Embree ray-tracing kernel library [WWB+14]. OptiX [PBD+10] enables generic ray-tracing applications leveraging the GPU.

Our direct illumination ray-tracer builds on top of Embree. The high-quality ray-tracer is a custom implementation [PGM16, PGTM16] that can leverage both CPU and GPU and supports several progressive global illumination techniques including the recent Vertex Connection and Merging [GKDS12].

7.2.2 Distributed Rendering

Even with the most optimized renderer, a single machine may not be able to uphold interactive performance for high-resolution ray-tracing. This is especially crucial in our setup that should be already operational with standard hardware. Ray-tracing is an embarrassingly parallel problem and thus ideally suitable for distributed execution. In Chapter 6, we described the importance of load balancing to account for the heterogeneity in the ray-tracing workload and achieve linear scalability. We also presented a distributed ray-tracing system with a web-based front-end. That system is the basis for the Dreamspace framework.

Benthin et al. [BDWS02] describe an interactive system that finds its application in the car industry. The foundation is OpenRT, which has native support to distribute the ray-tracing tasks. A more recent library with distributed rendering functionality is OSPray [Int16b], which utilizes Embree internally. However, our focus is on complete pipelines that support distributed ray-tracing with global illumination and are potentially suitable for an on-set environment.

Several professional solutions supporting distributed ray-tracing have emerged [Cha17, NVI16d, Aut16, OTO15b, Ren16]. We concentrate on the most prominent examples that specifically target interactive rendering. The recent MoonRay [LGXT17] is a fully vectorized production renderer that uses Embree internally. While MoonRay also targets interactive performance and cloud-based rendering [FHF+17], not enough details for a proper comparison are available at this time.

V-Ray [Cha17] provides a setup with a scattering load balancer, where the coordinating master node also acts as the display client. The master connects to the other nodes via TCP/IP to collect raw pixels. While V-Ray focuses on offline rendering, the extension V-Ray RT enables interactive performance. The rendering engine is split into two versions. The CPU version is feature-rich and resembles the regular V-Ray offline renderer. It uses Embree for acceleration. The GPU version is optimized for performance but has limited features.

NVIDIA offers a special hardware called Visual Computing Appliance (VCA) [NVI16d] at a price of around 50.000$ as of 2016. The VCA contains several high-end GPUs and comes with built-in support for OptiX and Iray [NVI16b]. Iray is a rendering solution built on top of OptiX and supports photo-realistic, interactive, and real-time rendering modes. Other production-level renderers accelerated with OptiX are FurryBall [Art15] and Mental Ray [NVI16c].

The display client connects with 1 or 10 GBit/s Ethernet to the VCA, which returns lossless images, JPEG, or H.264 video. The VCA takes care of distributing OptiX calculations and Iray rendering tasks to the GPUs. The interconnection of several VCAs via InfiniBand is possible. The cluster manager automatically handles scene distribution and load balancing for OptiX and Iray. Third-party applications are possible but must handle the pipeline on their own. An example is V-Ray RT, which can run on VCA.

Above solutions provide fully featured production-quality renderers and are integrated into post-production software like 3ds Max, Cinema 4D, and Maya. While a complete set of state-of-the-art rendering features could not be addressed within Dreamspace, we provide a more flexible framework. V-Ray RT requires to connect all nodes including the display client in a local high-bandwidth network to transfer raw pixels. The setup is not extendable with third-party renderers. While VCA supports streaming encoded images to a display client outside the cluster, the solution depends on dedicated high-end hardware. VCA only has built-in support

for OptiX and Iray. Third-party renderers must implement a custom scene distribution and load balancing pipeline.

The Dreamspace framework provides a flexible architecture that is already functional with commodity hardware and networking. The server supports several methods to stream images to the display client including bandwidth-efficient video, which facilitates access from a remote set or a best-effort network. Third-party renderers integrate with the existing infrastructure. Several renderers can coexist to provide different levels of service. The client can run in a web browser without requiring a plugin, giving users access from a standard web page with their possibly mobile devices. The solution already runs in LiveView, a display client specifically designed for on-set usage, and in Blender, but the integration into other clients like 3ds Max and Maya is possible.

The scene distribution approach is similar to Iray on VCA and V-Ray RT. There is an initial heavy distribution and caching step of the on-disk scene managed by the display client followed by subsequent incremental live updates.

7.3 Architecture Overview

Figure 7.1 illustrates the distributed rendering pipeline. The goal of the pipeline is to provide the display client with high-quality, globally illuminated rendering results. The pipeline should be real-time capable and give the user immediate feedback when navigating or manipulating the scene.

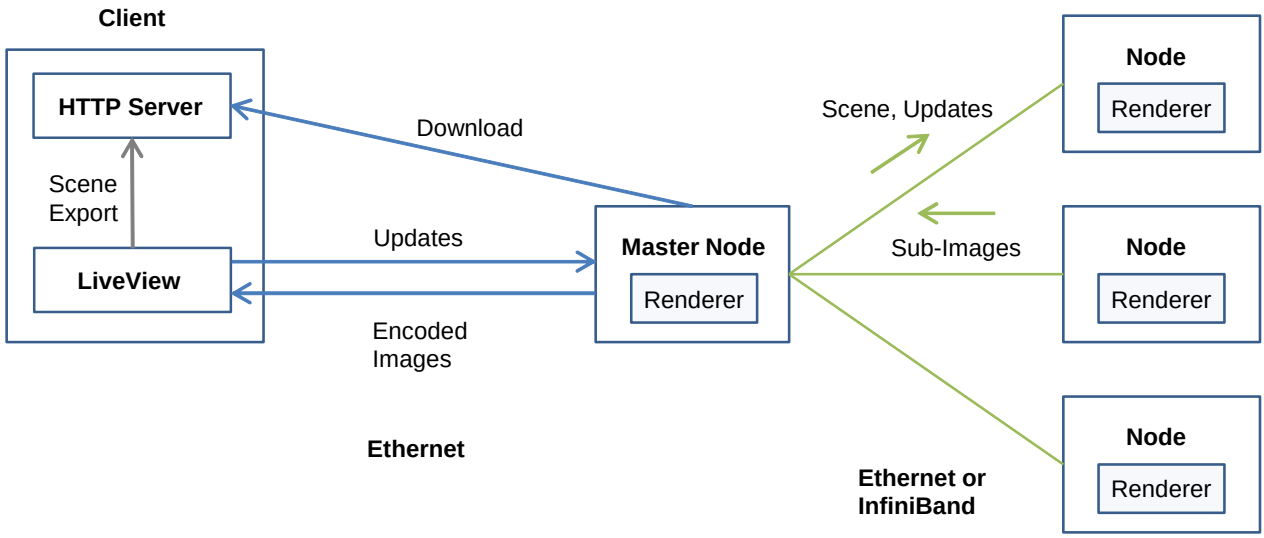


Figure 7.1: Architecture of the on-set distributed rendering system. LiveView connects to a rendering cluster that operates in a standard or dedicated InfiniBand network.

The main on-set client machine runs the LiveView application, which loads the virtual scene to be used during the production. LiveView supports a plugin mechanism to integrate renderers. A plugin has access to the initial scene and subsequent updates like camera movement and passes its rendering results back to LiveView for compositing and display.

The Dreamspace rendering plugin exports the scene into a generic and portable XML format. The export is made available to the outside via a HTTP server and only regenerated if LiveView loads a new or modified scene. The HTTP server is an independent process and therefore can provide the scene even if LiveView is not up.

The plugin connects to the master node of a rendering cluster, which downloads the scene from the HTTP server and distributes it among the rendering nodes. Due to server-side caching, download and distribution only occur if the HTTP server advertises a new version of the scene.

Once the renderer on each node has loaded the scene, updates issued by LiveView trigger the rendering of new frames. The plugin sends updates to the master node, which distributes them in the cluster. The master collects the partial images produced by the nodes and forwards the final result to LiveView for display.

The pipeline is specifically designed for real-time operation. The execution model is asynchronous to allow client, network, and rendering to operate in parallel. While the cluster renders the current frame, the network transfers the previous frame to the client, which already prepares and sends updates for subsequent frames. A server queues updates when occupied and restarts its renderer immediately from the queue, enabling full utilization.

7.4 Client Side

This section describes the display client applications to be used in the on-set environment: the stationary LiveView client, in particular our distributed rendering plugin for it, the flexible web client, and the Blender client. While we developed the rendering plugin for LiveView, LiveView itself was created by the project partner *The Foundry*.

7.4.1 LiveView

LiveView is the main display client, which runs on a Linux desktop machine on the set. LiveView can load virtual scenes authored in and exported from the professional lighting and look development tool Katana. Katana imports from modeling solutions like 3ds Max and Maya.

But while Katana is designed for production-level rendering, LiveView targets interactive usage. LiveView therefore provides an internal rasterizer, based on the physically-based shading model from Burley [MHH+12], as the default local renderer.

LiveView also provides an API to plug in other renderers. There are local production renderers like Arnold and PRMan as well as our network-attached plugin, which can access the distributed, high-performance rendering back-end. Multiple renderers can coexist at run-time, so switching back and forth to see and compare different outputs is possible. For example, a fast rasterizer may be adequate to review the geometry only. But to simulate the lighting and advanced effects realistically, a higher-quality renderer must be consulted.

LiveView gives the active rendering plugins access to the initial scene and lighting setup present on-disk as exported from Katana. LiveView also forwards subsequent run-time updates to the renderers. While LiveView currently only allows navigating the virtual camera via the GUI, it can receive changes to the appearance and transformation of lights and objects from an external source.

First, LiveView links to the light calibration system, which can capture the lighting conditions on the set automatically [EG15]. The goal is to harmonize the real and virtual lighting.

Second, LiveView links to the camera tracking and depth capture system [GHJ+16, Boi16]. The system enables the matching of the real and virtual camera in real-time. By capturing depth information for the real scene, it also enables LiveView to composite the filmed with the virtual content. This requires a renderer to return depth as well. The composition of virtual elements with a background video plate or the camera feed is an important task of LiveView.

Third, LiveView links to the on-set editing tools [TGS+15]. The tools run on tablet devices and provide an intuitive interface to control lighting, material, and object parameters. Each tablet stores a low level-of-detail version of the scene to allow interaction with a locally rasterized preview rendering. The tools can also directly connect to the camera tracking system to automatically match the real on-set camera.

To facilitate collaborative editing, parties other than a renderer that are interested in run-time updates can also plug into LiveView and distribute updates to external receivers. Consequently, there is a plugin to synchronize the scene and updates across the tablet editing tools. Another plugin communicates changes to the lighting parameters back to the real lights using DMX controllers.

7.4.2 Rendering Plugin

To allow real-time ray-tracing and global illumination to run in LiveView at the highest possible quality, we implemented a plugin that accesses a rendering cluster over the network. LiveView can be set up to load the plugin as the initial renderer, or the user can later select the plugin from the LiveView GUI. The plugin is also compatible with Katana, on which LiveView is based on and which provides a very similar plugin mechanism.

7.4.2.1 Configuration

There is a simple configuration file that points the plugin to the IP address and port of the cluster master node. The file may also contain the following optional settings.

Renderer: String
Like LiveView, the server side supports to integrate different renderers. This setting determines the renderer to use. If left empty, the server uses the global illumination ray-tracer by default.

Nodes: Integer
The number of rendering nodes to use. If this value exceeds the number of available nodes, it is automatically capped. Defaults to infinity.

Real-time Rendering Quality: Integer
An integer value telling the renderer the desired quality for real-time rendering, with 0 being the lowest quality. Real-time rendering should give immediate feedback in response to scene updates. The cluster must keep up to produce frames at the desired rate. The user may set this to 0 if only a single node is available and increase the value as more nodes are added. Increasing the value reduces the noise artifacts for global illumination as the renderer samples more rays, but this potentially requires additional nodes to maintain interactive performance. The scale is abstract to be applicable to any renderer. Thus, it is up to a specific renderer to map the values to internal parameters. Defaults to 0.

Progressive Rendering: Boolean
Once camera movement and scene updates stop, the server supports progressive rendering to gradually refine the quality of the static view. For the global illumination ray-tracer, this means adding more samples to the existing image with each pass. New updates interrupt the procedure and the system goes back to real-time rendering. This setting has no effect for renderers that do not support the feature. Defaults to “true”.

Maximum Frames Pending: Integer
The plugin synchronizes scene updates with the master node and thereby requests the rendering

of new frames. This setting determines the maximum number of frames the client may request in advance before having received a rendering result back. Setting this to 1 creates a synchronous pipeline since the client waits for the result of each requested frame before requesting the next one. A value higher than 1 enables asynchronous execution but may decouple the user interaction from the displayed result if the rendering is a noticeable amount of frames behind. Defaults to 2.

Encoder: String
Select how the master node encodes the rendering results for network transfer to the client. We discuss the available options in Section 7.4.2.4.

7.4.2.2 Loading and Exporting the Initial Scene

On start-up, the plugin iterates over the scene graph provided by LiveView to gain access to the geometry, materials, texture resources, and lights. The plugin exports this information to a portable XML format stored on disk. The format is generic and supports transformations, instancing, and object groups. It is based on and directly compatible with XML3D [SKR+10]. The approach allows us to load a scene directly into the web client described in Section 7.4.3.

To make the export available to the rendering cluster, we deploy a standard HTTP web server. The server runs independently to LiveView and thus gives access to the scene even if LiveView is not active.

The LiveView version of the scene and the export contain a timestamp. To speed up the next loading of the scene, the plugin uses the timestamps to determine whether the scene has changed and only triggers another export if that is the case.

Using a text-based, generic format is less efficient than a dedicated binary-only format, for both network transfer and loading into a renderer. However, we followed this concept including the exposure via a web server to make the scene easily available to and usable by potential third-parties other than the distributed rendering back-end. In our case, there is XML3D in particular. The approach goes along with one of the main goals for the overall architecture: flexibility. Also, while the scene structure and properties reside in XML files, geometry and textures, which are the vast majority of the scene data, reside in binary files. We utilize Blast [SSS14] as the geometry format. Further, the server side converts the export into an efficient binary format tuned for the distributed rendering back-end and cached on disk, thus enabling immediate consecutive loadings of the same scene.

7.4.2.3 Dynamic Updates

After loading the initial scene, the plugin connects to the master node of the rendering cluster via TCP/IP. During a handshake, the client communicates the settings from the configuration file to the server, which concludes the establishment of the rendering session.

LiveView now passes run-time updates to the plugin. There can be several updates passed at once. The plugin immediately forwards a set of updates to the master node and thereby requests the rendering of a new frame if the maximum number of pending frames has not been reached. Otherwise, the plugin accumulates and potentially overwrites redundant updates until the rendering result for a previous request comes in. The plugin then flushes the updates to the server. The mechanism allows asynchronous execution due to different frames being at different stages in the pipeline at once. However, the server side queues rendering requests and corresponding updates if it is still busy with a previous request. Restricting the number of frames that the client requests in advance prevents the queue to fill if the cluster cannot keep up with the update rate, which ultimately prevents an increasing delay in the visual response to updates during real-time rendering.

Since the LiveView scene graph and dynamic updates are string-based, the plugin maps the updates to a binary format for more efficient communication to the master node. This step induces only minor overhead.

The plugin currently supports camera, light, and material property updates, as well as light and object transformations. While the server back-end also supports updates of the geometry and textures, this is not exposed in the plugin at the moment as there was no corresponding use case in Dreamspace. However, the client already flags meshes as static, transformable, or unstructured to facilitate renderer-specific optimizations for building acceleration structures.

7.4.2.4 Receiving Rendering Results

The master node sends the images produced by the cluster to the client for display. Figure 7.2 shows the global illumination ray-tracer in LiveView.

Our goal is to allow a flexible network setup to connect the on-set and potentially other clients to the cluster. We consider the use case where a client connects from a remote location over a best-effort network like the Internet. Vice versa, there may be a remote rendering cluster not located on the set. The web client may run on mobile devices and therefore connect over a wireless network. Consequently, bandwidth and reliability of the link between client and master node may be limited.

Figure 7.2: The global illumination ray-tracer in LiveView rendering the San Miguel hacienda scene, which was used as a reference throughout Dreamspace.

The server therefore supports several methods to encode the images. The client can select the most appropriate option as required. Chapter 5 gives details about each option.

The fastest variant is S3TC. However, S3TC can produce artifacts for sharp edges and gradients and also is the least bandwidth-efficient method, apart from raw pixel transport. The server thus provides alternatives that induce more en- and decoding overhead but facilitate the deployment under restricted network conditions and potentially generate better image quality. The most bandwidth-efficient method is H.264 video. The server supports constant quality and constant bit rate streaming using the *x264* library as outlined in Section 5.4.6. Since *x264* is a software encoder, though a highly optimized one, we plan to incorporate hardware-accelerated alternatives in the future. In addition, there is support for JPEG, which any display device should be able to decode.

The distributed rendering framework currently does not return a depth map needed for compositing in LiveView. We want to address this limitation in a future revision. However, LiveView currently always requires floating point images with 32 Bit per color channel in CPU memory for the composition step. Since the encoded images provide only 8 Bit per channel, the plugin could perform a conversion after decoding. The conversion causes only minor overhead and avoids the substantially increased network load for sending pixels at their native precision.

7.4.3 Web Client

In addition to LiveView, we provide a client that runs in a web browser. The client utilizes the extended XML3D presented in Chapter 6, which supports server-based rendering and is compatible with the Dreamspace distributed rendering back-end. Users have access through a standard web page without requiring a plugin.

The basic thin version of the web client does not have a local copy of the scene. It is intended for display only, thus only allowing camera movement. This works as the server side automatically loads the latest scene it has cached if there is no new version advertised by the HTTP server or no HTTP server is available.

However, the XML scene format exported by the LiveView rendering plugin is compatible with XML3D. The web client can therefore potentially load part of or the entire scene, given the scene size does not exceed the capacity in the browser environment. The client can then use its local WebGL renderer or synchronize the scene with the server. This enables the web client to implement editing interfaces for lights and objects, which facilitates experimentation independent of the main LiveView client. The interfaces may also allow to add new lights and objects that are not present in the original scene.

While we have not implemented a web client with editing functionality yet, the concept enables a lot of flexibility in where and how to access the rendering cluster. Figure 7.3 showcases an exemplary setup with multiple levels of service that is possible with the display client and cluster architecture.

LiveView connects to a master node that has access to five rendering nodes. The master keeps all these nodes up-to-date if a new scene export occurred. The upper three nodes plus the master are dedicated to provide the best quality renderings to the main display client. They could be in a high-speed cluster network with InfiniBand support. But every node can assume the role of the master, which facilitates accessing the cluster at different entry levels. Consequently, the lower two nodes provide preview renderings to the remaining devices that run the web client. A commodity machine and network setup could be adequate here.

The clients can independently navigate and update the scene within their rendering session, enabling different settings to be tested in parallel. The flexible image transfer options promote the collaboration with remote clients.

A node may support both renderers. It is further possible that several server-side renderers exist at the same time in LiveView. This can be achieved by simply duplicating the plugin on disk and loading each one in LiveView with a different renderer and possibly server in its

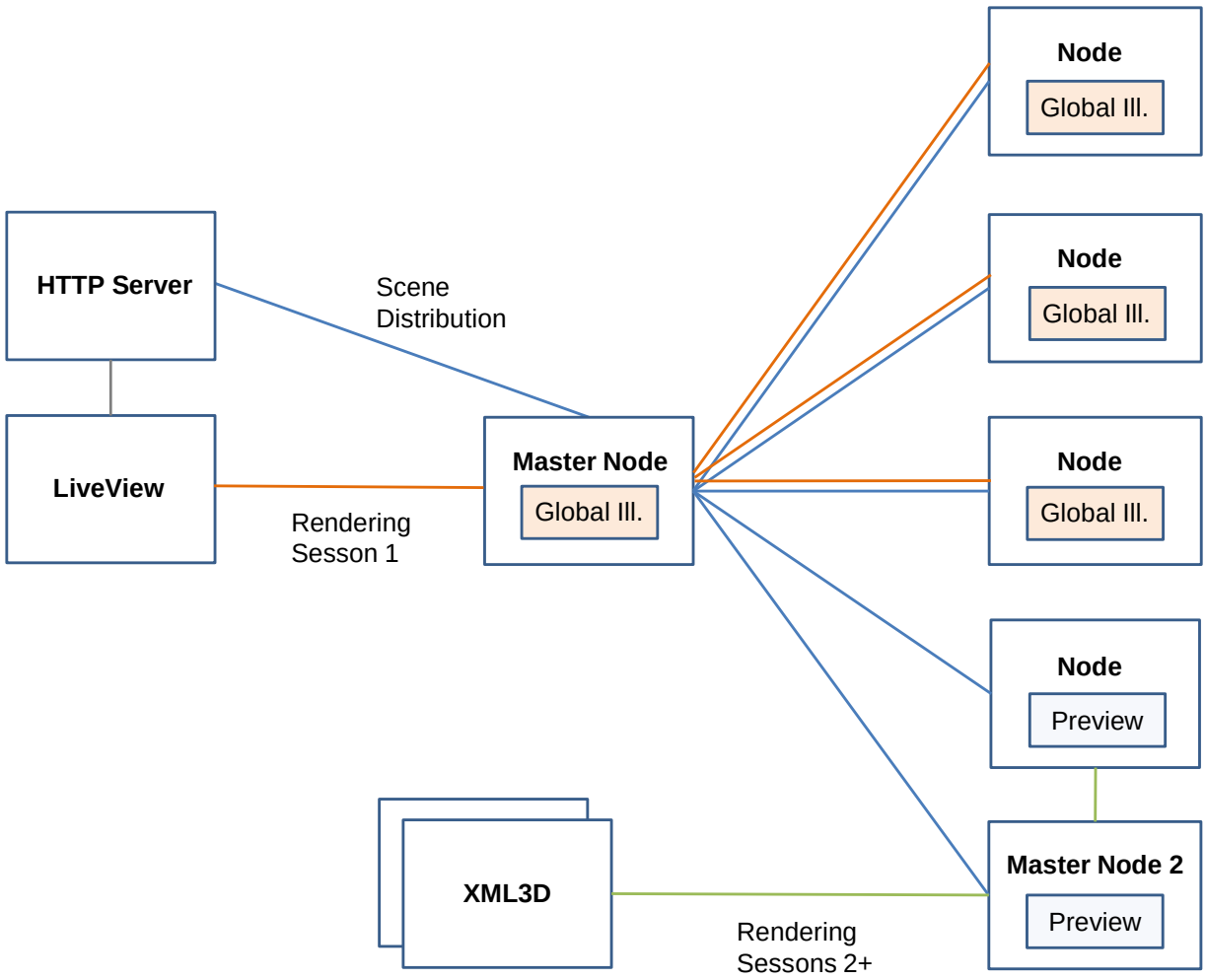


Figure 7.3: Exemplary distributed rendering architecture with web clients and two server-side renderers.

configuration file. The option enables to use multiple clusters in LiveView.

7.4.4 Blender

Blender is a portable and open-source solution for 3D modeling and animations that has a large worldwide user base and also has occasionally been used in professional productions. Blender provides an offline as well as an interactive rendering mode and can import scenes from various other tools like 3ds Max. Third-party renderers can plug in using a Python API. To make our server back-end available to the Blender community, we implemented an interactive rendering plugin for Blender.

Our plugin integrates with Blender's GUI and provides the options described in Section 7.4.2.1 to setup the renderer. Figure 7.4 shows the direct illumination ray-tracer running in Blender.

To synchronize the scene with the server, the plugin can generate an exported version of the

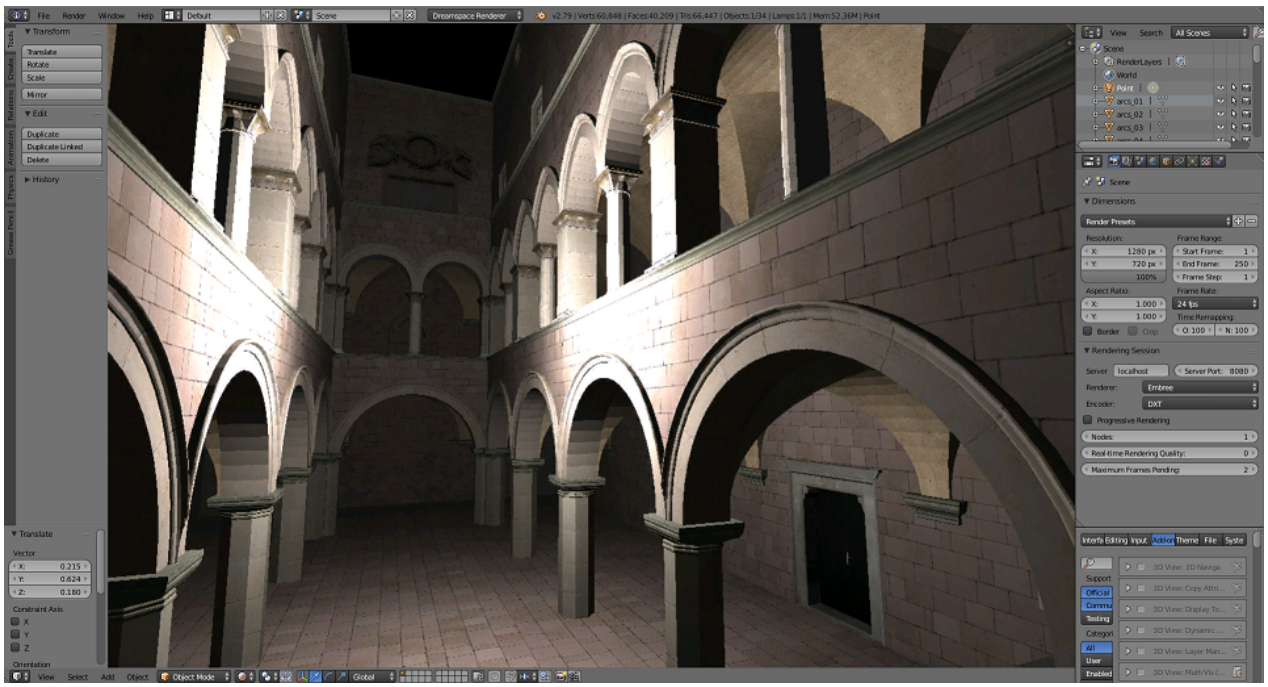


Figure 7.4: The direct illumination ray-tracer rendering a scene in Blender.

scene similar to the LiveView plugin. But in contrast to LiveView, the Blender plugin can directly send the export to the rendering server without requiring a HTTP server. Also, for efficiency reasons and to streamline the export pipeline, the plugin bypasses the XML format and directly generates the proprietary binary format the server side uses to distribute the scene in the cluster.

While using the export feature speeds up subsequent loadings of the scene, this is not mandatory. The plugin supports dynamic updates including geometry and textures during rendering, and the dynamic updater works together with the exporter. If there is no export available, the updater synchronizes the entire scene at each start-up of the renderer. Otherwise, the updater only synchronizes differences between the currently loaded scene in Blender and the export.

The plugin supports a subset of the scene, mesh, material, and lighting settings exposed in the GUI for Blender's native renderer. Blender also provides a second internal renderer named Cycles that uses physically-based rendering (PBR) materials and supports global illumination. To also facilitate a material workflow for PBR, our plugin supports glTF [Khr17b] as a second export format by reusing the official glTF exporter [Khr17a] for Blender. glTF is a generic delivery format for 3D scenes that supports PBR materials and has received considerable attention in recent years. However, since glTF does not support incremental updates yet, dynamic updates are disabled when using the format. Also, the integration is still experimental since none of our server-side renderers are capable of loading glTF at this time.

7.5 Server Side

This section describes the server side of the rendering architecture. The framework enables to run a renderer distributed in a cluster setup as the right side of Figure 7.1 depicts. The master node interfaces with the client side to receive scene updates and send generated images back for display.

7.5.1 Deployment

A major design goal of the rendering back-end is to allow a flexible deployment to facilitate the usage under different on-set conditions and production budgets. We target a dedicated high-end rendering node and network setup, which is recommended to provide the best results with the real-time global illumination ray-tracer. But we also target a less costly approach that interconnects commodity machines in a standard network. The investment in fewer or less powerful nodes may be adequate for use cases that do not rely on high rendering resolution and quality.

Another goal is to simplify the installation and configuration procedure for the on-set professionals, which do not necessarily have technical expertise.

The rendering server runs under both Windows and Linux. To form a cluster, the server must be installed on several rendering nodes that are able to connect to each other via Ethernet. Section 7.5.4 describes the distributed encoding that allows a low overhead transport for high-definition images already within a 1 GBit/s cluster network. However, the server also supports communication over InfiniBand to maximize the performance, either with a low-level implementation using the *libibverbs* [Ope17] library or indirectly via IP over InfiniBand (IPoIB) [Chu06].

To simplify the installation, we provide a self-contained package that the operator can copy onto a new system without the necessity to install missing libraries. In a future revision, we plan to provide system images for cloning onto rendering nodes that do not run the recommended or any operating system yet.

Once the server runs on all nodes, the display client may connect to any of the nodes to initiate a rendering session. The node that accepts a client connection automatically assumes the role of the master. The master coordinates the distributed rendering. It broadcasts scene updates in the cluster and collects the partial images produced by the nodes. Section 7.5.6 describes the load balancing method to assign rendering tasks to the nodes for linear scalability.

The master node may participate in the rendering, which is feasible to improve performance

and reduce the network load in the cluster. A setup with only a single node is thus possible. But the master can also be a network hub only, giving access to the rendering nodes. This allows to use a machine as the cluster entry point that is dedicated for networking but not necessarily suitable for rendering. Also, the master can encode images for transport to the client over a possibly bandwidth-limited network. It may therefore be beneficial to avoid competition with a local CPU-bound renderer. Due to asynchronous execution, the renderer may already work on the next frame while the encoder processes the current one.

There is a configuration file on the master node to specify IP address and port of each server. Also, the file tells the master whether to participate in the rendering.

While the operator can manually set up and edit the configuration file, the server also supports an automatic discovery mechanism to find rendering nodes. The mechanism is triggered if a client connects to the master and there is no configuration file. The master then attempts to find servers running on other nodes and generates the file. For this feature to work, the nodes must be able to communicate via multicast. The operator can still manually edit the configuration later, for example to disable the master as a renderer, which is the default setting, or to prefer different network interfaces, like an IPoIB over a lower-bandwidth Ethernet interface.

7.5.2 Downloading and Distributing the Initial Scene

When a display client connects, the master node downloads the most recent version of the scene from the HTTP server. The operator can set up any HTTP server in a configuration file. In the designated use case, the server runs along with LiveView on the main client machine. This allows the LiveView plugin to export the scene directly to the HTTP server without another intermediate distribution step being necessary. Therefore, if no HTTP server has been configured, the master automatically attempts to reach the server on the client side that connected. The concept enables a self-contained setup that requires minimal configuration but also allows to run the scene server anywhere independent of where LiveView runs.

Since the downloaded scene is a generic, text-based XML format, the master translates it into a proprietary binary format only containing raw data for fast loading into a renderer. The master caches this binary format on disk. It also creates a second cache that utilizes geometry buffer and texture compression to accelerate network transfer and distributes this cache across the rendering nodes. Each node reconstructs the raw binary cache from the network cache for fast local loading of the scene. Download and translation only occur if the HTTP server advertises a new version of the scene. Transfer to a rendering node only occurs if the node has not already cached the latest version. Like the client side, each node manages a timestamp to

determine whether it should receive an updated scene. The master node automatically loads the current cache if the HTTP server is unavailable. A constantly running HTTP server is thus not mandatory once the scene has been deployed in the cluster.

Depending on the size of the scene, the initial export and cluster distribution procedure can take several minutes. A major cause is the traversal and conversion of LiveView's scene representation. Due to the client- and server-side caching, subsequent loadings of the scene take seconds or even less.

Loading and caching a LiveView scene before commencing the rendering is the designated use case in Dreamspace. But we have built the server back-end with more generic functionality in mind. The server supports to receive scene data and updates while the rendering session is already active. The client can thus create the scene incrementally and add, remove, or manipulate elements anytime. This is supported in both the XML3D and the Blender client. The master can also receive the network cache directly from the client side, which is supported in the Blender client.

7.5.3 Dynamic Updates

After loading the initial scene, the master node is ready to receive run-time updates from the client. The master forwards the updates to the nodes that participate in the rendering session.

The server supports lightweight updates of camera, light, and material properties as well as dynamic geometry and textures. Meshes can reference and reuse data to facilitate implementing optimization strategies such as instancing in a renderer.

The display client may generate updates and corresponding rendering requests at a variable frequency. If a renderer completes a frame before the next update iteration arrives, it becomes idle for a short interval. If a new request arrives during the interval, the renderer can immediately start the next frame. Otherwise progressive rendering may commence, which must then be interrupted before starting the next frame. The idle time also frees resources for other clients that may be connected in parallel.

The client can fill the pipeline with multiple rendering requests to allow client, servers, and network to operate in parallel on different frames. Therefore, updates may also arrive at a higher frequency than a renderer can reflect. If a server is still busy with a previous frame, it queues incoming rendering requests. The server can immediately restart its renderer from the queue. However, the client should avoid filling the queue with outstanding requests, since this

results in the decoupling of the user input with the displayed image. The client consequently only requests a selectable number of frames in advance.

The ideal case is an exact alignment of the update frequency and the rendering frame rate, so the queue never fills up and the server side is still fully utilized. But the frame rate cannot be known beforehand a rendering session and may be subject to major fluctuation due to factors like server load and scene regions with heterogeneous rendering cost. Therefore, we plan to add a mechanism that automatically adapts the update frequency according to the frame rate.

7.5.4 Accumulating and Sending Rendering Results

The master node receives partial images from the rendering nodes and accumulates them in a final frame buffer. The server back-end supports two methods to transport the images in the cluster.

Each node can send its output as raw 32 Bit RGBA pixels. The master encodes the final frame buffer for bandwidth-efficient transport to the display client over a potentially unreliable, best-effort network. Section 7.4.2.4 describes the available encoding options. To avoid considerable overhead, raw pixel transport requires a high-bandwidth network especially for high image resolutions. We recommend to use this option only in a cluster setup with InfiniBand or 10 plus GBit/s Ethernet.

To achieve the lowest possible latency for both encoding and intra-cluster image transport, the back-end also supports a distributed encoding method. Each node encodes its partial rendering result using S3TC. The overhead is minimal and negligible. We rendered 720p images in an eight node cluster with two heavily outdated Xeon X5650 CPUs per node and measured only around 0.07 ms encoding time per node. The fixed compression ratio of 8:1 for raw RGBA input enables even a 1 GBit/s commodity cluster network to accumulate high-resolution images at the master node quickly. A costly high-bandwidth Ethernet or InfiniBand setup is therefore still recommended but not mandatory. The master can directly forward the final S3TC image to the client. Removing the encoding on the master is especially important when considering image resolutions of 2K and more. However, if the client chooses to receive images as JPEG or H.264 instead, the master must transcode to the selected format. While this induces additional overhead on the master and may result in a loss of perceivable image quality, the improved transport performance in the cluster may still outweigh the disadvantages.

7.5.5 Renderers

The server back-end provides an API that developers can implement to plug in their renderers. We have included four renderers so far. Figure 7.5 demonstrates the two main options.

Figure 7.5: The web client shows the San Miguel scene rendered by the direct illumination ray-tracer (top left). The global illumination ray-tracer renders the San Miguel (top right) and the battleground scene.

7.5.5.1 Dummy Renderer

The dummy renderer ignores all scene input and renders a procedural sphere in the center of the screen. Each node color-codes its output differently, so the sphere has a colored check pattern if multiple nodes participate. The cluster operator can use the dummy renderer to confirm a working framework setup in case other renderers are still error-prone or unavailable.

7.5.5.2 Rasterizer

We integrated the rasterizer from Chapter 6, which mimics XML3D's WebGL renderer. As a notable addition to the WebGL counterpart, the server side implements geometry instancing and therefore performs especially well for scenes with many objects that share the same resources.

We intend the rasterizer to mostly provide preview renderings to the mobile web clients on the set. LiveView has a native rasterizer and thus would only rely on the server-side option if local rendering is not possible or performs worse.

7.5.5.3 Direct Illumination

We integrated the direct illumination ray-tracer from Chapter 6. The ray-tracer runs on the CPU, using Embree for high-performance ray packet traversal and the CilkPlus work stealer to execute on multiple cores.

The direct illumination ray-tracer is the intermediate option when it comes to quality. It can simulate reflection and refraction and is thus more suitable than the rasterizer for scenes where these features are important.

There is also support for ambient occlusion. While a single machine equipped with a decent CPU can already provide interactive frame rates for the direct illumination, ambient occlusion is an expensive Monte Carlo technique that requires a good amount of sample rays to converge to a smooth result. The ray-tracer therefore supports progressive rendering to refine the ambient occlusion effect gradually for static views.

7.5.5.4 Global Illumination

The global illumination ray-tracer builds on top of the AnyDSL compiler framework [LBH+15]. Using AnyDSL, the renderer can describe its core routines in a high-level programming style. AnyDSL takes care to map the algorithms efficiently to CPU or GPU target hardware. Pérard-Gayot and Membarth [PGM16] show the traversal routines perform on a par with the hand-tuned equivalents of Embree and OptiX. The ability to generate highly optimized code from concise and readable algorithm descriptions reduces the programming effort substantially and thus facilitates extending and prototyping the renderer with future advancements. Being able to target different platforms enables flexible deployment.

The global illumination ray-tracer is the main on-set renderer that enables the director and other professionals to judge and experiment with realistic lighting conditions. The implementation supports three progressive Monte Carlo simulation methods [PGTM16]: Path Tracing, Bidirectional Path Tracing, and Vertex Connection and Merging. Choosing the most suitable method depends on the scene characteristics as well as the requirements on real-time rendering quality and overall convergence speed.

Global illumination consumes a lot of processing power. To enable interactive performance already with standard hardware, the renderer produces noisy preliminary images during real-time rendering and iteratively converges to the final image during progressive rendering. But the client can increase the real-time rendering quality indefinitely by choosing the number of ray samples per pixel. We therefore recommended to run the renderer in a cluster with a dedicated CPU and GPU setup to enable a higher number of samples and also speed up progressive refinement. The ultimate goal is to provide noise-free production-level quality already during real-time rendering, which requires a high-performance cluster of a magnitude that was not available to us.

7.5.6 Load Balancing

Ray-tracing is an embarrassingly parallel problem that allows to divide the image space among the processing units. Since the ray-tracing workload can be heterogeneous, a load balancer that keeps the workers busy is required to achieve linear scalability. Chapter 6 outlines the existing load balancing methods.

Dynamic load balancers initially assign tasks of possibly varying cost to the workers. When a worker becomes idle, it receives tasks that are still outstanding. The worker either requests tasks from a central queue or attempts to steal from other workers.

Under optimal conditions, dynamic approaches scale linearly and naturally handle heterogeneous workers. However, they induce communication and task management overhead during rendering, which increases with the number of workers. A low-latency link between the master and the workers and in case of work stealing between all workers is essential. To achieve linear speed-ups, the load balancer must split the frame into tasks of fine granularity. Otherwise, in the end of the frame, some workers might stall the rendering when busy with a demanding task while there are no more tasks left for the idle workers. Dynamic methods are therefore ideally suitable for local thread-level scheduling on a single machine with a moderate amount of CPUs.

The rendering back-end supports a commodity network setup that does not guarantee a low-latency link between the nodes. Also, the back-end targets a high number of rendering nodes to ultimately enable high-quality global illumination at interactive frame rates. The global illumination ray-tracer utilizes the GPU. Dynamically assigning or shifting small tasks is inefficient due to the transfer overhead between CPU and GPU and between the nodes.

The rendering back-end consequently uses a static load balancer. A static approach assigns a fixed task per frame to each node. There is no communication overhead during rendering. To

achieve linear scalability, the load balancer must find tasks to equalize the rendering time on the nodes.

Chapter 6 presents a static approach that exploits frame-to-frame coherence in the real-time context. Based on timings acquired for the previous frame, the method derives a balanced task distribution for the next frame. However, the fine-grained timing mechanism is not available in a GPU environment.

We instead use the classic approach to static load balancing: a pseudo-random scattering of pixels or small pixel blocks among nodes, which can achieve an even cost distribution. The disadvantage is the loss in cache locality as each node works across the entire image. Scattering easily integrates with any ray-tracer, while work stealing requires an invasive adaptation of the renderer to allow handing back tasks from the renderer's internal scheduling to the network and vice versa. This conflicts with our design goal of a flexible architecture that allows arbitrary third-party renderers to plug in and benefit from distributed execution.

The master node runs the scattering load balancer once in the beginning of each rendering session. The tasks do not change even between frames, so there is no further load balancing overhead. The scattering operates on pixel blocks that are sized to enable SIMD instructions. The larger the block size, the better the renderer on each node can benefit from cache locality. On the other hand, a higher granularity can enable a more accurate load balance. Since the impact of both effects is scene- and view-dependent, the operator can select the granularity in a configuration file to fine tune the performance. Figure 7.6 illustrates the distribution of the image space among a set of nodes.

The current limitation of the load balancer is that it does not account for heterogeneous nodes. We therefore recommend to use a cluster with homogeneous nodes to achieve linear scalability. In a future revision, we plan to incorporate the weighing of nodes based on their capability.

To also distribute the rasterizer, the master assigns one coherent tile to each node. While this was not necessary for the scenarios we encountered in Dreamspace, a rasterizer may also benefit from distributed execution at very high resolutions due to the reduced pixel shading cost per node and for large scenes due to the ability to quickly cull geometry outside the view.

7.6 Results

The first part of this section presents performance measurements for the distributed rendering architecture. The second part evaluates the usage of the solution during a real-world, multi-day test production that utilized the technologies developed in Dreamspace.

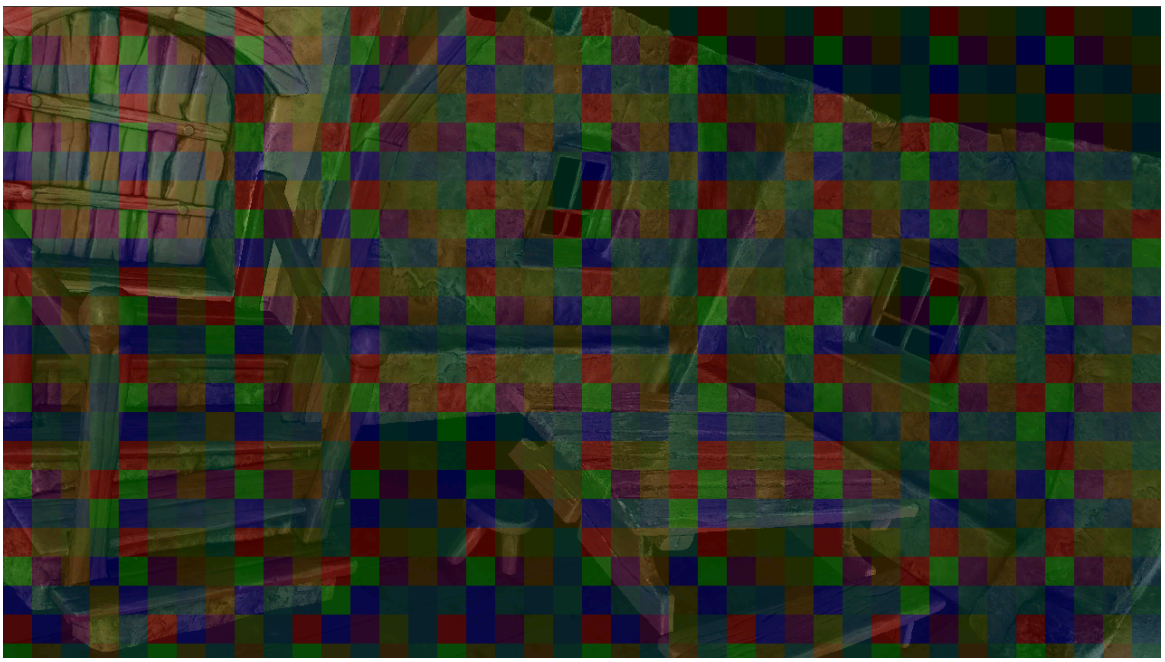


Figure 7.6: In this example, the load balancer splits a 720p image into blocks of 32x32 pixels and scatters the blocks to ten nodes. Each color indicates one node.

7.6.1 Timings

We used a cluster that consists of ten nodes connected via IPoIB with 10 GBit/s. Each node has two Intel Xeon X5650 six-core processors. Since the nodes do not have GPUs, we preferred the direct over the global illumination ray-tracer. On the display side, we ran LiveView and the web client in the latest Chrome browser. The connection to the cluster is 1 GBit/s Ethernet.

The image resolution is 1920x1080. The cluster employs distributed S3TC encoding. The master sends the final S3TC image directly to the client.

The test scene is the San Miguel hacienda, which is the main reference scene of Dreamspace. We already used the scene in Chapter 6. The hacienda has 7.7 million triangles and contains many parts with heterogeneous rendering cost. There are many small leafs, glasses, bottles, and a water fountain. The scene has seven lights.

To compare the results for different node counts, we recorded a camera movement through the scene and replayed it for each run. The following tables present average measurements over all frames of a run.

7.6.1.1 Scalability

The scalability tests involve the following per-frame measurements on each rendering node.

- **Kernel**: The total ray-tracing cost spent across all threads. The kernel cost does not include the thread management overhead attributed to the local work stealer that runs on each node. The measurement is therefore best suitable to show the efficiency of the cluster-level scattering load balancer.
- **Rendering**: The time to render the frame. Since the kernel executes on multiple cores, this value includes the thread management overhead.

Table 7.1 shows that both kernel cost and rendering time show a strong scaling efficiency. Figure 7.7 illustrates the linear performance increase.

Table 7.1: The scaling efficiency and its coefficient of variation (CV) for different node counts with reference to the single node performance.

Single Node Rendering: 854.8 ms

	2	3	4	5	6
Kernel	98.7%	97.5%	97.7%	97.4%	97.3%
Kernel CV	0.002	0.003	0.003	0.004	0.004
Rendering	98.6%	97.3%	97.3%	96.7%	96.5%
Rendering CV	0.007	0.007	0.011	0.014	0.014
	7	8	9	10	
Kernel	96.9%	96.8%	96.9%	96.8%	
Kernel CV	0.005	0.006	0.006	0.006	
Rendering	95.7%	95.5%	95.1%	94.9%	
Rendering CV	0.02	0.019	0.025	0.025	

The rendering time includes overhead attributed to the multi-threaded execution. Since the overhead is about constant, it has a higher impact as the kernel cost per node decreases with a larger cluster. Therefore, the scaling efficiency for the rendering time is expectedly slightly below the kernel and tends to drop gradually with more nodes.

The low coefficient of variation demonstrates that the high efficiency stays stable over the entire runs. The coefficient is even lower for the kernel cost due to the multi-threading being no factor that can cause fluctuation or outliers. The result is a smooth user experience without noticeable slowdowns.

As the node count increases, the chance for fluctuation or an outlier on any node increases, which causes the CV to slightly go up with more nodes. The effect can also cause the scaling efficiency to slightly drop with more nodes independent of the accuracy of the load balancer.

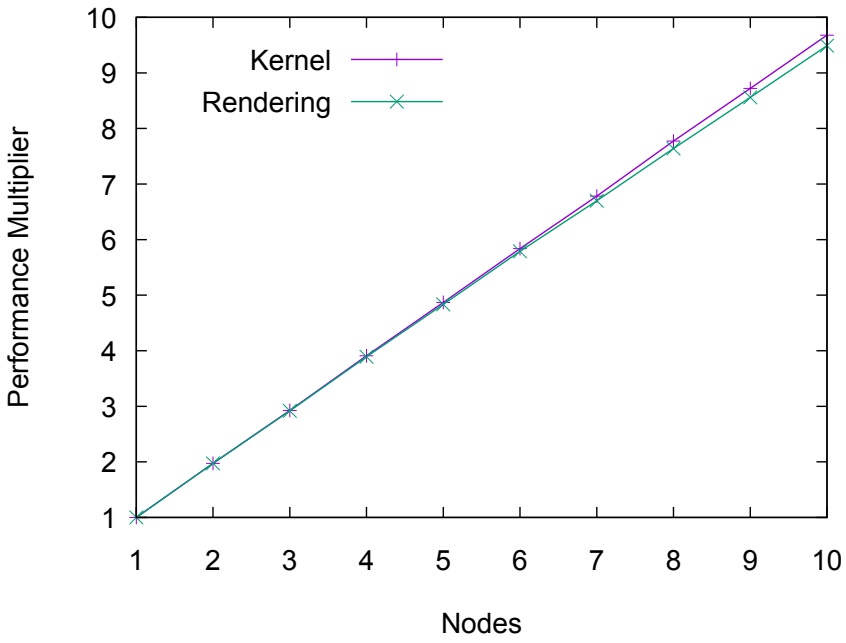


Figure 7.7: The kernel and the rendering performance increase as a function of the node count.

7.6.1.2 Pipeline

The pipeline tests involve the following per-frame measurements.

On each Rendering Node

- **Encoding**: The time to encode the sub-image to be sent to the master node.

Master Node

- **Framework**: The time from starting the frame until having sent the final image to the display client excluding the rendering time. This measurement includes the encoding and the intra-cluster image transfer.

Display Client

- **Network**: The time from requesting the rendering of the frame until having received the final image for display excluding the cluster framework and rendering time.

Table 7.2 shows the minimal latency the framework induces in addition to the rendering.

The encoding time is negligible even for a single node and decreases further with more nodes due to the distributed encoding. Since the cluster network transports already encoded image data,

Table 7.2: The pipeline measurements in milliseconds for the different node counts.

	1	2	3	4	5
Encoding per Node	0.607	0.336	0.256	0.211	0.176
Cluster Framework	1.259	1.64	1.368	1.215	1.235
Web Client Network	13.38	13.18	13.19	13.16	13.55
LiveView Network	9.06	9.07	9.1	9.09	9.26
	6	7	8	9	10
Encoding per Node	0.155	0.148	0.135	0.123	0.116
Cluster Framework	1.376	1.42	1.512	1.52	1.577
Web Client Network	13.03	13.15	13.27	13.19	13.34
LiveView Network	9.08	9.09	9.11	9.29	9.29

the framework overhead stays consistently below two milliseconds despite the high-definition 1080p image resolution. The overhead is even lower than the one presented in Chapter 6 for 720p images. The framework time tends to increase slightly with more nodes. The master participates in the rendering. As the task size assigned to the master decreases with more nodes, the network load increases.

The time to transport the final image to the display client is higher due to the 1 GBit/s standard network. For the web client, we attribute some of the delay to the browser environment, in particular to the JavaScript WebSocket API. We measured a clearly lower delay with the native LiveView client.

Due to the asynchronous execution that allows the renderers to process the next frame while the current one is on its way to the master and client, the framework and network time does not affect the frame rate on the display side as long as the overhead is below the rendering time per node.

7.6.2 Test Production

We evaluated the rendering architecture in the *Battleground* test production [SCM+16, HSK+16], which was conducted by Stargate Studios in collaboration with the other Dreamspace partners in a studio of the Filmakademie Ludwigsburg, Germany. The goal was to test the different Dreamspace technologies together in a representative scenario under a typical timescale. The production involved around twelve team members and spanned five days including a final day for visitors. The visitors included visual effects companies from the area as well as the

minister of education and cultural affairs of Baden-Württemberg, Germany.

The production created a teaser for a new type of interactive virtual reality TV show where two teams of real contestants control computer-generated robots that fight in a virtual battleground. The distributed rendering framework was used to provide preview and high-quality renderings while shooting, thus allowing to coordinate the final rendering already before post-production. Figure 7.8 shows photographs from the production.

Figure 7.8: Impressions from the battleground test production. On the top left, a team member uses a tablet to manipulate the scene rendered by the global illumination ray-tracer.

During the first two days, the set was prepared and the hardware deployed. Camera and depth tracking, light capturing, and tablet editing tools were available and connected to the main

LiveView client. LiveView also connected to the rendering cluster, which consisted of four commodity machines. Each machine was equipped with an NVIDIA GTX 970 GPU. Three of the machines had an Intel i7-4790 processor. The last machine, which we used as the master node, had an Intel i7-6700 processor. The network setup between all machines including the LiveView client was 1 GBit/s Ethernet.

The setup of the cluster proved to be straightforward. The self-contained installation package allowed us to quickly launch the rendering server on each node. Due to the node discovery feature, the master could find the other nodes automatically. In addition, we launched a standard HTTP server on the LiveView machine to make the scene available to the rendering back-end.

On the third day, the shooting of the first setting happened, which is the preparation room where the teams configure their robot before entering the arena. The setting consisted of a rendered robot as the foreground composited with the live camera feed.

The fourth day was the production effort for the main setting, where the two teams control their robots via motion capture to fight each other in the arena. The scenario was the central use case for the distributed rendering architecture. Throughout the shooting, the professionals switched regularly from LiveView's native rasterizer to the global illumination ray-tracer to review more realistic rendering results.

Using the limited cluster, the framework could not provide production-level image quality in real-time. However, the low-budget setup already was able to provide interactive globally illuminated previews in 1080p resolution at around 20 FPS. Progressive rendering quickly reduced the noise artifacts for static views to allow a more accurate judgment of the lighting conditions. Stargate Studio's review [HSK+16] states a substantially improved rendering performance compared to V-Ray.

The professionals edited the base scene frequently in Maya and Katana and re-exported it from LiveView to the cluster. After the caching procedure, the plugin could display the scene instantly for subsequent loadings. This was an advantage over LiveView's local renderer that currently performs a lengthy traversal of the scene graph each time. The plugin and the server side responded to run-time updates generated by the camera tracking, light capturing, and tablet editing tools.

Alternatively to LiveView, the portable web client proved to be popular to access the rendering back-end at various stages in the production. The web client was also used to demonstrate the renderer during the final visiting day.

There were minor issues with the matching of the lighting between LiveView's native rasterizer,

the production renderer Arnold, and our renderer. However, the inconsistency is caused by a missing global, physically-based specification of the lighting parameters in LiveView and thus renderer-specific interpretations. Other feedback states that the set up and also run-time calibration of the cluster and the rendering could be further improved, for example by introducing management software and UI-based controls.

Overall, the rendering architecture proved to be stable and easy to use for the non-technical professionals. Feedback indicates that the experimentation with realistic rendering quality is beneficial to prepare and already work towards the post-production. On the other hand, in the fast-paced on-set environment, trading quality for faster response times was often preferred to facilitate quick and smooth experimentation. These heterogeneous demands fit our flexible framework, which supports different levels of rendering service and scales in a cluster environment to ideally achieve both high quality and performance.

7.7 Conclusion and Future Work

This chapter presented the Dreamspace distributed rendering architecture for virtual production. The architecture provides interactive, high-quality renderings of the virtual scene in the on-set environment. The goal is to narrow the gap to post-production by enabling creative control of the visual effects already while filming.

The client side of the architecture interfaces with traditional pre-production modeling tools to receive the scene data. On the set, the client receives live updates from other components of the Dreamspace pipeline: the camera tracking, the light calibration, and the tablet editing tools. The client forwards the scene and incremental updates to the rendering back-end. We also provide a portable web client to allow mobile access to the visualization and a client running in Blender.

The server side of the architecture is a flexible and scalable solution that runs in a cluster setup. The back-end supports different renderers including a global illumination ray-tracer for realistic image quality. To enable the deployment under different production budgets, the framework is already operational with commodity hardware. We successfully evaluated the solution in a real-world test production that combined the technologies developed in Dreamspace.

While the main intention of the architecture is real-time rendering on the set, the high-performance distributed global illumination can also benefit the offline rendering in post-production. Being able to cut the rendering time by only a few percent could already translate to major savings considering final quality rendering still may take hours per frame. However,

our renderer does currently not support the full set of features required for a post-production system.

Since the framework is flexible and allows to plug in any renderer, we plan to investigate use cases other than film production. The integration of real-time ray-tracing in games is of particular interest. To increase the availability, we intend the integration in other client software like Maya and 3ds Max as well as the Dreamspace on-set editing tools.

The Dreamspace pipeline allows experimentation with the real and virtual scene on the set. A crucial feature is the ability to store and reproduce the run-time changes and transport them seamlessly to the post-production phase for final adjustments. Within Dreamspace, we could not address this aspect thoroughly. A major future goal is therefore to efficiently translate incremental updates back to disk and port the modified scene to post-production software.

Chapter 8

Conclusion

8.1 Summary of Achievements

In this thesis we set out to develop a comprehensive and representative set of rendering architectures that utilize the available client, network, and server-side resources to bring scalable and consistent 3D experiences to heterogeneous computing environments and display setups. We thereby widely covered the problem statements we originally formulated in Chapter 1.

The term "Ubiquitous 3D" encompasses a myriad of use cases and application areas, which we could not explicitly address in their entirety within the thesis. We instead focused on a wide coverage of the display setups and platforms that visualization applications can target today, ranging from mobile devices to desktop and distributed displays as well as from native platforms to the web browser as a portable execution environment. We utilize both client and server-side resources to enable the desired level of quality, performance, and scalability for the target applications and the associated display devices.

Ultimately, the described architectures enable to develop various kinds of applications, and we tested our technology in the major areas of scientific and medical visualization, web-based 3D applications, and virtual film production. The practicability of the solutions is not limited to specific use cases, since our frameworks have been designed with general concepts in mind and thus provide generic means to plug in different renderers, distribute data and rendering results, and, in case of ZAPP and the XML3D-enabled frameworks, implement custom application logic. We collaborated with experts from the medical field and the movie industry to transfer our technology from the research realm to real-world usage.

The architectures described in this thesis build on an innovative utilization and combination of technologies as well as on new methods and algorithms to enable the efficient utilization

of the heterogeneous computing resources. We incorporated the major rendering techniques volume rendering, rasterization, and ray-tracing. We employ local, server-based, hybrid, and distributed rendering and present results on mobile, desktop, and distributed displays. Using the browser as the client-side execution platform, we provide an accessible and portable user interface. Given the broad scope of the thesis, an indirect contribution is also the wide literature review across the different architectures.

The following paragraphs reiterate the contributions of each chapter in a summarized form.

In Chapter 2, we presented the ZAPP framework to develop and manage visualization applications for distributed and tiled display setups. In contrast to previous solutions, ZAPP lays its focus on the usability side and provides accessible interfaces to install and maintain the framework as well as to develop and run applications. We developed a distributed visualization solution for multi-resolution data sets as a central ZAPP example application.

In Chapter 3, we presented the mobile visualization application IV3Dm and its deployment to aid in the selection of parameters for DBS treatment of PD patients. A major component of the setup is the data distribution and management framework that utilizes instant messaging technology to simplify the data retrieval for the users. Using the mobile devices the clinicians are accustomed to in combination with the simplified, one-tap data transfer mechanism, we achieved a minimally invasive integration into the clinical workflow and thus a high acceptance of the system. The evaluation conducted with real clinicians and patient data demonstrates the potential for large time savings to determine the DBS parameters compared to the current standard of care.

In Chapter 4, we combined hybrid rendering and scheduling under uncertainty. We presented a hybrid rendering method that utilizes both client and server in the visualization of large, multi-resolution data sets in dynamic environments. Given a separation of a data set into multiple levels of quality that are consecutively rendered, the goal is to provide interactive performance for at least the lowest level and to reach the highest quality as fast as possible. The probabilistic scheduler obtains performance timings at run-time for each QL to determine which QLs to render on which side. The scheduler is able to adapt to initially unknown and possibly changing conditions. We demonstrated the advantage of the approach over pure remote rendering and over deterministic scheduling.

In Chapter 5, we established the browser as a viable, promising platform to implement accessible and portable visualization applications. We described, implemented, and tested the state-of-the-art methods that enable plugin-free remote rendering in the browser, which includes the first utilization of WebRTC in that context. We presented a novel remote visualization system in the browser that combines all methods and is thus highly adaptable to different application

requirements, target platforms, and network conditions.

In Chapter 6, we presented the first extension of a framework for declarative 3D in the web with server-based rendering. We choose XML3D, which is a HTML5 extension and JS library that enables the common web developer to create arbitrary 3D applications in the browser. On the server side, we presented a distributed rendering framework that supports a hierarchy of nodes and different types of renderers. To run our custom real-time ray-tracer in this architecture, we developed a static load balancing methods that builds on temporal coherence between adjacent frames during real-time rendering. Unlike previous related approaches that ultimately need to incorporate a dynamic load balancer to counteract inaccuracies, our method achieves a stable linear scalability and can even compete with dynamic load balancers for local thread-level scheduling.

In Chapter 7, we presented a novel virtual production platform that provides interactive renderings of the virtual scene composited with the real world already on the set. Our contribution to this platform is a highly flexible and efficient distributed rendering framework that interfaces with other components of the production pipeline. While existing production-level solutions are limited to specific renderers or require dedicated hardware, our solution supports multiple types of rendering service, including interactive global illumination and web-based access, and is already operational with commodity hardware and networking. We successfully evaluated the system in a real-world, multi-day film studio test production.

8.2 Future Work

Since we wrote each chapter as an experiment that can stand on its own, we already presented specific future directions at the end of each chapter. Here, we focus on the overall scope and direction the work of this thesis could evolve into.

The ultimate vision is one overall, generic system that provides a compute continuum for ubiquitous 3D. The thesis developed a strong foundation for this long-term goal. The frameworks developed in the thesis address a wide range of computing architectures, display setups, and potential applications. We can identify several major components that are heavily re-utilized across the rendering architectures. The visualization applications from Chapter 3 to 5 share the same rendering library, which enables local, server-based, and hybrid rendering of multi-resolution data sets. The distributed visualization application for the ZAPP framework also builds on the library. The image transport methods introduced in Chapter 5 resurface in the distributed rendering frameworks and the corresponding clients from Chapter 6 and 7. The distributed rendering framework presented in Chapter 7 builds on the solution from Chapter 6

and is also compatible with the XML3D browser client. Consequently, a central future task is to unify the comprehensive range of related concepts and technologies presented in the thesis into a single, generic framework capable to cover the various application requirements.

One aspect that is especially challenging for a system that supports different types of rendering service is a common ground for material descriptions, shading parameters, and custom shaders. The available material and shading parameters and also the shading functionality may vary considerably across different rendering applications, algorithms, and target platforms. Sons et al. [SKSS14] present the shade.js system, which provides application-independent material descriptions and an environment-aware shading language. Shaders can inspect their environment to adapt to different conditions. A run-time compiler then produces executable code specialized according to the actual conditions. Since shade.js has already been integrated into XML3D, which is one of our main client solutions, we consider to use the concept as the basis for a unified shading system.

The distributed rendering solutions we presented are designed for maximum performance. However, the clusters that were available to us for the implementation and testing are limited in size or do not offer up-to-date hardware. This especially limited the quality we could achieve for the interactive global illumination ray-tracer. A future step is to run the distributed rendering on a dedicated, high-end cluster infrastructure. We were already able to generate and run an initial build of the framework and global illumination ray-tracer from Chapter 7 on the Hazel Hen supercomputer [HLR16] at the High-Performance Computing Center in Stuttgart, Germany. To see how the solution scales to hundreds or even thousands of nodes is an interesting prospect. Though, the project is in an early stage and performance tests have yet to be carried out.

Bibliography

[A-F17] A-Frame authors. A-Frame, 2017. https://aframe.io.

[ADD+07] Matt Aranha, Piotr Dubla, Kurt Debattista, Thomas Bashford-Rogers, and Alan Chalmers. A physically-based client-server rendering solution for mobile devices. In *Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*, MUM '07, pages 149–154, New York, NY, USA, 2007. ACM.

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[Ama16] Amazon Web Services, Inc. Amazon EC2, 2016. https://aws.amazon.com/ec2.

[App16a] Apple. Metal for Developers, 2016. https://developer.apple.com/metal.

[App16b] Apple. Notifications, 2016. https://developer.apple.com/notifications.

[Art15] Art and Animation studio. FurryBall GPU renderer, 2015. http://furryball.aaa-studio.eu.

[Aut16] Autodesk Inc. VRED, 2016. http://www.autodesk.com/products/vred/overview.

[aut17a] FFmpeg authors. FFmpeg, 2017. https://www.ffmpeg.org.

[aut17b] Libav authors. Libav, 2017. https://libav.org.

[BA15] R. Bundulis and G. Arnicans. Use of H.264 real-time video encoding to reduce display wall system bandwidth consumption. In *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in*, pages 1–6, Nov 2015.

[BBB+14] Marco Bertoni, Alessandro Bertoni, Henk Broeze, Gilles Dubourg, and Clive Sundhurst. Using 3D CAD models for value visualization: an approach with SIEMENS

NX HD3D visual reporting. *Computer-Aided Design and Applications*, 11(3):284–294, 2014.

[BBQL16] Jens Buergin, Philippe Blaettchen, Chuanqi Qu, and Gisela Lanza. Assignment of customer-specific orders to plants with mixed-model assembly lines in global production networks. *Procedia CIRP*, 50:330 – 335, 2016. 26th CIRP Design Conference.

[BC05] Stefano Burigat and Luca Chittaro. Location-aware visualization of VRML models in GPS-based mobile guides. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, pages 57–64, New York, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1050491.1050499.

[BCH+11] C. R. Butson, S. E. Cooper, J. M. Henderson, B. Wolgamuth, and C. C. McIntyre. Probabilistic analysis of activation volumes generated during deep brain stimulation. *NeuroImage*, 54:2096–2104, 2011. http://www.ncbi.nlm.nih.gov/pubmed/20974269.

[BCH12] E. Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. Chapman & Hall/CRC, 1st edition, 2012.

[BCHM06] C. R. Butson, S. E. Cooper, J. M. Henderson, and C. C. McIntyre. Predicting the effects of deep brain stimulation with diffusion tensor based electric field models. *Med Image Comput Comput Assist Interv*, 9 (Pt 2):429–437, 2006.

[BCHM07] C. R. Butson, S. E. Cooper, J. M. Henderson, and C. C. McIntyre. Patient-specific analysis of the volume of tissue activated during deep brain stimulation. *NeuroImage*, 34(2):661–70, 2007. http://linkinghub.elsevier.com/retrieve/pii/S1053811906009669.

[BDH+10] C. R. Butson, K Driesslein, S.W. Hung, M.A. Matthews, C. J. Sheridan, B. H. Kopell, and J.A. Bobholz. Probabilistic atlas of neuropsychological outcomes from subthalamic nucleus deep brain stimulation. In *Movement Disorders Society Congress*, 2010.

[BDWS02] Carsten Benthin, Tim Dahmen, Ingo Wald, and Philipp Slusallek. Interactive headlight simulation: A case study of interactive distributed ray tracing. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pages 83–88, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[BEJZ09] Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. X3DOM: a DOM-based HTML5/X3D integration model. In *Proceedings of the 14th International*

Conference on 3D Web Technology, Web3D '09, pages 127–135, New York, NY, USA, 2009. ACM.

[BG01] J. Balasubramanian and I. E. Grossmann. A novel branch and bound algorithm for scheduling flowshop plants with uncertain processing times. *Computers and Chemical Engineering*, 26:4–1, 2001.

[BGR03] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, January 2003.

[BIHGO16] Henk Broekhuizen, Maarten J. IJzerman, A. Brett Hauber, and Catharina G. M. Groothuis-Oudshoorn. Weighing clinical evidence using patient preferences: An application of probabilistic multi-criteria decision analysis. *PharmacoEconomics*, pages 1–11, 2016.

[Bik07] Jacco Bikker. Real-time ray tracing through the eyes of a game developer. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 1–10, Washington, DC, USA, 2007. IEEE Computer Society.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[Ble17] Blender Foundation. Home of the Blender project - free and open 3D creation software, 2017. `https://www.blender.org`.

[BM05] C. R. Butson and C. C. McIntyre. Tissue and electrode capacitance reduce neural activation volumes during deep brain stimulation. *Clin Neurophysiol*, 116(10):2490–500, 2005. 1388-2457.

[BM06] C.R. Butson and C. C. McIntyre. Role of electrode design on the volume of tissue activated during deep brain stimulation. *J Neural Eng*, 3(1):1–8, 2006.

[BM07] C. R. Butson and C. C. McIntyre. Differences among implanted pulse generator waveforms cause variations in the neural response to deep brain stimulation. *Clin Neurophysiol*, 118(8):1889–94, 2007.

[BM08] C. R. Butson and C. C. McIntyre. Current steering to control the volume of tissue activated during deep brain stimulation. *Brain Stimulation*, 1(1):7–15, 2008.

[BMM06] C. R. Butson, C. B. Maks, and C. C. McIntyre. Sources and effects of electrode impedance during deep brain stimulation. *Clin Neurophysiol*, 117(2):447–54, 2006.

[BMP+15] Johannes Behr, Christophe Mouton, Samuel Parfouru, Julien Champeau, Clotilde Jeulin, Maik Thöner, Christian Stein, Michael Schmitt, Max Limper, Miguel de Sousa, Tobias Alexander Franke, and Gerrit Voss. webVis/Instant3DHub: Visual computing as a service infrastructure to deliver adaptive, secure and scalable user centric data visualisation. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 39–47, New York, NY, USA, 2015. ACM.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[BN05] Robert Ball and Chris North. Effects of tiled high-resolution display on basic visualization and navigation tasks. In *CHI '05 extended abstracts on Human factors in computing systems*, CHI EA '05, pages 1196–1199, New York, NY, USA, 2005. ACM.

[Boi16] Boivin, Sam. Short range depth maps. Technical report, ncam, 2016. `http://www.dreamspaceproject.eu/Docs`.

[Bom13] Boman, B. and Isomaki, M. and Aboba, B. and Martin-Cocher, G. and Mandyam, G. and Marjou, X. H.264 as mandatory to implement video codec for WebRTC, 2013. `http://tools.ietf.org/id/draft-burman-rtcweb-h264-proposal-01.html`.

[BRE05] Praveen Bhaniramka, P. C. D. Robert, and S. Eilemann. OpenGL multipipe SDK: a toolkit for scalable parallel rendering. In *VIS 05. IEEE Visualization, 2005.*, pages 119–126, Oct 2005.

[Bri11] Bringuier, L. OTT streaming, September 2011. `http://www.bogotobogo.com/VideoStreaming/images/mpeg_dash/Anevia_White-Paper_OTT-Streaming_2nd_Edition.pdf`.

[BTJ+13] Christopher Butson, Georg Tamm, Sanket Jain, Thomas Fogal, and Jens Krüger. Evaluation of interactive visualization on mobile computing platforms for selection of deep brain stimulation parameters. *IEEE Transactions on Visualization and Computer Graphics*, 19:108–117, 2013.

[BTl+00] Wes Bethel, Brian Tierney, Jason lee, Dan Gunter, and Stephen Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[Bux07] Bill Buxton. Multi-touch systems that I have known and loved, Jan 2007. `http://www.billbuxton.com/multitouchOverview.html`.

[BWDS02] Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek. Interactive headlight simulation - a case study for distributed interactive ray tracing. In Dirk Bartz, Xavier Pueyo, and Eric Reinhard, editors, *Proceedings of EUROGRAPHICS Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88. Saarland University, 2002.

[CBPS06] Steven P. Callahan, Louis Bavoil, Valerio Pascucci, and Claudio T. Silva. Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1307–1314, September 2006.

[CCDC+08] B. Cosenza, G. Cordasco, R. De Chiara, U. Erra, and V. Scarano. Load balancing in mesh-like computations using prediction binary trees. In *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on*, pages 139–146, July 2008.

[CCT+11] Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 1269–1272, New York, NY, USA, 2011. ACM.

[CDE13] Biagio Cosenza, Carsten Dachsbacher, and Ugo Erra. GPU cost estimation for load balancing in parallel ray tracing. In *International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 139–151, 2013.

[CDR02] Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[CG02] Chun-Fa Chang and Shyh-Haur Ger. Enhancing 3D graphics on mobile devices by image-based rendering. In *PCM í02: Proceedings of the Third IEEE Pacific Rim Conference on Multimedia*, pages 1105–1111. Springer-Verlag, 2002. `http://www.cs.nthu.edu.tw/~chunfa/pcm2002.pdf`.

[Cha17] Chaos Software. Rendering & simulation software - V-Ray, VRscans & Phoenix FD, 2017. `https://www.chaosgroup.com`.

[Chi06] Luca Chittaro. Visualizing information on mobile devices. *Computer*, 39(3):40–45, 2006. http://dx.doi.org/10.1109/MC.2006.109.

[CHS+12] Per H. Christensen, George Harker, Jonathan Shade, Brenden Schubert, and Dana Batali. Multiresolution radiosity caching for global illumination in movies. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, pages 47:1–47:1, New York, NY, USA, 2012. ACM.

[Chu06] Chu, J. and Kashyap, V. Transmission of IP over InfiniBand (IPoIB), 2006. https://tools.ietf.org/html/rfc4391.

[CJM97] G. E. Christensen, S. C. Joshi, and M. I. Miller. Volumetric transformation of brain anatomy. *IEEE Trans Med Imaging*, 16(6):864–77, 1997.

[CKS02] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[CR10] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, April 2010.

[CS11] Scientific Computing and Imaging Institute (SCI). SCIRun: A scientific computing problem solving environment, 2011. http://www.scirun.org.

[CSK+11] John Congote, Alvaro Segura, Luis Kabongo, Aitor Moreno, Jorge Posada, and Oscar Ruiz. Interactive visualization of volumetric data with WebGL in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pages 137–146, New York, NY, USA, 2011. ACM.

[Deu96] Deutsch, P. DEFLATE compressed data format specification, 1996. https://www.ietf.org/rfc/rfc1951.txt.

[DGE04] Joachim Diepstraten, Martin Gorke, and Thomas Ertl. Remote line rendering for mobile devices. In *Proceedings of the Computer Graphics International*, CGI '04, pages 454–461, Washington, DC, USA, 2004. IEEE Computer Society.

[DGP04] David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker. Memory-savvy distributed interactive ray tracing. In *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '04, pages 93–100, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[DHK12] Juergen Doellner, Benjamin Hagedorn, and Jan Klimke. Server-based rendering of large 3D scenes for mobile devices using G-buffer cube maps. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 97–100, New York, NY, USA, 2012. ACM.

[DJX14] Jonathan Ding, Jennifer Jin, and Samuel Xu. A journey towards a compute continuum: Client-aware cloud services for smart clients. *Service Technology Magazine*, LXXXII, 2014.

[DK11] Kai-Uwe Doerr and Falko Kuester. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):320–332, May 2011.

[DKHS14] Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. Progressive light transport simulation on the GPU: Survey and improvements. *ACM Trans. Graph.*, 33(3):29:1–29:19, June 2014.

[DKW09] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[DLS+12] C. Dyken, K.O. Lye, J. Seland, E.W. Bjonnes, J. Hjelmervik, J.O. Nygaard, and T.R. Hagen. A framework for OpenGL client-server rendering. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 729–734, 2012.

[Don10] Donovan, A. and Muth, R. and Chen, B. and Sehr, D. PNaCl: Portable Native Client executables, February 2010. `http://www.chromium.org/nativeclient/reference/research-papers`.

[DSBK+06] G. Deuschl, C. Schade-Brittinger, P. Krack, J. Volkmann, H. Schafer, K. Botzel, C. Daniels, A. Deutschlander, U. Dillmann, W. Eisner, D. Gruber, W. Hamel, J. Herzog, R. Hilker, S. Klebe, M. Kloss, J. Koy, M. Krause, A. Kupsch, D. Lorenz, S. Lorenzl, H. M. Mehdorn, J. R. Moringlane, W. Oertel, M. O. Pinsker, H. Reichmann, A. Reuss, G. H. Schneider, A. Schnitzler, U. Steude, V. Sturm, L. Timmermann, V. Tronnier, T. Trottenberg, L. Wojtecki, E. Wolf, W. Poewe, and J. Voges. A randomized trial of deep-brain stimulation for Parkinson's disease. *N Engl J Med*, 355(9):896–908, 2006. `http://www.ncbi.nlm.nih.gov/pubmed/16943402`.

[EEH+00] K. Engel, T. Ertl, P. Hastreiter, B. Tomandl, and K. Eberhardt. Combining local and remote visualization techniques for interactive volume rendering in medical

applications. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 449–452, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[EFK95] J. L. Encarnação, M. Frühauf, and T. Kirste. Mobile visualization: Challenges and solution concepts. In *In Proc. CAPE'95*, 1995.

[EG15] Farshad Einabadi and Oliver Grau. Discrete light source estimation from light probes for photorealistic rendering. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 43.1–43.10. BMVA Press, September 2015.

[EMP09] Stefan Eilemann, Maxim Makhinya, and Renato Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, May 2009.

[EOEI00] Klaus Engel, Frank Oellien, Thomas Ertl, and Wolf-Dietrich Ihlenfeldt. Client-server-strategien zur visualisierung komplexer struktureigenschaften in digitalen dokumenten der chemie. *it+ti - Informationstechnik und Technische Informatik*, 42(6):17–23, 2000.

[Evo16] EvoStream. EvoStream, 2016. `https://evostream.com`.

[EWE99] Klaus Engel, Rüdiger Westermann, and Thomas Ertl. Isosurface extraction techniques for web-based volume visualization. In *Proceedings of the conference on Visualization '99: celebrating ten years*, VIS '99, pages 139–146, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[exa10] Report of the advanced scientific computing advisory committee (ascac) subcommittee, u.s. department of energy. The Opportunities and Challenges of Exascale Computing, 2010. `https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf`.

[exa11] Office of advanced scientific computing research (ascr), u.s. department of energy. Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, 2011. `https://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf`.

[FCS+10] Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large data visualization on distributed memory multi-GPU clusters. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 57–66, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[FE10] P. Fechteler and P. Eisert. Accelerated video encoding using render context information. In *2010 IEEE International Conference on Image Processing*, pages 2033–2036, Sept 2010.

[Fen03] Simon Fenney. Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 84–91, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FGD+06] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. Exploring the use of ray tracing for future games. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, pages 41–50, New York, NY, USA, 2006. ACM.

[FHF+17] Luca Fascione, Johannes Hanika, Marcos Fajardo, Per Christensen, Brent Burley, and Brian Green. Path tracing in production - part 1: Production renderers. In *ACM SIGGRAPH 2017 Courses*, SIGGRAPH '17, pages 13:1–13:39, New York, NY, USA, 2017. ACM.

[FHK97] B. Freisleben, D. Hartmann, and T. Kielmann. Parallel raytracing: a case study on partitioning and scheduling on workstation clusters. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, volume 1, pages 596–605 vol.1, Jan 1997.

[Fin09] Tony Finch. Incremental calculation of weighted mean and variance, 2009. `https://www.researchgate.net/publication/266884380_Incremental_calculation_of_weighted_mean_and_variance`.

[FIW15] FIWARE. Synchronization - FiVES, 2015. `https://catalogue.fiware.org/enablers/synchronization-fives`.

[FK10] Thomas Fogal and Jens Krüger. Tuvok - an architecture for large scale volume rendering. In *Proceedings of the 15th Vision, Modeling and Visualization Workshop 2010*, 2010.

[FNM+14] A. Febretti, A. Nishimoto, V. Mateevitsi, L. Renambot, A. Johnson, and J. Leigh. Omegalib: A multi-view application framework for hybrid reality display environments. In *2014 IEEE Virtual Reality (VR)*, pages 9–14, March 2014.

[FWN+10] A. M. Frankemolle, J. Wu, A. M. Noecker, C. Voelcker-Rehage, J. C. Ho, J. L. Vitek, C. C. McIntyre, and J. L. Alberts. Reversing cognitive-motor impairments in Parkinson's disease patients using a computational modelling approach to deep brain stimulation programming. *Brain*, 133(Pt 3):746–61, 2010.

[GDC05] Richard Gillibrand, Kurt Debattista, and Alan Chalmers. Cost prediction maps for global illumination. In Louise M. Lever and Mary McDerby, editors, *EG UK Theory and Practice of Computer Graphics*. The Eurographics Association, 2005.

[GGH12] Jochen Grün, Tobias Gerber, and Thorsten Herfet. Multi-reality interfaces - remote monitoring and maintenance in modular factory environments. In Jochen Grün, Tobias Gerber, and Thorsten Herfet, editors, *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 12). IFAC Symposium on Information Control Problems in Manufacturing (INCOM-2012), May 23-25, Bucharest, Romania*. IFAC, IFAC, 2012.

[GGH13] J. Gruen, M. Gorius, and T. Herfet. Interactive RTP services with predictable reliability. In *2013 IEEE Third International Conference on Consumer Electronics Berlin (ICCE-Berlin)*, pages 371–375, Sept 2013.

[GHJ+16] Oliver Grau, Volker Helzle, Eric Joris, Thomas Knop, Brice Michoud, Philipp Slusallek, Philippe Bekaert, and Jonathan Starck. Dreamspace: A platform and tools for collaborative virtual production. In *Proceedings of the IBC conference 2016*, 2016.

[GKDS12] Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. Light transport simulation with Vertex Connection and Merging. *ACM Trans. Graph.*, 31(6):192:1–192:10, November 2012.

[Goo16] Google. webports, 2016. `https://chromium.googlesource.com/webports`.

[Goo17] Google. WebAssembly migration guide, 2017. `https://developer.chrome.com/native-client/migration`.

[GS08] I. Georgiev and P. Slusallek. RTfact: Generic concepts for flexible and high performance ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 115–122, Aug 2008.

[GSH12] M. Gorius, Y. Shuai, and T. Herfet. Dynamic media streaming under predictable reliability. In *IEEE international Symposium on Broadband Multimedia Systems and Broadcasting*, pages 1–6, June 2012.

[HA89] Chris Hendrickson and Tung Au. *Project Management for Construction*. Prentice Hall, 1989. http://pmbook.ce.cmu.edu.

[HA98] Alan Heirich and James Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing*, 12(1-2):57–68, 1998.

[Hag11] Tor-Magne Stien Hagen. *Interactive Visualization on High-Resolution Tiled Display Walls with Network Accessible Compute- and Display-Resources*. PhD thesis, University of Tromsø, Tromsø, 2011.

[Han06] Handley, H. and Jacobson, V. and Perkins, C. SDP: Session Description Protocol, 2006. https://tools.ietf.org/html/rfc4566.

[HBEH00] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[HEB+01] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 129–140, New York, NY, USA, 2001. ACM.

[HHN+02a] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 693–702, New York, NY, USA, 2002. ACM.

[HHN+02b] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002. http://doi.acm.org/10.1145/566654.566639.

[HIN99] Z. Hong, K.I. Iourcha, and K.S. Nayak. System and method for fixed-rate block-based image compression with inferred pixel values, September 21 1999. US Patent 5,956,431.

[HJNS+13] Mahdi Hemmati, Abbas Javadtalab, Ali Asghar Nazari Shirehjini, Shervin Shirmohammadi, and Tarik Arici. Game as video: bit rate reduction through adaptive object encoding. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '13, pages 7–12, New York, NY, USA, 2013. ACM.

[HKR+06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., 2006. http://www.real-time-volume-graphics.org.

[HLR16] HLRS. CRAY XC40 (HAZEL HEN), 2016. http://www.hlrs.de/de/systems/cray-xc40-hazel-hen.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[HSC+05] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24:547–555, 2005.

[HSK+16] Volker Helzle, Simon Spielmann, Thomas Knop, Eric Joris, Andreas Schuster, Marc Philipp Schmitz, Kai Gotz, and Farshad Einabadi. Final field trial and user evaluation report. Technical report, Dreamspace Project, 2016. http://www.dreamspaceproject.eu/Docs.

[HSW+05] K. Hunka, O. Suchowersky, S. Wood, L. Derwent, and Z. H. Kiss. Nursing time to program and assess deep brain stimulators in movement disorder patients. *Journal of Neuroscience Nursing*, 37(4):204–10, 2005. http://journals.lww.com/jnnonline/Abstract/2005/08000/Nursing_Time_to_Program_and_Assess_Deep_Brain.6.aspx.

[IBH11] Thiago Ize, Carson Brownlee, and Charles D. Hansen. Real-time ray tracer for visualizing massive models on a cluster. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pages 61–69, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[IL09] Marianthi Ierapetritou and Zukui Li. Modeling and managing uncertainty in process planning and scheduling. In Wanpracha Chaovalitwongse, Kevin C. Furman, and Panos M. Pardalos, editors, *Optimization and Logistics Challenges in the Enterprise*, volume 30 of *Springer Optimization and Its Applications*, pages 97–144. Springer US, 2009.

[Int98] Intel Corporation. Using the RDTSC instruction for performance monitoring. Technical report, 1998. https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[Int16a] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual - volume 2B. Technical report, September 2016. https:

//www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf.

[Int16b] Intel Corporation. OSPRay, 2016. http://www.ospray.org.

[JAW+12] G. P. Johnson, G. D. Abram, B. Westing, P. Navr'til, and K. Gaither. DisplayCluster: An interactive visualization environment for tiled displays. In *2012 IEEE International Conference on Cluster Computing*, pages 239–247, Sept 2012.

[Jeo09] Byungil Jeong. *Visualcasting - scalable real-time image distribution in ultra-high resolution display environments*. PhD thesis, University of Illinois, Chicago, 2009.

[JFE+09] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, and C. Bouras. Platform for distributed 3D gaming. *Int. J. Comput. Games Technol.*, 2009:1:1–1:15, January 2009.

[JJAM11] Julien Jomier, Sebastien Jourdain, Utkarsh Ayachit, and Charles Marion. Remote visualization of large datasets with MIDAS and ParaViewWeb. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pages 147–150, New York, NY, USA, 2011. ACM.

[JKKM+03] T. J. Jankun-Kelly, Oliver Kreylos, Kwan-Liu Ma, Bernd Hamann, Kenneth I. Joy, John Shalf, and E. Wes Bethel. Deploying web-based visual exploration tools on the grid. *IEEE Comput. Graph. Appl.*, 23(2):40–50, March 2003.

[JLF07] Stacy L. Janak, Xiaoxia Lin, and Christodoulos A. Floudas. A new robust optimization approach for scheduling under uncertainty: II. uncertainty with known probability distribution. *Computers and Chemical Engineering*, 31(3):171 – 195, 2007.

[JMWJ09] G.P. Johnson, S.A. Mock, B.M. Westing, and G.S. Johnson. EnVision: A web-based tool for scientific visualization. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 603–608, 2009.

[JRJ+06] Byungil Jeong, Luc Renambot, Ratko Jagodic, Rajvikram Singh, Julieta Aguilera, Andrew Johnson, and Jason Leigh. High-performance dynamic graphics streaming for Scalable Adaptive Graphics Environment. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[JRS+13] Jacek Jankowski, Sandy Ressler, Kristian Sons, Yvonne Jung, Johannes Behr, and Philipp Slusallek. Declarative integration of interactive 3D graphics into the world-wide web: Principles, current approaches, and research agenda. In *Proceedings of*

the 18th International Conference on 3D Web Technology, Web3D '13, pages 39–45, New York, NY, USA, 2013. ACM.

[KFBK+14] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie Hendren. Using JavaScript and WebCL for numerical computations: A comparative study of native and web technologies. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 91–102, New York, NY, USA, 2014. ACM.

[KFC+10] Jaroslav Křivánek, Marcos Fajardo, Per H. Christensen, Eric Tabellion, Michael Bunnell, David Larsson, and Anton Kaplanyan. Global illumination across industries. In *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10, New York, NY, USA, 2010. ACM.

[KGB+16] Andrey Khrekov, Jürgen Güninger, Kevin Baum, David McCann, and Jens Küger. MorphableUI: A hypergraph-based approach to distributed multimodal interaction for rapid prototyping and changing environments. In *Proceedings of the 24th Conference on Computer Graphics, Visualization and Computer Vision, WSCG 2016*, 2016.

[Khr17a] Khronos Group. Blender glTF 2.0 exporter, 2017. `https://github.com/KhronosGroup/glTF-Blender-Exporter`.

[Khr17b] Khronos Group. glTF - runtime 3D asset delivery, 2017. `https://github.com/KhronosGroup/glTF`.

[Khr17c] Khronos Group. WebGL overview, 2017. `https://www.khronos.org/webgl`.

[Kim06] Sung-Jin Kim. *The Diva Architecture and a Global Timestamp-based Approach for High-performance Visualization on Large Display Walls and Realization of High Quality-of-service Collaboration Environments*. PhD thesis, Long Beach, CA, USA, 2006. AAI3209299.

[KKW+13] Alexander Keller, Tero Karras, Ingo Wald, Timo Aila, Samuli Laine, Jacco Bikker, Christiaan Gribble, Won-Jong Lee, and James McCombe. Ray tracing is the future and ever will be... In *ACM SIGGRAPH 2013 Courses*, SIGGRAPH '13, pages 9:1–9:7, New York, NY, USA, 2013. ACM.

[KPS10] Mathias Kaspar, Nigel M. Parsad, and Jonathan C. Silverstein. CoWebViz: interactive collaborative sharing of 3D stereoscopic visualization among browsers with no added software. In *Proceedings of the 1st ACM International Health Informatics Symposium*, IHI '10, pages 809–816, New York, NY, USA, 2010. ACM.

[KRS+13] Felix Klein, Dmitri Rubinstein, Kristian Sons, Farshad Einabadi, Stephan Herhut, and Philipp Slusallek. Declarative AR and image processing on the web with Xflow. In *Proceedings of the 18th International Conference on Web 3D Technology*, 2013.

[Krü07] Jens Krüger. *GI-Edition Lecture Notes in Informatics (LNI)*, chapter A GPU Framework for Interactive Simulation and Rendering of Fluid Effects. GI, 2007.

[Krü10] Jens Krüger. A new sampling scheme for slice based volume rendering. In *Proceedings of the IEEE/EG International Symposium on Volume Graphics 2010*, 2010.

[KSSS14] Felix Klein, Torsten Spieldenner, Kristian Sons, and Philipp Slusallek. Configurable instances of 3D models for declarative 3D in the web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, Web3D '14, pages 71–79, New York, NY, USA, 2014. ACM.

[KVV+04] N. K. Krishnaprasad, V. Vishwanath, S. Venkataraman, A. G. Rao, L. Renambot, J. Leigh, A. E. Johnson, and B. Davis. JuxtaView - a tool for interactive visualization of large imagery on scalable tiled displays. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 411–420, Washington, DC, USA, 2004. IEEE Computer Society.

[LBH+15] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[Lev16] Levent-Levi, Tsahi. 4 reasons to choose H.264 for your WebRTC service (or why H.264 just won over VP8), 2016. `https://bloggeek.me/h264-webrtc`.

[LGCV05] Javier Lluch, Rafael Gaitán, Emilio Camahort, and Roberto Vivó. Interactive three-dimensional rendering on mobile computer devices. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 254–257, New York, NY, USA, 2005. ACM. `http://doi.acm.org/10.1145/1178477.1178520`.

[LGXT17] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized production path tracing. In *Proceedings of High Performance Graphics*, HPG '17, pages 10:1–10:11, New York, NY, USA, 2017. ACM.

[LH02] Eric J. Luke and Charles D. Hansen. Semotus Visum: a flexible remote visualization framework. In *Proceedings of the conference on Visualization '02*, VIS '02, pages 61–68, Washington, DC, USA, 2002. IEEE Computer Society.

[LI08] Zukui Li and Marianthi Ierapetritou. Process scheduling under uncertainty: Review and challenges. *Computers and Chemical Engineering*, 32(4 - 5):715 – 727, 2008. Festschrift devoted to Rex Reklaitis on his 65th Birthday.

[Lor11] Loreto, S. and Saint-Andre, P. and Salsano, S. and Wilkins, G. Known issues and best practices for the use of long polling and streaming in bidirectional HTTP, 2011. `https://tools.ietf.org/html/rfc6202`.

[LPHS12] Alexander Löffler, Luciano Pica, Hilko Hoffmann, and Philipp Slusallek. Networked displays for VR applications: Display as a service. In Ronan Boulic, Carolina Cruz-Neira, Kiyoshi Kiyokawa, and David Roberts, editors, *Virtual Environments 2012: Proceedings of Joint Virtual Reality Conference of ICAT. Joint Virtual Reality Conference (JVRC-2012), October 17-19, Madrid, Spain*. Eurographics Association, 2012.

[LR12] S. Loreto and Simon Pietro Romano. Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. *Internet Computing, IEEE*, 16(5):68–73, 2012.

[LS07] Fabrizio Lamberti and Andrea Sanna. A streaming-based solution for remote visualization of 3D graphics on mobile devices. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):247–260, 2007. `http://dx.doi.org/10.1109/TVCG.2007.29`.

[LSK11] Ming Li, Arne Schmitz, and Leif Kobbelt. Pseudo-immersive real-time display of 3D scenes on mobile devices. In *Proceedings of the 2011 International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission*, 3DIMPVT '11, pages 41–48, Washington, DC, USA, 2011. IEEE Computer Society.

[LSSS16] Stefan Lemme, Jan Sutter, Christian Schlinkmann, and Philipp Slusallek. The basic building blocks of declarative 3D on the web. In *Proceedings of the 21st International Conference on Web3D Technology*, Web3D '16, pages 17–25, New York, NY, USA, 2016. ACM.

[LZWH13] Yihua Lou, Haikuo Zhang, Wenjun Wu, and Zhenghui Hu. Magic View: An optimized ultra-large scientific image viewer for SAGE tiled-display environment. In *Proceedings of the 2013 IEEE 9th International Conference on e-Science*, ESCIENCE '13, pages 262–269, Washington, DC, USA, 2013. IEEE Computer Society.

[Ma09] Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Comput. Graph. Appl.*, 29(6):14–19, November 2009.

[Mah10] Mahy, R. and Matthews, P. and Rosenberg, J. Traversal Using Relays around NAT (TURN): Relay extensions to Session Traversal Utilities for NAT (STUN), 2010. `https://tools.ietf.org/rfc/rfc5766.txt`.

[MAN+14] T. Marrinan, J. Aurisano, A. Nishimoto, K. Bharadwaj, V. Mateevitsi, L. Renambot, L. Long, A. Johnson, and J. Leigh. SAGE2: A new approach for data intensive collaboration using scalable resolution shared displays. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on*, pages 177–186, Oct 2014.

[MBW+09] C. B. Maks, C. R. Butson, B. L. Walter, J. L. Vitek, and C. C. McIntyre. Deep brain stimulation activation volumes and their association with neurophysiological mapping and therapeutic outcomes. *J. Neurol Neurosurg Psychiatry*, (In Press), 2009. `http://www.ncbi.nlm.nih.gov/pubmed/18403440`.

[MCY+10] Jonathan C. McLane, W. Walter Czech, David A. Yuen, Mike R. Knox, Shuo Wang, Jim B. S. Greensky, and Erik O. D. Sevre. Ubiquitous interactive visualization of large-scale simulations in geosciences over a Java-based web-portal. *Concurr. Comput. : Pract. Exper.*, 22(12):1750–1773, August 2010.

[MDG+10] L. Marsalek, A. K. Dehof, I. Georgiev, H. P. Lenhof, P. Slusallek, and A. Hildebrandt. Real-time ray tracing of complex molecular scenes. In *2010 14th International Conference Information Visualisation*, pages 239–245, July 2010.

[MF12] M. M. Mobeen and Lin Feng. Ubiquitous medical volume rendering on mobile devices. In *Information Society (i-Society), 2012 International Conference on*, pages 93–98, June 2012.

[MHH+12] Stephen McAuley, Stephen Hill, Naty Hoffman, Yoshiharu Gotanda, Brian Smits, Brent Burley, and Adam Martinez. Practical physically-based shading in film and game production. In *ACM SIGGRAPH 2012 Courses*, SIGGRAPH '12, pages 10:1–10:7, New York, NY, USA, 2012. ACM.

[MHLK06] Andreas Moll, Andreas Hildebrandt, Hans-Peter Lenhof, and Oliver Kohlbacher. BALLView: a tool for research and education in molecular modeling. *Bioinformatics (Oxford, England)*, 22(3):365–6, feb 2006.

[Mic15] Microsoft Edge Team. A break from the past, part 2: Saying goodbye to ActiveX, VBScript, attachEvent..., 2015. https://blogs.windows.com/msedgedev/2015/05/06.

[Mic16] Microsoft. Microsoft Azure, 2016. https://azure.microsoft.com/en-us.

[MLM+12] J. Miroll, A. Löffler, J. Metzger, P. Slusallek, and T. Herfet. Reverse genlock for synchronous tiled display walls with smart internet displays. In *Consumer Electronics - Berlin (ICCE-Berlin), 2012 IEEE International Conference on*, pages 236–240, Sept 2012.

[MMN+07] S. Miocinovic, C. B. Maks, A. M. Noecker, C. R. Butson, and C. C. McIntyre. Cicerone: Deep brain stimulation neurosurgical navigation software system. *Acta Neurochir Suppl (Wien)*, 97:561–567, 2007.

[MMS+04] C. C. McIntyre, S. Mori, D. L. Sherman, N. V. Thakor, and J. L. Vitek. Electric field and stimulating influence generated by deep brain stimulation of the subthalamic nucleus. *Clin Neurophysiol*, 115(3):589–95, 2004.

[Moz16] Mozilla. Live streaming web audio and video, 2016. https://developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery/Live_streaming_web_audio_and_video.

[MPJ+13] Charles Marion, Joachim Pouderoux, Julien Jomier, Sebastien Jourdain, Marcus Hanwell, and Utkarsh Ayachit. A hybrid visualization system for molecular models. In *Proceedings of the 18th International Conference on 3D Web Technology*, Web3D '13, pages 117–120, New York, NY, USA, 2013. ACM.

[MR09] Arie Meir and Boris Rubinsky. Distributed network, wireless and cloud computing enabled 3-D ultrasound; a new medical technology paradigm. *PLoS ONE*, 4(11):e7974, 11 2009. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2775631.

[MT03] Kenneth Moreland and David Thompson. From cluster to wall with VTK. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 5–, Washington, DC, USA, 2003. IEEE Computer Society.

[MW08] M. Moser and D. Weiskopf. Interactive volume rendering on mobile devices. In *Workshop on Vision, Modelling, and Visualization VMV '08*, pages 217–226, 2008. http://www.vis.uni-stuttgart.de/~weiskopf/publications/vmv08.pdf.

[MWMS07] Brendan Moloney, Daniel Weiskopf, Torsten Möller, and Magnus Strengert. Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '07, pages 45–52, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[MWP01] Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, PVG '01, pages 85–92, Piscataway, NJ, USA, 2001. IEEE Press.

[NDB+14] D. Nachbaur, R. Dumusc, A. Bilgili, J. Hernando, and S. Eilemann. Remote parallel rendering for high-resolution tiled display walls. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, pages 117–118, Nov 2014.

[NDV+10] Sungwon Nam, Sachin Deshpande, Venkatram Vishwanath, Byungil Jeong, Luc Renambot, and Jason Leigh. Multi-application inter-tile synchronization on ultra-high-resolution display walls. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 145–156, New York, NY, USA, 2010. ACM.

[Nel17] Nelson, Brad. Goodbye PNaCl, hello WebAssembly!, 2017. `https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html`.

[NFLC12] Paul A. Navrátil, Donald S. Fussell, Calvin Lin, and Hank Childs. Dynamic scheduling for large-scale distributed-memory ray tracing. In Hank Childs, Torsten Kuhlen, and Fabio Marton, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012.

[NHM11] B. Neal, P. Hunkin, and A. McGregor. Distributed OpenGL rendering in network bandwidth constrained environments. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pages 21–29, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[NHN07] Nirnimesh, P. Harish, and P. J. Narayanan. Garuda: A scalable tiled display wall using commodity PCs. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):864–877, September 2007.

[NJR+09] Sungwon Nam, Byungil Jeong, Luc Renambot, Andrew Johnson, Kelly Gaither, and Jason Leigh. Remote visualization of large scale data for ultra-high resolution display environments. In *Proceedings of the 2009 Workshop on Ultrascale Visualization*, UltraVis '09, pages 42–44, New York, NY, USA, 2009. ACM.

[NSOJA11] José M. Noguera, Rafael J. Segura, Carlos J. Ogáyar, and Robert Joan-Arinyo. Navigating large terrains using commodity mobile devices. *Computers & Geosciences*, 37(9):1218 – 1233, 2011.

[NSOJA13] José M. Noguera, Rafael J. Segura, Carlos J. Ogáyar, and Robert Joan-Arinyo. A scalable architecture for 3D map navigation on mobile devices. *Personal Ubiquitous Comput.*, 17(7):1487–1502, October 2013.

[NVI16a] NVIDIA Corporation. NVIDIA GRID, 2016. `http://www.nvidia.com/object/nvidia-grid.html`.

[NVI16b] NVIDIA Corporation. NVIDIA IRAY, 2016. `http://www.nvidia-arc.com/iray.html`.

[NVI16c] NVIDIA Corporation. NVIDIA MENTAL RAY, 2016. `http://www.nvidia-arc.com/products/nvidia-mental-ray.html`.

[NVI16d] NVIDIA Corporation. NVIDIA QUADRO VCA, 2016. `http://www.nvidia.com/object/visual-computing-appliance.html`.

[OFW+09] M. S. Okun, H. H. Fernandez, S. S. Wu, Kirsch-Darrow, Bowers L., Bova D., F., M. Suelter, Charles E Jacobson, Xinping Wang, Clifford W Gordon, Pam Zeilman, Janet Romrell, Pam Martin, Herbert Ward, Ramon L Rodriguez, and Kelly D Foote. Cognition and mood in Parkinson's disease in subthalamic nucleus versus globus pallidus interna deep brain stimulation: the COMPARE trial. *Annals of Neurology*, 65:586–95, 2009.

[OnL16] OnLive. OnLive, 2016. `http://www.onlive.com`.

[OOI11] Yasuhide Okamoto, Takeshi Oishi, and Katsushi Ikeuchi. Image-based network rendering of large meshes for cloud computing. *Int. J. Comput. Vision*, 94(1):12–22, August 2011.

[Ope17] OpenFabrics Alliance. libibverbs - a library for direct userspace access to InfiniBand hardware, 2017. `https://git.kernel.org/pub/scm/libs/infiniband/libibverbs.git`.

[Oto13] Otoy. ORBX 2 technical introduction, October 2013. `https://aws.otoy.com/docs/ORBX2_Whitepaper.pdf`.

[Oto15a] Otoy. ORBX, 2015. `https://home.otoy.com/stream/orbx`.

[OTO15b] OTOY Inc. OctaneRender, 2015. `https://home.otoy.com/render/octane-render`.

[PB13] Bernhard Preim and Charl P. Botha. *Visual Computing for Medicine: Theory, Algorithms, and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[PBD+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA, 2010. ACM.

[PDK10] Kevin Ponto, Kai Doerr, and Falko Kuester. Giga-stack: A method for visualizing giga-pixel layered imagery on massively tiled displays. *Future Gener. Comput. Syst.*, 26(5):693–700, May 2010.

[PGM16] Arsène Pérard-Gayot and Richard Membarth. Implementation of selected modules for real-time ray tracing and advanced lightning simulation. Technical report, Saarland University, 2016. `http://www.dreamspaceproject.eu/Docs`.

[PGTM16] Arsène Pérard-Gayot, Georg Tamm, and Richard Membarth. Real-time rendering and lighting demonstrator. Technical report, Saarland University, 2016. `http://www.dreamspaceproject.eu/Docs`.

[PHKH04] Steffen Prohaska, Andrei Hutanu, Ralf Kahler, and Hans-Christian Hege. Interactive exploration of large remote micro-CT scans. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 345–352, Washington, DC, USA, 2004. IEEE Computer Society.

[PKI08] Sanghun Park, Wontae Kim, and Insung Ihm. Mobile collaborative medical display system. *Computer Methods and Programs in Biomedicine*, 89(3):248 – 260, 2008. `http://dx.doi.org/10.1016/j.cmpb.2007.11.012`.

[Pla02] Tomas Plachetka. Perfect load balancing for demand-driven parallel ray tracing. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 410–419, London, UK, UK, 2002. Springer-Verlag.

[PMS+99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 119–126, New York, NY, USA, 1999. ACM.

[PRR03] Volker Paelke, Christian Reimann, and Waldemar Rosenbach. A visualization design repository for mobile devices. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and*

interaction in Africa, pages 57–62, New York, NY, USA, 2003. ACM. http://doi.acm.org/10.1145/602330.602341.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[PV06] Jorge-Leon Posada-Velasquez. *A Methodology for the Semantic Visualization of Industrial Plant CAD Models for Virtual Reality Walkthroughs*. PhD thesis, Technische Universität, Darmstadt, April 2006.

[PZB16] D. Pohl, X. Zhang, and A. Bulling. Combining eye tracking with optimizations for lens astigmatism in modern wide-angle HMDs. In *2016 IEEE Virtual Reality (VR)*, pages 269–270, March 2016.

[QLB+16] Peter Quax, Jori Liesenborgs, Arno Barzan, Martijn Croonen, Wim Lamotte, Bert Vankeirsbilck, Bart Dhoedt, Tom Kimpe, Kurt Pattyn, and Matthew Mclin. Remote rendering solutions using web technologies. *Multimedia Tools Appl.*, 75(8):4383–4410, April 2016.

[RA12] Marcos Balsa Rodríguez and Pere Pau Vázquez Alcocer. *Practical Volume Rendering in Mobile Devices*, pages 708–718. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Ray13] Raymond, Robin. SDP the WebRTC boat anchor, 2013. https://webrtc.is/2013/03/06/sdp-the-webrtc-boat-anchor.

[RCSW14] Florian Reichl, Matthäus G. Chajdas, Jens Schneider, and Rüdiger Westermann. Interactive rendering of giga-particle fluid simulations. *Proceedings of High Performance Graphics 2014*, 2014.

[Ren16] Render Legion s.r.o. Corona Renderer, 2016. https://corona-renderer.com.

[RJJ+06] Luc Renambot, Byungil Jeong, Ratko Jagodic, Andrew Johnson, and Jason Leigh. Collaborative visualization using high-resolution tiled displays. In *In ACM CHI Workshop on Information Visualization Interaction Techniques for Collaboration Across Multiple Displays*, 2006.

[RLRS09] Michael Repplinger, Alexander Löffler, Dmitri Rubinstein, and Philipp Slusallek. DRONE: A flexible framework for distributed rendering and display. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I*, ISVC '09, pages 975–986, Berlin, Heidelberg, 2009. Springer-Verlag.

[RMA+16] Luc Renambot, Thomas Marrinan, Jillian Aurisano, Arthur Nishimoto, Victor Mateevitsi, Krishna Bharadwaj, Lance Long, Andy Johnson, Maxine Brown, and Jason Leigh. SAGE2. *Future Gener. Comput. Syst.*, 54(C):296–305, January 2016.

[RND13] Mario Rincón-Nigro and Zhigang Deng. Cost-based workload balancing for ray tracing on multi-GPU systems. In *ACM SIGGRAPH 2013 Posters*, SIGGRAPH '13, pages 41:1–41:1, New York, NY, USA, 2013. ACM.

[Ros08] Rosenberg, J. and Mahy, R. and Matthews, P. and Wing, D. Session Traversal Utilities for NAT (STUN), 2008. `https://tools.ietf.org/html/rfc5389`.

[Ros10] Rosenberg, J. Interactive Connectivity Establishment (ICE): A protocol for Network Address Translator (NAT) traversal for offer/answer protocols, 2010. `https://tools.ietf.org/html/rfc5245`.

[RSFWH98] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, Jan 1998.

[SAK15] Alexander Schiewe, Mario Anstoots, and Jens Krüger. State of the art in mobile volume rendering on iOS devices. In E. Bertini, J. Kennedy, and E. Puppo, editors, *Eurographics Conference on Visualization (EuroVis) - Short Papers*. The Eurographics Association, 2015.

[Sar14] Sarkar, Samit. PlayStation Now game-streaming service coming summer 2014 (update), 2014. `https://www.polygon.com/2014/1/7/5284504/sony-playstation-now-gaikai-based-streaming-service-ps-ps2-ps3`.

[SBEF14] Christoph Schinko, René Berndt, Eva Eggeling, and Dieter Fellner. A scalable rendering framework for generative 3D content. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, Web3D '14, pages 81–87, New York, NY, USA, 2014. ACM.

[Sch13] Schuh, Justin. Saying goodbye to our old friend NPAPI, 2013. `http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html`.

[SCM+16] Jonathan Starck, Adam Cherbetji, Brice Michoud, Oliver Grau, Farshad Einabadi, Volker Helzle, Philipp Slusallek, Richard Membarth, Sammy Rogmans, and Vincent Jacobs. Final prototype Dreamspace tool sets. Technical report, Dreamspace Project, 2016. `http://www.dreamspaceproject.eu/Docs`.

[SDHS13] Kristian Sons, Georg Demme, Wolfgang Herget, and Philipp Slusallek. Fortress city saarlouis: Development of an interactive 3D city model using web technologies.

In A. Traviglia, editor, *Proceedings of the CAA 2013 Conference Across Space and Time. Annual International Conference on Computer Applications and Quantitative Methods in Archaeology (CAA-2013), March 25-28, Perth,, WA, Australia*. TBA, 2013.

[SE01] Ove Sommer and Thomas Ertl. Comparative visualization of instabilities in crashworthiness simulations. In *Proceedings of the 3rd Joint Eurographics - IEEE TCVG Conference on Visualization*, EGVISSYM'01, pages 319–328, Aire-la-Ville, Switzerland, Switzerland, 2001. Eurographics Association.

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974.

[SH08] J. M. Schwalb and C. Hamani. The history and future of deep brain stimulation. *Neurotherapeutics*, 5(1):3–13, 2008.

[SHNC11] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 103–112, New York, NY, USA, 2011. ACM.

[SJR+04] R. Singh, Byungil Jeong, L. Renambot, A. Johnson, and J. Leigh. TeraVision: A distributed, scalable, high resolution graphics streaming system. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 391–400, Washington, DC, USA, 2004. IEEE Computer Society.

[SKR+10] Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. XML3D: interactive 3D graphics for the web. In *Web3D '10: Proceedings of the 15th International Conference on Web 3D Technology*, pages 175–184, New York, NY, USA, 2010. ACM.

[SKSS14] Kristian Sons, Felix Klein, Jan Sutter, and Philipp Slusallek. shade.js: Adaptive material descriptions. *Computer Graphics Forum*, 33(7):51–60, 2014.

[SLTB15] Christian Stein, Max Limper, Maik Thöner, and Johannes Behr. hare3d - rendering large models in the browser. *WebGL Insights*, pages 317–332, 2015.

[Sme15] Smedberg, Benjamin. NPAPI plugins in Firefox, 2015. `https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox`.

[SNC12] Shu Shi, Klara Nahrstedt, and Roy Campbell. A real-time remote rendering system for interactive mobile graphics. *ACM Trans. Multimedia Comput. Commun. Appl.*, 8(3s):46:1–46:20, October 2012.

[Sod11] I. Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *MultiMedia, IEEE*, 18(4):62–67, 2011.

[Sof15] Software-Cluster Coordination Office. SINNODIUM - software innovations for the digital enterprise, 2015. `http://software-cluster.com/projects/sinnodium`.

[SRCW13] Bernhard Sadransky, Hrvoje Ribicic, Robert Carnecky, and Jürgen Waser. Visdom Mobile: Decision support on-site using visual simulation control. In *Proceedings of the 29th Spring Conference on Computer Graphics*, SCCG '13, pages 099:99–099:106, New York, NY, USA, 2013. ACM.

[SSB09] R. Suselbeck, G. Schiele, and C. Becker. Peer-to-peer support for low-latency massively multiplayer online games in the cloud. In *Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop on*, pages 1–2, 2009.

[SSS14] Jan Sutter, Kristian Sons, and Philipp Slusallek. Blast: A binary large structured transmission format for the web. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies*, Web3D '14, pages 45–52, 2014.

[Sta11] Stanford Computer Graphics Laboratory. The Stanford 3D scanning repository, September 2011. `http://graphics.stanford.edu/data/3Dscanrep`.

[Str08] Strom, J. GL_OES_compressed_ETC1_RGB8_texture, 2008. `https://www.khronos.org/registry/OpenGL/extensions/OES/OES_compressed_ETC1_RGB8_texture.txt`.

[SVK+12] Luiz Scheidegger, Huy T. Vo, Jens Krüger, Cludio T. Silva, and Joo L. D. Comba. Parallel large data visualization with display walls. volume 8294, pages 82940C–82940C–11, 2012.

[SVR+04] Nicholas Schwarz, Shalini Venkataraman, Luc Renambot, Naveen Krishnaprasad, Venkatram Vishwanath, Jason Leigh, Andrew Johnson, Graham Kent, and Atul Nayak. Vol-a-Tile - a tool for interactive exploration of large volumetric data on scalable tiled displays. In *Proceedings of the Conference on Visualization '04*, VIS '04, pages 598.19–, Washington, DC, USA, 2004. IEEE Computer Society.

[SY17] Myungbae Son and Sung-Eui Yoon. Timeline scheduling for out-of-core ray batching. In *Proceedings of High Performance Graphics*, HPG '17, pages 11:1–11:10, New York, NY, USA, 2017. ACM.

[TFF05] J.D. Teresco, J. Fair, and J.E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science Engineering*, 7(2):40–50, 2005.

[TFK12] Georg Tamm, Thomas Fogal, and Jens Krüger. Hybrid distributed rendering. In *IEEE LDAV Symposium 2012 Posters*, 10 2012.

[TGS+15] Jonas Trottnow, Kai Götz, Stefan Seibert, Simon Spielmann, Volker Helzle, Farshad Einabadi, Clemens K. H. Sielaff, and Oliver Grau. Intuitive virtual production tools for set and light editing. In *Proceedings of the 12th European Conference on Visual Media Production*, CVMP '15, pages 6:1–6:8, New York, NY, USA, 2015. ACM.

[thr17] three.js authors. three.js, 2017. `https://threejs.org`.

[TK14] Georg Tamm and Jens Krüger. Hybrid rendering with scheduling under uncertainty. *IEEE Transactions on Visualization and Computer Graphics*, 20:767–780, 2014.

[TL04] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 469–476, New York, NY, USA, 2004. ACM.

[TMP11] N. Tizon, C. Moreno, and M. Preda. ROI based video streaming for 3D remote rendering. In *2011 IEEE 13th International Workshop on Multimedia Signal Processing*, pages 1–6, Oct 2011.

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[TS15] Georg Tamm and Philipp Slusallek. Plugin free remote visualization in the browser. In *Proceedings of SPIE 9397. SPIE Conference on Visualization and Data Analysis, February 8-12, San Francisco,, CA, USA*. o. A., 2015.

[TS16] Georg Tamm and Philipp Slusallek. Web-enabled server-based and distributed real-time ray-tracing. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV-16), June 6-7, Groningen, Netherlands*. o. A., 6 2016.

[TSK11] Georg Tamm, Alexander Schiewe, and Jens Krüger. ZAPP - a management framework for distributed visualization systems. In *Proceedings of the IADIS Computer Graphics, Visualization, Computer Vision and Image Processing 2011. IADIS International Conference Computer Graphics, Visualization, Computer Vision and Image Processing (CGVCVIP-2011), July 24-26, Rome, Italy*. o.A., 2011.

[TV10] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, November 2010.

[Unr16] Unreal Streaming Technologies. Unreal Media Server, 2016. `http://umediaserver.net/index.html`.

[U.S12] U.S. National Library of Medicine. The Visible Human Project, April 2012. `http://www.nlm.nih.gov/research/visible`.

[Vid17] VideoLAN Organization. x264, 2017. `http://www.videolan.org/developers/x264.html`.

[VP01] C.G. Vassiliadis and E.N. Pistikopoulos. Maintenance scheduling and process optimization under uncertainty. *Computers and Chemical Engineering*, 25(2 - 3):217 – 236, 2001.

[VVR07] Sivakumar Viswanathan, Bharadwaj Veeravalli, and Thomas G. Robertazzi. Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1450–1461, October 2007.

[W3C16a] W3C. Media Source Extensions, 2016. `https://w3c.github.io/media-source`.

[W3C16b] W3C ORTC Community Group. ORTC - object API for RTC mobile, server, web, 2016. `https://ortc.org`.

[W3C16c] W3C WebRTC Working Group. WebRTC 1.0: Real-time communication between browsers, 2016. `http://www.w3.org/TR/webrtc`.

[WAB+05] Grant Wallace, Otto J. Anshus, Peng Bi, Han Chen, Yuqun Chen, Douglas Clark, Perry Cook, Adam Finkelstein, Thomas Funkhouser, Anoop Gupta, Matthew Hibbs, Kai Li, Zhiyan Liu, Rudrajit Samanta, Rahul Sukthankar, and Olga Troyanskaya. Tools and applications for large-scale display walls. *IEEE Comput. Graph. Appl.*, 25(4):24–33, July 2005.

[WBDS03] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive ray tracing on commodity PC clusters. In Harald Kosch, Lszl Bszrmnyi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 499–508. Springer Berlin Heidelberg, 2003.

[Web17] WebRTC project authors. WebRTC adapter, 2017. `https://github.com/webrtc/adapter`.

[WFS+09] F. M. Weaver, K. Follett, M. Stern, K. Hur, C. Harris, Jr. Marks, W. J., J. Rothlind, O. Sagher, D. Reda, C. S. Moy, R. Pahwa, K. Burchiel, P. Hogarth, E. C. Lai, J. E. Duda, K. Holloway, A. Samii, S. Horn, J. Bronstein, G. Stoner, J. Heemskerk, and G. D. Huang. Bilateral deep brain stimulation vs best medical therapy for patients with advanced Parkinson disease: a randomized controlled trial. *JAMA*, 301(1):63–73, 2009. `http://www.ncbi.nlm.nih.gov/pubmed/19126811`.

[WPJR11] Andrew Wessels, Mike Purvis, Jahrain Jackson, and Syed (Shawon) Rahman. Remote data visualization through WebSockets. In *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations*, ITNG '11, pages 1050–1051, Washington, DC, USA, 2011. IEEE Computer Society.

[WPSB03] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, and Carsten Benthin. Realtime ray tracing and its use for interactive global illumination. In *In Eurographics State of the Art Reports*, 2003.

[WS01] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. In *Eurographics*, pages 21–42, 2001.

[WWB+14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.*, 33(4):143:1–143:8, July 2014.

[Yan06] Wei Yan. Integrating web 2D and 3D technologies for architectural visualization: Applications of SVG and X3D/VRML in environmental behavior simulation. In *Proceedings of the Eleventh International Conference on 3D Web Technology*, Web3D '06, pages 37–45, New York, NY, USA, 2006. ACM.

[YN00] I. Yoon and U. Neumann. Web-based remote rendering with IBRAC (Image-Based Rendering Acceleration and Compression). *Computer Graphics Forum*, 2000.

[YSD+09] B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93, 2009.

[YWG+10] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Comput. Graph. Appl.*, 30(3):45–57, May 2010.

[ZMM+12] M. Zorrilla, Á. Martín, F. Mogollón, J. García, and I. G. Olaizola. End to end solution for interactive on demand 3D media on home network devices. In *IEEE international Symposium on Broadband Multimedia Systems and Broadcasting*, pages 1–6, June 2012.

[ZMS+14] Mikel Zorrilla, Angel Martin, Jairo R. Sanchez, Iñigo Tamayo, and Igor G. Olaizola. HTML5-based system for interoperable 3D digital home applications. *Multimedia Tools Appl.*, 71(2):533–553, July 2014.