Interactive Web-based Visualization

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Finian Mwalongo

aus Njombe, Tansania

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. sc. Filip Sadlo
Tag der mündlichen Prüfung:	11. Oktober 2018

Visualisierungsinstitut
der Universität Stuttgart

2018

Contents

Acknowledgments **xi**

Abstract **xiii**

German Abstract — Zusammenfassung **xv**

1 Introduction **1**
- 1.1 Thesis Contributions . . . 3
- 1.2 Structure of the Thesis . . . 4

2 Fundamentals **7**
- 2.1 Introduction to Data Visualization . . . 7
  - 2.1.1 Molecular Data Visualization . . . 9
  - 2.1.2 GPU-based Volume Visualization . . . 11
- 2.2 OpenGL and WebGL Rendering . . . 12
  - 2.2.1 OpenGL Rendering Pipeline . . . 13
  - 2.2.2 WebGL Rendering Pipeline . . . 14
- 2.3 Modern Browsers and HTML5 Technologies . . . 16
- 2.4 Approaches for 3D Graphics in the Browser . . . 19

3 Remote Visualization **23**
- 3.1 Motivation . . . 23
- 3.2 Remote Rendering Approaches . . . 25
- 3.3 Visualization as a Web Service . . . 26
- 3.4 Scaling Server-side Infrastructure . . . 29
  - 3.4.1 Grid-based Visualization . . . 29
  - 3.4.2 Cloud-based Visualization . . . 33
  - 3.4.3 Design Considerations for Cloud-based Visualization Service . . . 36
  - 3.4.4 Resource and Data Management in the Cloud . . . 38
- 3.5 Interactive Web-based Visualization . . . 39
  - 3.5.1 Data Encoding and Transfer Techniques . . . 41
  - 3.5.2 Local Rendering in the Browser . . . 44
- 3.6 Web-based Visualization Applications . . . 46
- 3.7 Conclusion and Challenges . . . 54
  - 3.7.1 Conclusion . . . 54
  - 3.7.2 Challenges . . . 56

4 GPU-based Molecular Data Visualization in the Browser **59**

4.1 GPU-based Ray Casting 60
4.2 Acceleration Data Structures 62
4.3 Implementation 63
4.4 Results and Discussion 66

5 GPU-based Remote Visualization of Dynamic Molecular Data on the Web 73
5.1 Overview 75
5.2 Implementation 75
5.2.1 Data Encoding and Quantization 77
5.2.2 WebGL GPU-based Ray Casting 78
5.2.3 Triangle Rendering 79
5.2.4 Deferred Shading 81
5.3 Results and Discussion 83
5.4 Conclusion 87

6 Web-based Visualization of Bricked Volumetric Data with Levels of Detail 89
6.1 Algorithmic Pipeline 92
6.2 Implementation 93
6.2.1 Server-side Brick Generation 93
6.2.2 Data Encoding and Transfer 94
6.2.3 Client-side Data Processing 94
6.2.4 Prioritization of Volume Bricks 95
6.2.5 Client-side Rendering using WebGL 97
6.3 Results and Discussion 97
6.4 Summary and Conclusion 102

7 Conclusion and Outlook 105
7.1 Conclusion 105
7.2 Outlook 108
7.2.1 Visualization as a Cloud Service 108
7.2.2 Collaborative Web-based Visualization 109

Bibliography 113

List of Figures

Chapter 2

2.1 Visualization pipeline 8
2.2 Example visualizations produced by some web-based molecular visualization tools 9
2.3 The volume ray marching rendering technique 12
2.4 Relationship between OpenGL, OpenGL ES and WebGL 13
2.5 A Simplified WebGL 2.0 rendering pipeline 14
2.6 Simplified OpenGL rendering pipeline 15
2.7 Client-side optimization technologies in the browser 17
2.8 Architecture of XML3D and X3DOM 20
2.9 Example of rendering simple 3D shapes using X3DOM library . . 21
2.10 Example visualizations of different datasets using X3DOM 22

Chapter 3

3.1 Grid-based visualization pipeline based on web services 30
3.2 GPU support in the cloud through virtualization software and GPU passthrough Approaches 35
3.3 Interactive visualization of the chaperonin complex 47
3.4 Visualization of molecular structures at interactive frame rates using WebGL-based ray casting techniques 48
3.5 Example of medical volume rendering in WebGL using volume ray marching 49
3.6 Example of a web-based medical volume rendering 50
3.7 Examples of web-based information visualization using FluidDiagrams 50
3.8 Examples of web-based geospatial visualization applications 52
3.9 Examples of web-based 3D model visualizations 53

Chapter 4

4.1 Visualization of spacefilling model for proteins using our WebGL glyph ray casting method 61
4.2 Data structures for the grid that is used for ray casting the spheres 62
4.3 Server-side preprocessing 63
4.4 Client-side preprocessing 64

4.5 Fragment shader code for accessing voxel data from the grid data structure . . . 65
4.6 Example visualizations generated by our GPU-based ray casting technique . . . 67
4.7 Visualization of an insulin protein with a close-up view (PDB ID: 1RWE, 823 atoms) . . . 67
4.8 Volume rendering of an insulin protein (PDB ID: 1RWE) combined with a triangulated Cartoon representation . . . 70

Chapter 5

5.1 Our application architecture . . . 76
5.2 Memory layout for the data buffer sent by the server . . . 77
5.3 The visualization of the capsid of a papillomavirus (PDB ID: 3IYJ) consisting of 1.3 million atoms . . . 80
5.4 GPU-based sphere ray casting combined with triangle rendering . 81
5.5 Different postprocessing effects via deferred shading . . . 82

Chapter 6

6.1 Algorithmic pipeline of our client - server architecture for the bricked volume rendering . . . 91
6.2 Memory layout for brick data serialization between server and client 93
6.3 Effects of importance-based prioritization of bricks . . . 96
6.4 Different snapshots of the bricked volume rendering method showing the aneurism data set at different levels of detail . . . 98
6.5 Visualization of an hazelnuts volume data set using maximum intensity projection . . . 99
6.6 Transfer and decompression times for the Hazelnut data set . . . 100
6.7 Performance of the WebGL volume renderer for different data sets and rendering technique . . . 101
6.8 Visualization of the engine volume data set using a transfer function 102
6.9 Visualization of the flower data set with 1024^3 voxels using a transfer function . . . 103

List of Tables

Chapter 4

4.1 Performance measurements for the client-side preprocessing and the server-side preprocessing 66

Chapter 5

5.1 The systems used for performance measurement including their specifications 83
5.2 Performance results showing *Transfer times* for the *Laptop* client machine for various molecules in both LAN and WiFi network environments 84
5.3 Rendering performance of the GPU ray casting in frames per second for various molecules using the *PC* and *Laptop* client machines 84
5.4 Rendering performance of the GPU ray casting in frames per second of various molecules using the *PC* and *Laptop* client machines with the optimized data layout 86

List of Abbreviations and Acronyms

CPU Central Processing Unit
GPGPU General-Purpose Computation on Graphics Processing Units
GPU Graphics Processing Unit
API Application Programming Interface
WebGL Web Graphics Library
VTK Visualization Toolkit
GVK Grid Visualization Kernel
vGPU Virtual Graphics Processing Unit
LOD Level of Detail
OpenGL Open Graphics Library
FPS Frames Per Second
PCIe PCI Express—Peripheral Component Interconnect Express

Acknowledgments

First and foremost, I thank God for the gift of life and good health, without which nothing could have been done. I am extremely grateful to my supervisor, Thomas Ertl for accepting to supervise my thesis and for his constant and dedicated support through out my PhD work. It is this quality supervision and support that has made it possible to bring this work to a successful end. I feel indebted and will always remain grateful. I also thank Filip Sadlo for accepting to serve as an external examiner for my thesis.

I thank Guido Reina for the support and collaboration that we had. Many thanks to Michael Krone for being a great officemate and collaborator. Through these collaborations, I have learnt a lot. I also thank Grzegorz Karch and Michael Becher who collaborated with me on some projects.

I thank all my officemates and colleagues at VISUS and VIS for being very kind and supportive. In a special way, I should mention Gustavo Machado for his great friendship and Christoph Müller for his help with printing my dissertation.

I thank the DAAD for granting me a scholarship to pursue my studies in Germany and Dar es Salaam Institute of Technology for granting me a study leave and for financial support.

It would have been difficult to reach this far without the support of my best friend and wife, Angelina. Thank you for your understanding, prayers, and the sacrifices that you had to make just to ensure that I have full peace of mind to concentrate on my research. May God bless you abundantly.

I also acknowledge all the support that I received from my family and friends. Every one in his or her own way helped to propel me forward.

Abstract

The visualization of large amounts of data, which cannot be easily copied for processing on a user's local machine, is not yet a fully solved problem. Remote visualization represents one possible solution approach to the problem, and has long been an important research topic. Depending on the device used, modern hardware, such as high-performance GPUs, is sometimes not available. This is another reason for the use of remote visualization. Additionally, due to the growing global networking and collaboration among research groups, collaborative remote visualization solutions are becoming more important. The additional use of collaborative visualization solutions is eventually due to the growing global networking and collaboration among research groups.

The attractiveness of web-based remote visualization is greatly increased by the wide availability of web browsers on almost all devices; these are available today on all systems—from desktop computers to smartphones. In order to ensure interactivity, network bandwidth and latency are the biggest challenges that web-based visualization algorithms have to solve. Despite the steady improvements in available bandwidth, these improvements are still significantly slower than, for example, processor performance, resulting in increasing the impact of this bottleneck. For example, visualization of large dynamic data in low-bandwidth environments can be challenging because it requires continuous data transfer. However, bandwidth improvement alone cannot improve the latency because it is also affected by factors such as the distance between server and client and network utilization.

To overcome these challenges, a combination of techniques is needed to customize the individual processing steps of the visualization pipeline, from efficient data representation to hardware-accelerated rendering on the client side. This thesis first deals with related work in the field of remote visualization with a particular focus on interactive web-based visualization and then presents techniques for interactive visualization in the browser using modern web standards such as WebGL and HTML5. These techniques enable the visualization of dynamic molecular data sets with more than one million atoms at interactive frame rates using GPU-based ray casting. Due to the limitations which exist in a browser-based environment, the concrete implementation of the GPU-based ray casting had to be customized. Evaluation of the resulting performance shows that GPU-based techniques enable the interactive rendering of large data sets and achieve higher image quality compared to polygon-based techniques.

In order to reduce data transfer times and network latency, and improve rendering speed, efficient approaches for data representation and transmission are

used. Furthermore, this thesis introduces a GPU-based volume-ray marching technique based on WebGL 2.0, which uses progressive brick-wise data transfer, as well as multiple levels of detail in order to achieve interactive volume rendering of datasets stored on a server.

The concepts and results presented in this thesis contribute to the further spread of interactive web-based visualization. The algorithmic and technological advances that have been achieved form a basis for further development of interactive browser-based visualization applications. At the same time, this approach has the potential for enabling future collaborative visualization in the cloud.

German Abstract
—Zusammenfassung—

Die Visualisierung großer Datenmengen, welche nicht ohne Weiteres zur Verarbeitung auf den lokalen Rechner des Anwenders kopiert werden können, ist ein bisher nicht zufriedenstellend gelöstes Problem. Remote-Visualisierung stellt einen möglichen Lösungsansatz dar und ist deshalb seit langem ein relevantes Forschungsthema. Abhängig vom verwendeten Endgerät ist moderne Hardware, wie etwa performante GPUs, teilweise nicht verfügbar. Dies ist ein weiterer Grund für den Einsatz von Remote-Visualisierung. Durch die zunehmende globale Vernetzung und Kollaboration von Forschungsgruppen gewinnt kollaborative Remote-Visualisierung zusätzlich an Bedeutung.

Die Attraktivität web-basierter Remote-Visualisierung wird durch die weitreichende Verfügbarkeit von Web-Browsern auf nahezu allen Endgeräten enorm gesteigert; diese sind heutzutage auf allen Systemen – vom Desktop-Computer bis zum Smartphone – vorhanden. Bei der Gewährleistung der Interaktivität sind Bandbreite und Latenz der Netzwerkverbindung die größten Herausforderungen, welche von web-basierten Visualisierungs-Algorithmen gelöst werden müssen. Trotz der stetigen Verbesserungen hinsichtlich der verfügbaren Bandbreite steigt diese signifikant langsamer als beispielsweise die Prozessorleistung, wodurch sich die Auswirkung dieses Flaschenhalses immer weiter verstärkt. So kann beispielsweise die Visualisierung großer dynamischer Daten in Umgebungen mit geringer Bandbreite eine Herausforderung darstellen, da kontinuierlicher Datentransfer benötigt wird. Dennoch kann die alleinige Verbesserung der Bandbreite keine entsprechende Verbesserung der Latenz bewirken, da diese zudem von Faktoren wie der Distanz zwischen Server und Client sowie der Netzwerkauslastung beeinflusst wird.

Um diese Herausforderungen zu bewältigen, wird eine Kombination verschiedener Techniken für die Anpassung der einzelnen Verarbeitungsschritte der Visualisierungspipeline benötigt, angefangen bei effizienter Datenrepräsentation bis hin zu hardware-beschleunigtem Rendering auf der Client-Seite. Diese Doktorarbeit befasst sich zunächst mit verwandten Arbeiten auf dem Gebiet der Remote-Visualisierung mit besonderem Fokus auf interaktiver web-basierter Visualisierung und präsentiert danach Techniken für die interaktive Visualisierung im Browser mit Hilfe moderner Web-Standards wie WebGL und HTML5. Diese Techniken ermöglichen die Visualisierung dynamischer molekularer Datensätze mit mehr als einer Million Atomen bei interaktiven Frameraten durch die Verwendung GPU-basierten Raycastings. Aufgrund der Einschränkungen, welche in einer Browser-basierten Umgebung vorliegen, musste die konkrete

Implementierung des GPU-basierten Raycastings angepasst werden. Die Evaluation der daraus resultierenden Performanz zeigt, dass GPU-basierte Techniken das interaktive Rendering von großen Datensätzen ermöglichen und eine im Vergleich zu Polygon-basierten Techniken höhere Bildqualität erreichen.

Zur Verringerung der Übertragungszeiten, Reduktion der Latenz und Verbesserung der Darstellungsgeschwindigkeit werden effiziente Ansätze zur Datenrepräsentation und übertragung verwendet. Des Weiteren wird in dieser Doktorarbeit eine GPU-basierte Volumen-Ray-Marching-Technik auf Basis von WebGL 2.0 eingeführt, welche progressive blockweise Datenübertragung verwendet, sowie verschiedene Detailgrade, um ein interaktives Volumenrendering von auf dem Server gespeicherten Datensätzen zu erreichen.

Die in dieser Doktorarbeit präsentierten Konzepte und Resultate tragen zur weiteren Verbreitung von interaktiver web-basierter Visualisierung bei. Die erzielten algorithmischen und technologischen Fortschritte bilden eine Grundlage für weiterführende Entwicklungen von interaktiven Visualisierungsanwendungen auf Browser-Basis. Gleichzeitig hat dieser Ansatz das Potential, zukünftig kollaborative Visualisierung in der Cloud zu ermöglichen.

CHAPTER 1

Introduction

Visualization has established itself as a critical component in the data analysis pipeline. Large data sets from scientific experiments, medical scanners, sensors, and numerical simulations are hard to understand without the aid of visualization tools. Normally a user (typically a scientist or an engineer) would download data to his or her PC to visualize the data using a desktop-based visualization tool. However, this work flow is becoming infeasible due to increasingly large data sets that are difficult or impossible to move. This has been due to technological advancements in computing power that have enabled complex problems and high resolution models to be simulated, thus generating huge amounts of data. Additionally, these data may be stored in machines that are distributed, for example, in a cluster, grid, or cloud. The main challenge for visualization tools is how to visualize and gain insight from these data. Another challenge is the trend towards collaborative research involving teams that are geographically distributed but working on a single problem. These researchers could benefit from visualization tools that can be accessible easily without requiring installation of software in the machine of each member, who may be using different computing platforms.

In response to these challenges, researchers have looked at remote visualization as a viable approach for addressing the problems of large data visualization. Different client-server approaches have been proposed, which will be discussed in detail in chapter 2 and chapter 3.

On the server side, computing cluster, grids, and recently clouds have been exploited to scale the visualization algorithms. On the client side, browser-based

techniques have also received considerable attention due to their ubiquity and being cross-platform. These attributes make browsers important because of their potential to alleviate problems associated with software maintenance that sometimes take up much research time. Additionally, the browser can be used as a deployment platform for collaborative visualization tools.

Despite these attractive features, until recently, lack of powerful browser technologies was a limiting factor for interactive visualization. Due to this limitation, early approaches in web-based visualization have relied on server-side rendering and used the browser for displaying the rendered images.

Another factor that has motivated server-side rendering has been lack of powerful computational resources on the client. However, this approach suffers from limited network bandwidth and latency issues. These two factors are critical for interactive visualization. By rendering the images on the server, any changes on rendering or visualization parameters on the client have to be sent to the server to generate a new image, thus incurring round-trip network latency.

In order to achieve high-performance interactive visualization, minimizing latency is important. The best approach to achieve this, is to perform rendering on the client machine. This is because rendering is a critical step in the visualization pipeline for interactive visualization. By rendering the data on the client, the generation of a new image does not require a round trip to the server that would increase both network bandwidth and latency. Other approaches have attempted to use plugins to circumvent browser limitations in order to provide interactivity with limited success due to browser compatibility problems and potential security dangers that they pose [Heule et al., 2015; Labour et al., 2013; Carlini et al., 2012].

Recent advances in web technologies like HTML5 [W3C, 2017] and WebGL [Khronos, 2013, 2011b] combined with improvements in mobile hardware has brought a renewed interest in interactive web-based visualization. By providing GPU-access in the browser through WebGL, visualization techniques that were only confined to the desktop platforms can be deployed on the browsers and benefit from its ubiquity and leverage other web standards that have led to the success of the web as a global distributed system.

JavaScript, the programming language of the web, has also improved significantly in terms of performance by employing just-in-time (JIT) compilation techniques that have largely narrowed the performance gap with the native compiled languages [Jeon and Choi, 2012; Gal et al., 2009].

The availability of the browser in a wide range of devices from smartphones and tablets over laptops equipped with multi-core CPUs and GPUs rivaling those found on desktops of few years ago, to desktops, provide a common

platform for harnessing the power of these distributed and mobile computing resources.

Current visualization techniques exploit the highly parallel processing power of GPUs to achieve high performance rendering for large datasets. For example, GPU-based ray casting is considered the state-of-the-art technique for molecular visualization [Grottel et al., 2015]. This technique allows the rendering of implicitly defined surfaces on the GPU instead of generating corresponding geometry. It achieves high interactive frame rates even for very large molecules and generates high-quality images compared to polygon-based rendering techniques. Another advantage is, that it is less demanding in terms of memory bandwidth because it requires only the parameters defining the implicit surface to be uploaded to the GPU.

Although many web-based visualization tools that exploit HTML5 and WebGL have been introduced, they often still rely on polygon rendering techniques. For example, most existing web-based protein viewers use triangulation techniques to approximate the spherical surface patches, thus limiting the quality of images rendered and visualization of molecules with only small number of atoms [Li et al., 2014; Rego and Koes, 2015; Pettit and Marioni, 2013; JSmol, 2013]. Supporting visualization of large dynamic datasets becomes even more challenging with this approach, as huge amount of triangles create huge demand for storage and bandwidth.

This thesis demonstrates the feasibility of visualizing large dynamic datasets in the browser by leveraging modern web technologies and using GPU-based ray casting techniques without the use of plugins. Implementations of these techniques use molecular data from the protein data bank [Berman et al., 2000] for static data and from molecular simulations for dynamic data. Furthermore, the thesis demonstrates the feasibility of web-based volume rendering using multi-resolution and bricked volume representation with importance-based data transfer allowing progressive visualization of volumetric datasets in the browser.

1.1 Thesis Contributions

The main contribution of this thesis is the application of GPU-based ray casting techniques combined with modern web technologies to enable interactive visualization of large dynamic molecular data and the use of a combined multi-resolution and bricked volume data representation for interactive volume rendering in the browser. Specifically, the thesis

- introduces GPU-based ray casting techniques for interactive visualization

of large dynamic molecular data tailored to the specific restrictions of the browser environment,

- presents and discusses performance results that demonstrate the feasibility of visualizing large dynamic molecular data at interactive frame rates in the browser,
- describes the implementation of GPU-based ray casting with a 3D uniform grid-based acceleration structure in a WebGL 1.0 environment with limited hardware capabilities like lack of 3D textures and fragment depth writes,
- describes the implementation of progressive rendering of large volumetric data using bricked multi-resolution volume representation and importance-based data transfer in the browser.

1.2 Structure of the Thesis

This section gives an overview of the structure of this thesis and summarizes the contributions of the author to the discussed publications. All subsequent publications were co-authored by the author's PhD advisor Thomas Ertl. Unless noted otherwise, the co-authors mentioned below were affiliated with the Visualization Research Center of the University of Stuttgart (VISUS) at the time of the collaboration.

Chapter 2 introduces several topics that are important for understanding the thesis. Since most of the implementations in this thesis have used molecular and volumetric data, introduction to both molecular and volumetric data visualization is given. HTML5, WebGL, and other important web technologies that have been used to implement the techniques described in this thesis are also briefly discussed. The chapter then discusses the WebGL rendering pipeline. Finally, declarative and imperative approaches to 3D web graphics are presented.

Chapter 3 gives an overview of previous work in the area of remote visualization focusing on web-based, grid-based, and cloud-based visualization, and web services. Since computing resources employed in remote visualization are distributed and heterogeneous, research on this topic has also attempted to apply the concept of web services to visualization in these computing environments. This chapter is based on a survey paper [Mwalongo et al., 2016b], for which the author of this thesis selected the literature and wrote major parts of the text. The chapter discusses both early approaches using SOAP-based web services and the recent trend towards RESTful web services. It also discusses efficient data encoding and transfer techniques for interactive web-based visualization. Local rendering in the browser covering polygon rendering and GPU-ray casting rendering approaches are also discussed. Finally, the chapter

introduces visualization applications in various domains including volume, geospatial, particle, and information visualization.

Chapter 4 addresses the problem of visualizing large static molecular datasets in the browser using GPU-based ray casting. It uses a 3D uniform acceleration data structure to speed-up ray-sphere intersections. Different approaches for generating the acceleration structure on the client side and on the server are implemented and their performance evaluated. Implementation details to circumvent the limitations of WebGL are also presented. Finally, the chapter discusses rendering and data transfer performance results. The results presented in this chapter were published at the Web3D conference [Mwalongo et al., 2014]. The project was a joint work between the author of this thesis and Michael Krone, Guido Reina, Michael Becher and Grzegorz Karch. The author of this thesis was the lead author of the paper and contributed major parts of the prototype implementation used for testing. The volume rendering was contributed by Michael Becher as part of his bachelor thesis, which was supervised by Grzegorz Karch and the author of this thesis.

Chapter 5 presents new techniques for the visualization of large dynamic molecular datasets in the browser. It discusses efficient data encoding and transfer techniques that save bandwidth and minimize client side decoding time. To further reduce the amount of data transferred, quantization techniques for atom and vertex data are introduced. The chapter shows how different web technologies including web workers, typed arrays, and web sockets can be combined with WebGL to enable interactive rendering of large molecules with over one million atoms. Finally, the chapter presents rendering and data transfer performance results in both LAN and WiFi environments for various molecular trajectories. The results discussed in this chapter were published at the Web3D conference [Mwalongo et al., 2015] and an extended version of the paper was invited to a journal [Mwalongo et al., 2016a]. The project was a joint work between the author of this thesis and Michael Krone, Guido Reina, and Michael Becher. The author of this thesis was the lead author of the paper and contributed major parts of the implementation of the prototype used for testing.

Chapter 6 discusses web-based volume rendering based on progressive importance-based data transfer. The implementation uses a bricked multi-resolution volumetric data representation. The technique allows users to start viewing bricks of high importance based on user-defined importance criteria. Low-resolution brick data are given high importance to allow them to be downloaded first. These brick data are rendered immediately as they are received thus allowing the user to view a low-resolution image that gets refined as the high-resolution brick data are being downloaded asynchronously in a background

thread. This way the latency is minimized. Optional compression can be used to further reduce the amount of transferred brick data. To achieve efficient data transfer and decoding times on the client, an efficient binary serialization format for volumetric data is introduced. Finally, the chapter presents and discusses the data transfer and rendering performance results for various volume datasets. The project and the resulting VMV publication [Mwalongo et al., 2018] was a joint work between the author of this thesis and Michael Krone, and Guido Reina. The author of this thesis contributed to the idea of using levels of detail and progressive data transfer to address the problem of latency for interactive rendering volumetric data in the browser, and implemented major parts of the prototype used for testing. The idea was further refined by Michael Krone and Guido Reina by introducing bricking and importance-based progressive data transfer.

Chapter 7 sums up the main conclusions of the thesis and provides an outlook for interactive web-based visualization focusing on the role of cloud computing, web services, and modern web technologies for supporting geographically distributed collaborative visualization services and automated scientific work flows that integrate visualization services with other tools to simplify large data analysis.

CHAPTER 2

Fundamentals

This chapter introduces several topics that are important for understanding the following chapters. First, an introduction to data visualization is given. Since most of the implementations in this thesis have used molecular and volumetric data, the chapter next gives an introduction to both molecular and volumetric data visualization. There after HTML5, WebGL, and other important web technologies that have been used to implement the techniques described in this thesis are also briefly discussed. The chapter then discusses the WebGL rendering pipeline. Finally, the declarative and imperative approaches to 3D web graphics are discussed.

2.1 Introduction to Data Visualization

Data visualization is a process of creating visual representation of data using computer graphics techniques [Telea, 2014]. The data to be visualized usually come from a variety of sources like sensors and computer simulation. The data to be visualized can be spatial or non-spatial. When the data is spatial, it is usually called *scientific visualization* and for non-spatial data it is called *information visualization*. Examples of spatial data are data from physical sciences that have inherent three-dimensional coordinates in space or medical datasets while non-spatial include mainly tables, graphs, or textual data.

The purpose of visualization is to help gain insight from the data. For example, it is easy to see relationships or patterns in the data when viewed graphically rather than when the raw numeric data are presented to the user. This is due to

high bandwidth and cognitive functions of the brain dedicated to processing of visual information [Glassner, 1994].

The visualization process is usually presented as a pipeline consisting of a number of transformation stages that transform data from its raw format to a final image. These visualization pipelines serve as reference models for the visualization process. Weiskopf [Weiskopf, 2006] presents an abstract visualization pipeline that is a modification of the original visualization pipeline introduced by Haber and McNabb [Haber and McNabb, 1990]. Based on this model, there are three main stages: filtering, mapping, and rendering (see Figure 2.1). The first stage, transforms the raw data to visualization data. This stage involves data cleaning, data reduction, interpolation, resampling, data format conversions, and preprocessing computations that make the data visualization-ready. The mapping stage involves transforming the visualization data to a renderable representation. This stage involves assigning geometric primitives and other visual attributes to the data. For example, for visualizing molecular data where a molecule is made up of atoms, each atom can be mapped to a sphere and the center and the radius of the atom is mapped to the center and radius of its sphere. Moreover, each sphere can be assigned a different color depending on the respective atom. Rendering stage involves creating an image for static data or video or for dynamic data from the renderable representation created in the mapping stage. This stage results in an image or video being displayed to the user.

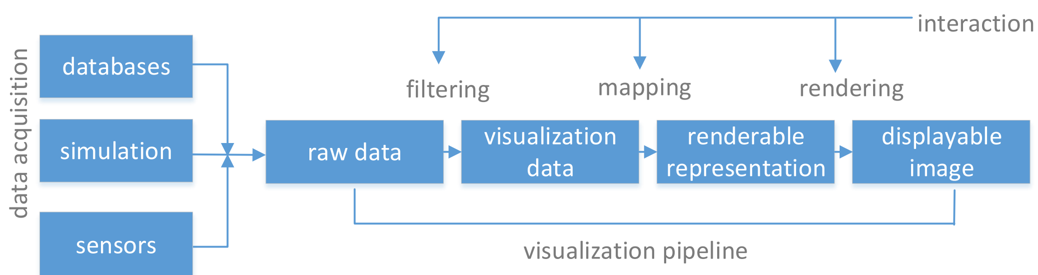


Figure 2.1 — Visualization pipeline [Weiskopf, 2006].

For interactive visualization, all these stages are interactive. This means that the user can change the parameters for each of this stage and immediately see changes in the image generated. Interactive visualization allows scientists to explore the data and to experiment with different *what-if* scenarios or questions to test the effect of different parameters to the process generating the data or the data themselves.

The work presented in this thesis deals with molecular and volume visualization. The next two sections therefore introduce concepts in these topics.

2.1.1 Molecular Data Visualization

Molecular simulations play an important role in the study of the dynamics of biomolecules. The data produced by these simulations are important in understanding their functions. Molecular visualization is crucial to analyze and gain insight from these data, particularly how structure relates to the functions of these molecules.

The result from these molecular simulations is usually a set of coordinates that show the positions of atoms of the molecule at different time steps of the simulation. The entire data set is called *a trajectory* that is usually written to a file by the simulation program after every time step or after several time steps. This trajectory file can later be loaded into a visualization tool and played back as an animation for dynamic data or a single snapshot can be visualized as a static image for static data. Figure 2.2 shows visualizations produced by some web-based molecular visualization tools.

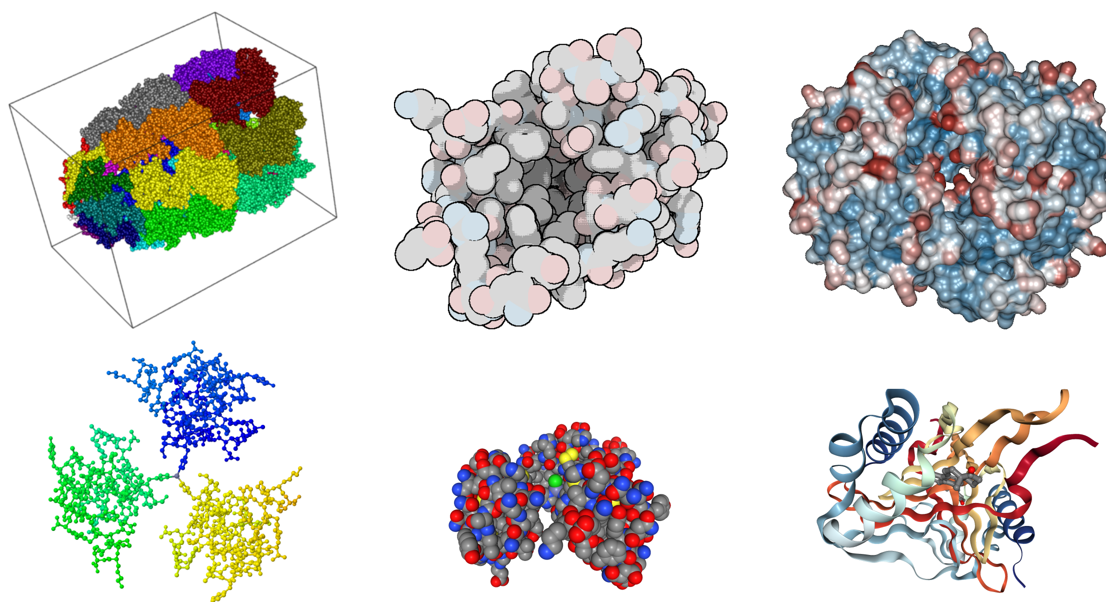


Figure 2.2 — Example visualizations produced by some web-based molecular visualization tools: *top row* [Mwalongo et al., 2014, 2015], *bottom row-first* JSmol [JSmol, 2013], and *bottom row-second and third* NGL Viewer [Rose and Hildebrand, 2015].

Apart from visualizing the dynamics of the molecules from a simulation, their structures can be visualized using publicly available structure data. A common

source for structure data is the protein data bank [Berman et al., 2000]. This structure data also acts as a source for molecular simulations, providing the initial starting atom coordinates.

Several three dimensional models for the visual representation of molecular structure have been introduced. Simple models include space-filling and ball-and-stick. There are also surface models called *Solvent-Excluded Surfaces (SES)* and *Solvent Accessible Surface (SAS)* that show the surface of the molecule not accessible by another molecule and a surface accessible by another molecule respectively. These models depict different structural attributes of the molecule. The choice of which model is used, depends on the analysis task at hand. A detailed discussion of these different molecular model representations is found in a survey paper by Kozlíková et al. [2016]. Although visualization of the simulation results is usually done as a post-processing step after the simulation has stopped, sometimes the visualization application can be connected directly to the simulation application in an approach that is called *computational steering*. Here the user can drive the simulation by changing the parameters of the simulation based on the results of the visualization of data from previous simulation time steps. This is also advantageous especially for long running simulations. For example, corrective measures can be taken to stop the simulation early in case of incorrect computations due to wrong parameters or misconfiguration.

Connecting visualization and simulation applications can also be done to avoid moving huge data to a different machine for visualization in a post-processing step. This approach, also known as *in-situ visualization* allows the visualization of data to be done on the same machine where the simulation runs. With the huge data from complex simulations that cannot be easily moved, this approach is increasingly gaining attention as a research area in both computational and visualization communities.

Since, several existing molecular visualization tools have been developed only for desktops (e.g., VMD [Humphrey et al., 1996], MegaMol [Grottel et al., 2015]), visualization researchers have long been interested in using the browser as a deployment platform for molecular viewers due to its ubiquity across devices. Popular among these web-based molecular viewers is JMol [Jmol, 2009], which is available as a Java applet and installed in the browser as a plugin. However, plugins need regular maintenance on every machine they are installed on (e.g., installing security updates), thus creating installation and maintenance overhead from the client perspective.

These limitations of the browser have forced many techniques to rely on server-side rendering and sending images to the client only for display. This approach, however, suffers from lack of interactivity. Furthermore, the use of plugins in the

browser has created security loopholes, leading to many modern browsers to discontinue plugins in the browser. Another weakness was on the performance of JavaScript, the language of the web. Its single-threaded and interpreted nature made it unsuitable for the demands of visualization applications on the web.

Despite these limitations, the ubiquity of the web browser has remained attractive as a favorable cross-platform deployment environment for molecular visualization tools and as a potential platform for global collaborative visualization [Viegas et al., 2007; Isenberg et al., 2011]. This attractiveness of the browser has led to efforts for improving the performance of JavaScript and introduction of modern web technologies that meet the demands of interactive visualization applications on the web. We discuss these technologies further in section 2.2.2 and section 2.3.

2.1.2 GPU-based Volume Visualization

Volume data is usually defined as scalar field in 3D space. Volume visualization deals with techniques that generate 2D images from these volumetric data for the purpose of gaining insight. These data usually come from medical scanners like CT and MRI or computer simulations that generate 3D data.

GPU-based volume visualization leverages the computational power of the GPU in order to efficiently render these volumetric datasets. Although there are several techniques for visualizing volumetric data, for the purpose of this dissertation, we only focus on GPU-based volume ray casting also known as *GPU-based volume ray marching*, which is currently the state-of-the-art method for rendering volumetric data. Therefore, our technique for web-based volume visualization, discussed in chapter 6 is based on this rendering technique. The book by Engel et al. [2006] provides a thorough discussion on GPU-based volume visualization techniques.

GPU-based volume ray marching is the state-of-the-art approach for direct volume rendering due to its performance and high image quality. This technique uses 3D textures (assuming the use of WebGL 2.0) or a set of 2D textures (when using WebGL 1.0) for data storage and requires support for dynamic looping in the fragment shader. This technique initializes rendering by first rendering a proxy geometry, usually the bounding box of the volume data. There are two approaches for implementing this technique in WebGL: multi-pass and single-pass GPU-based ray marching.

In the multi-pass approach, the proxy geometry is rendered into a texture in order to get the entry and exit points of the rays in the volume data. These rays are then used during the volume traversal pass in order to get the starting

point and the direction of each of the sampling rays. In the fragment shader, the volume is sampled along the rays, classified, optionally shaded, and iteratively composited to get the final color of the pixel.

In the single-pass approach, either the front or the back faces of the volume's bounding box are rasterized. Ray traversal is also performed in the fragment shader by marching along the view ray in discrete steps through the volume and compositing the sample colors to get the final pixel color. Figure 2.3 shows an overview of the volume ray marching rendering technique.

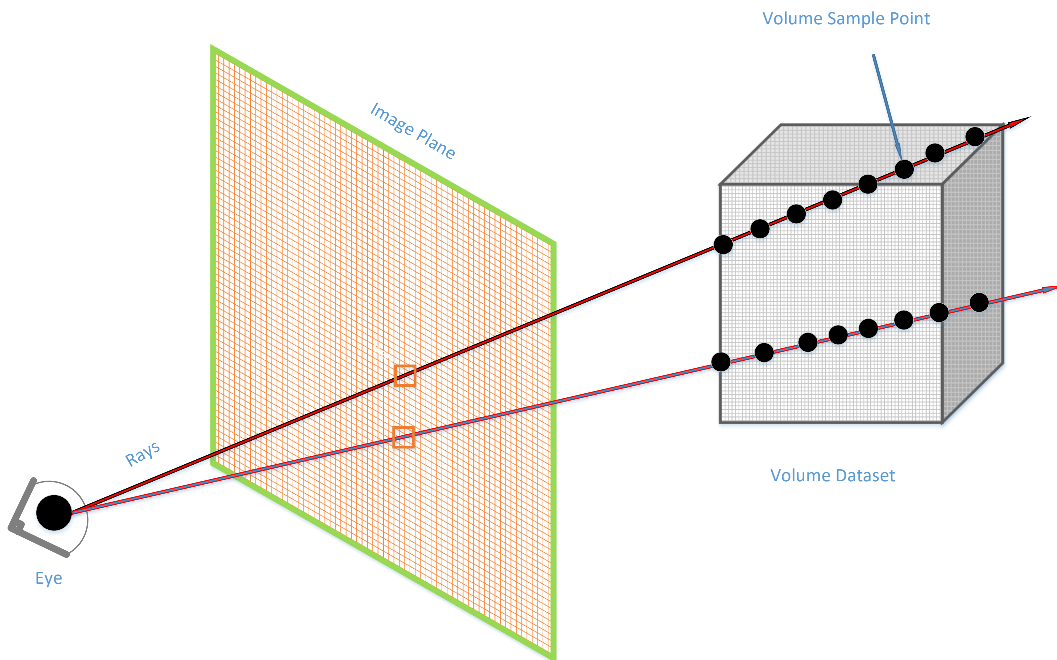


Figure 2.3 — The volume ray marching rendering technique. A ray is cast from the eye position for each pixel towards the center of each pixel in the image plane and the volume is sampled along the ray. Each sample is shaded and finally all samples are composited to get the final color of the pixel. For GPU-based volume rendering, this technique is implemented in a fragment shader.

2.2 OpenGL and WebGL Rendering

As discussed in Section 2.1, rendering is the last stage in the visualization pipeline. The performance of this stage is very crucial for interactive visualization. Modern interactive visualization applications exploit the power of

Graphics Processing Units (GPUs) in order to achieve better performance. Access to the GPU is through standards-based application programming interfaces (APIs) like OpenGL. Our work, in particular, uses WebGL [Khronos, 2011b], which is based on OpenGL ES which in turn is based on OpenGL (see Figure 2.4). The next sections provide an overview on the rendering pipelines of these APIs.

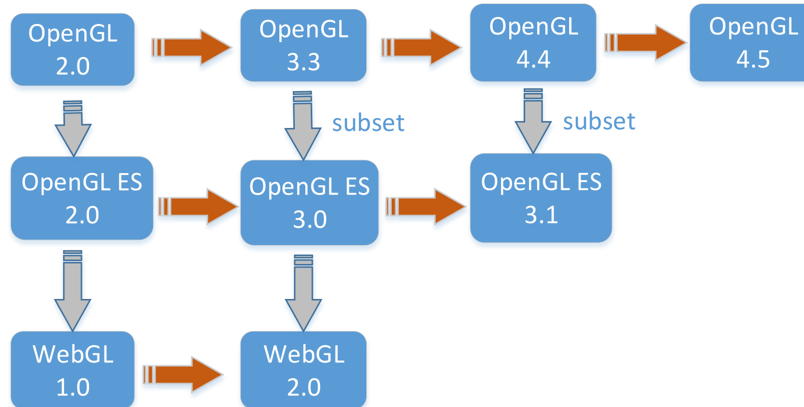


Figure 2.4 — Relationship between OpenGL, OpenGL ES and WebGL (*adapted from*: Matsuda and Lea [2013]).

2.2.1 OpenGL Rendering Pipeline

Open GL is an application programming interface (API) that allows graphics applications to access graphics hardware for image rendering. Modern graphics processing units (GPUs) are programmable, therefore OpenGL also has programmable stages in its rendering pipeline, which allow users to write shader programs that are executed on the GPUs. The standard is divided into three main groups: OpenGL for the desktop, OpenGL ES for embedded devices, and WebGL for the browser. The relationship between these different standards is as shown in Figure 2.4. In contrast to the OpenGL for desktop which has more programmable stages in its rendering pipeline, OpenGL ES versions upon which WebGL 1.0 and WebGL 2.0 are based, support only two programmable stages: vertex and fragment processing. OpenGL ES 3.0 supports shading language versions 3.0 and 1.0, in contrast to OpenGL ES 2.0 which supports only shading language version 1.0.

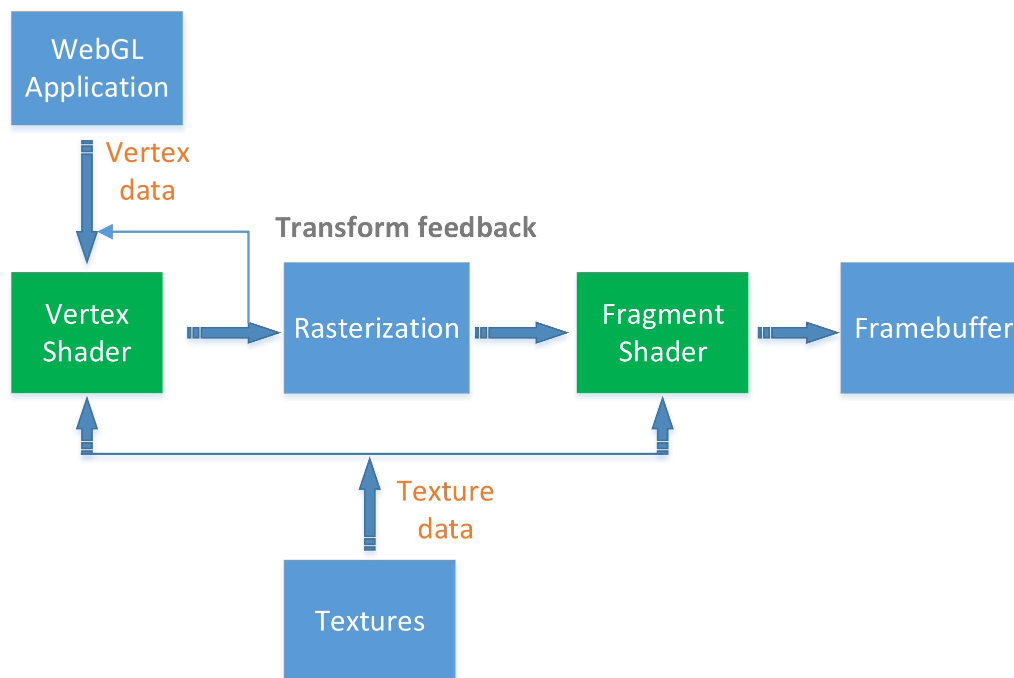


Figure 2.5 — A Simplified WebGL 2.0 rendering pipeline. WebGL 1.0 has similar pipeline, except that it does not support *Transform feedback*. Pipeline stages colored in green are programmable.

2.2.2 WebGL Rendering Pipeline

WebGL provides a JavaScript API to the GPU based on the OpenGL ES standard. Currently, browsers implement WebGL 1.0 based on OpenGL ES 2.0 and WebGL 2.0 (based on OpenGL ES 3.0). WebGL 2.0 supports OpenGL ES Shading Language (GLSL) 3.0 and 1.0, while WebGL 1.0 supports only shading language 1.0. Due to browser constraints, including security issues, however, not all features of the underlying OpenGL ES standards are exposed to WebGL. It is these constraints that make implementing visualization techniques designed for desktop computing platforms in WebGL not a straight forward task. Some adaptations are required in order for these techniques to work within the constraints of the browser environment. Both WebGL versions expose only two programmable stages of the graphics pipeline: *vertex shader* and *fragment shader* (see figure 2.5). The main computation done by the vertex shader is transformation of the vertex data received from a web application written in JavaScript from model to clip space coordinates. The output vertices from this stage are then assembled into primitives and fed into the rasterizer for fragments generation. These fragments are the inputs to the fragment shader

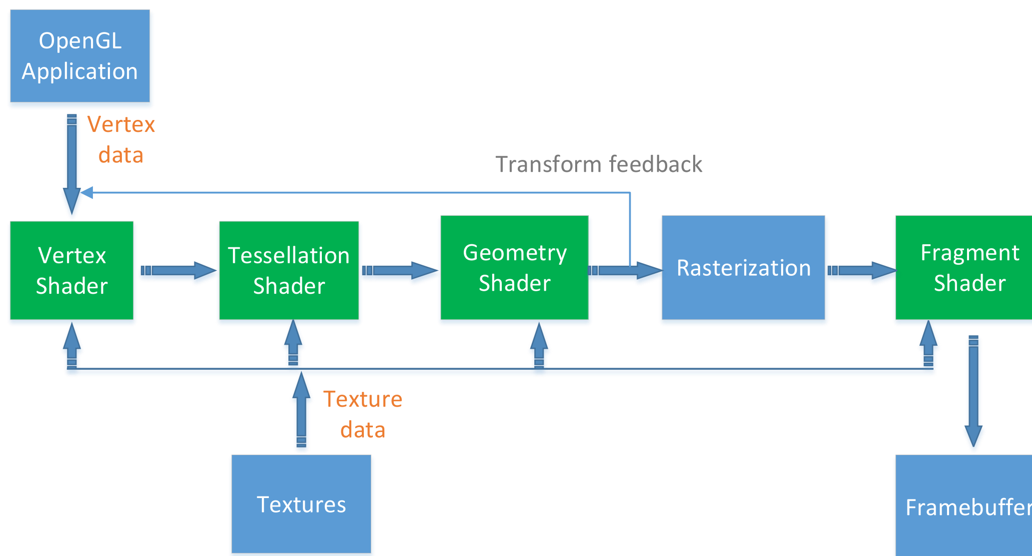


Figure 2.6 — Simplified OpenGL rendering pipeline (OpenGL 4.5), pipeline stages colored in green are programmable [Kessenich et al., 2016]).

which after processing them, writes the results into a framebuffer.

Therefore, visualization techniques exploiting the programmable graphics pipeline can only be implemented in a vertex or fragment shader. This is in contrast to OpenGL that has additional programmable stages like tessellation and geometry shaders (see Figure 2.6).

Another main difference between OpenGL and WebGL is that WebGL runs under the control of the browser and has no direct communication with the underlying operating system. Since WebGL draws on a canvas element, the browser compositor takes the contents rendered by WebGL and combines them with other visual elements of the web page to display the final page content to the user.

Apart from additional layer of indirection that may affect performance, a single frame in WebGL has less frame budget compared to a frame in OpenGL because for each frame a fraction of the time has to be reserved for compositing by the browser. Browsers, however, rely on the graphics APIs of the underlying platform for their implementations of WebGL. On Windows, the ANGLE project [Koch and Capens, 2012] provides an OpenGL ES 2.0 implementation based on Direct3D. On Linux and other platforms, ANGLE is not required as browser implementations are built on top of native OpenGL for that particular platform. Mobile browsers implement WebGL directly on top of OpenGL ES on the respective mobile device operating systems.

WebGL 2.0 [Khronos, 2013], which is based on OpenGL ES 3.0, provides more capabilities than WebGL 1.0 [Khronos, 2011b]. Some of the important additions to WebGL 2.0 that are relevant for visualization are instancing, multiple render targets, fragment depth writes and 3D textures. Instancing, fragments depth writes, and multiple render targets are supported through extensions in WebGL 1.0, but are included in the core WebGL 2.0 specification.

Instancing allows several copies of the same objects to be rendered once, thus minimizing the number of draw calls and CPU-GPU bandwidth. The geometry of each unique object is sent to the GPU only once and only per instance data like position, color, and transformations are uploaded to the GPU in every frame. This is important especially for polygon rendering where large meshes may contain many similar objects. Multiple render targets (MRTs) allow the fragment shader to write fragment data to more than one buffer at once. This is useful for visualization techniques that require multiple render passes like deferred shading.

Other visualization techniques require modifying the depth value in the fragment shader. Therefore depth write support enables such techniques to be implemented in WebGL. Support for 3D textures is also important for volume rendering where data from medical images like MRI and CT scans can easily be visualized without requiring conversion of 3D volumetric data into 2D texture atlases. 3D textures also allow trilinear interpolation to be performed directly by the hardware, thus increasing performance compared to when the interpolation is performed in user code in the fragment shader.

2.3 Modern Browsers and HTML5 Technologies

In the context of this thesis, we define a modern web browser as a browser that supports HTML5 [W3C, 2017] and WebGL [Khronos, 2011b, 2013] standards. HTML5 is the latest version of the standard for the HTML markup language at the time of writing this thesis. It introduces new elements and several web technologies that provide new capabilities for the browser. For example, the canvas element provides a programmable drawing surface through JavaScript, the video element provides native capabilities for viewing video, and the audio element provides native audio playback. Moreover, the performance of the JavaScript language itself has improved significantly due to advances in compiler technology. For example, modern JavaScript engines use Just-In-Time (JIT) compilation and optimization techniques in order to achieve high performance in terms of execution speeds.

Additionally, there are other technologies in the browser that can be exploited for optimized processing that can improve rendering speeds (see Figure 2.7).

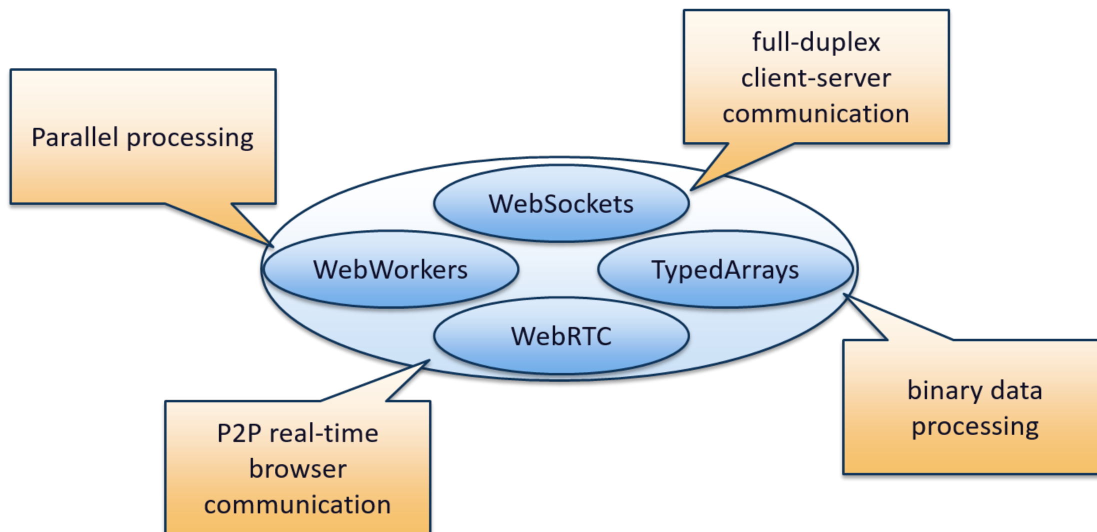


Figure 2.7 — Client-side optimization technologies in the browser.

For example, typed arrays [Khronos, 2011a]—defined as part of the JavaScript language, provide efficient binary data processing in the browser, web workers [W3C, 2015] allow long running computations to be performed without affecting the responsiveness of the application because they run on a different thread than the main thread that handles user interactions. WebSocket [Fette and Melnikov, 2011] provides bi-directional full duplex persistent communication channels between client and server. This section discusses core HTML5 elements and other web technologies that make it feasible to build interactive visualization applications in the browser.

Typed Arrays were first introduced as a separate specification to allow efficient handling of binary data in the browser for WebGL applications. Currently they are part of the JavaScript language. Raw binary data is stored in an object called *ArrayBuffer*, which is of fixed length once created. Access to this data is through views. For example, to view the data as an array of 32-bit floating point values, a *Float32Array* view is used. For a buffer that contains heterogeneous data similar to *structs* in C or C++ language, a low-level interface is provided through an object called *Dataview*. Raw binary data can be from a file or received from a network. Data encoded as an ArrayBuffer is GPU-friendly in that it can be uploaded to the GPU directly without requiring further processing on the CPU-side. This is important in the browser because heavy computations in JavaScript can affect performance significantly.

Web workers enable parallel processing in the browser, which allows long running scripts to be executed in a separate thread than the main thread. Since JavaScript is a single-threaded language, everything runs otherwise in the so

called *main thread* including handling user interactions and rendering. Also scheduling of tasks is non-preemptive, that is once a task has been started it runs up to completion. If a function in a script takes much time, it can cause the user interface to be unresponsive. To alleviate this problem, these long running scripts can be executed in the background in their own thread.

However, these threads do not share memory, each web worker runs in a separate virtual machine with its own memory and, therefore, communication between the main thread and the workers is possible only through message passing. Since current browser implementations allow WebGL to render only from the main thread, data processed in a worker has to be passed back to the main thread for rendering. This can be challenging for large data. One optimization technique for binary data encoded as ArrayBuffer is called *transferable objects*. This allows memory ownership to be transferred from a worker to the main thread and vice versa, which is also known as zero copy.

WebSocket is a communication protocol based on TCP that allows client and server to maintain full-duplex persistent communication between them. To initiate the communication, a client sends to the server a normal HTTP request, but with an *upgrade* header requesting the server to upgrade to a websocket connection if it supports it. If the server supports the websocket protocol, the connection is upgraded to become a websocket connection. This connection remains open until a client or server explicitly closes it. As long as the communication is open, both client and server can send data at the same time just like a regular TCP connection. Websocket avoids the initial cost of establishing a new connection for every request as it is the case with HTTP. The cost of initiating a connection in the case of a websocket is therefore incurred once. This is important for the visualization of dynamic data which requires continuous data updates and the ability to communicate simultaneously in both directions using a single communication channel.

WebRTC [W3C, 2018] is another technology that allows low-latency peer-to-peer real-time communication between browsers supporting video, audio, and data transfer. By combining video, audio and data communication in a unified standard, it becomes easier to build collaborative visualization applications in the browser. Although collaborative visualization is not addressed in this thesis, the techniques introduced can serve as a foundation for the development of such tools.

The <*canvas ...*> element introduced in HTML5 standard provides a surface for graphics drawing through JavaScript. Currently, it supports 2D and WebGL rendering contexts. The 2D rendering context is CPU-based while WebGL provides an OpenGL ES based API for GPU-access through JavaScript.

Since WebGL is a low-level API, many libraries and frameworks have been built to provide high-level functionality on top of it to make it easier to develop applications that leverage the power of the GPU. Frameworks include X3DOM [Behr et al., 2010] and XML3D [Sons et al., 2010] that aim at integrating 3D graphics with the HTML5 document object model (DOM). On the part of libraries, Three.js [Threejs, 2010] and BabylonJS [BabylonJS, 2013] are some of the popular JavaScript libraries that provide a scene graph-based API on top of WebGL. A survey paper on 3D web graphics by Evans et al. [Evans et al., 2014] discusses these libraries and frameworks in more details.

Another web technology is the SVG standard upon which other high level visualization tools like D3.js [Bostock et al., 2011] are built. D3.js is a popular JavaScript library for information visualization that builds on SVG and 2D canvas standards. The functionality of the SVG standard can be included in an HTML document through the HTML5 <*svg ...*> element tag. Depending on browser implementations, these elements can be hardware accelerated in order to improve performance.

2.4 Approaches for 3D Graphics in the Browser

Main approaches for 3D graphics in the browser can be categorized as declarative or imperative. Declarative approaches try to extend the HTML markup language to support custom markup elements that allow 3D content authors to define their scenes and embed them in the browser for rendering without the need for programming. Prominent examples for these approaches are X3DOM [Behr et al., 2010] and XML3D [Sons et al., 2010]. These two technologies use XML as the format for encoding the model of the scene to be rendered as a *scene graph* and are currently implemented on top of WebGL. Their architectures are as shown in Figure 2.8.

Figure 2.9 shows how X3DOM can be integrated with a HTML page for rendering simple 3D shapes. X3DOM can also be used for visualization of various datasets as shown in Figure 2.10

Despite the simplicity of declarative approaches as far as programming is concerned, these approaches suffer from computational and storage overhead by using XML as the data format to encode the model to be rendered. Moreover, declarative approaches may be too restrictive for applications which require low-level access to the GPU.

On the other hand, WebGL [Khronos, 2011b] is an imperative approach to 3D web graphics that provides direct access to the GPU. Although the low-level graphics API requires more programming skills compared to declarative approaches, it offers more flexibility and can handle rendering of even larger

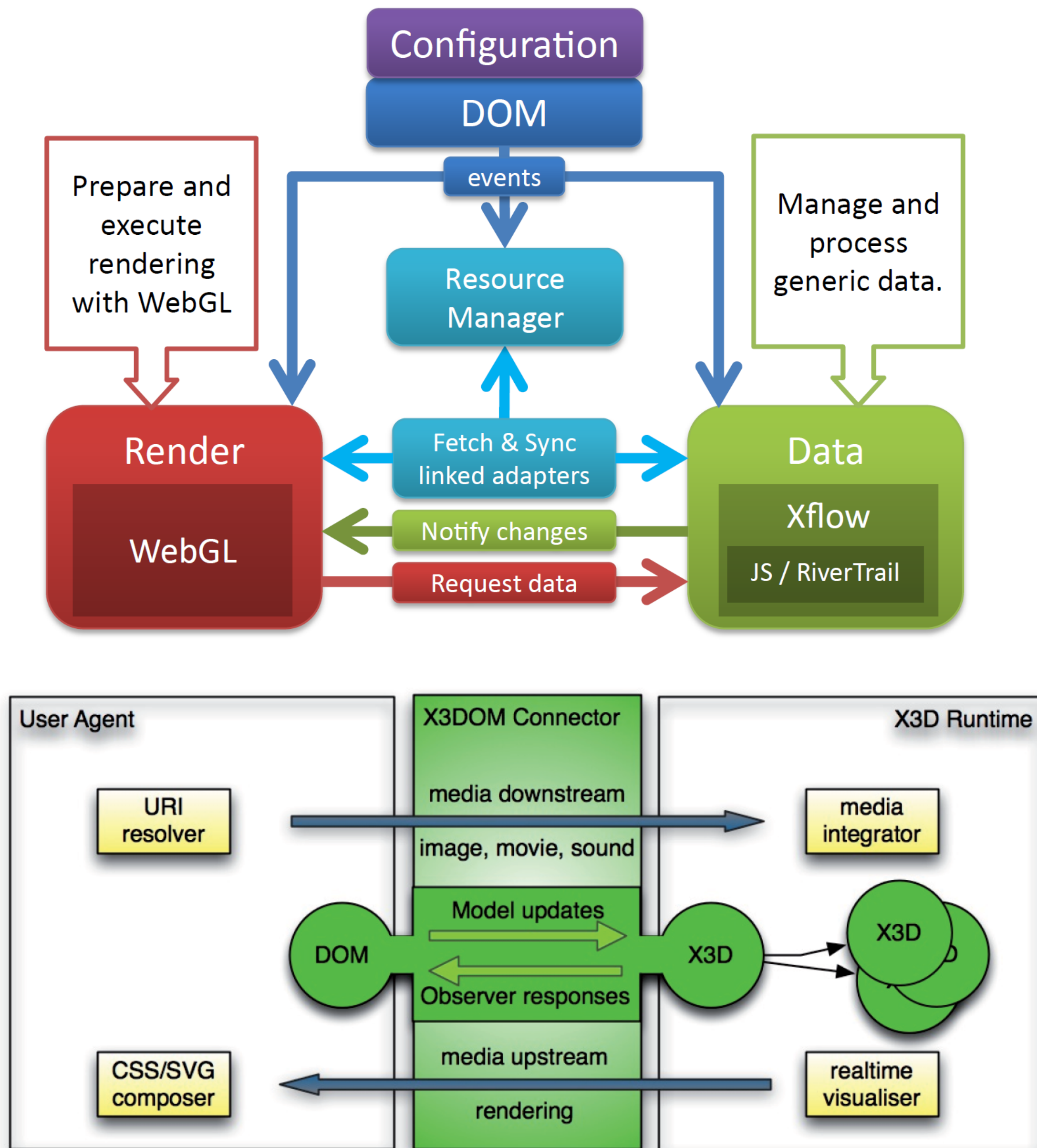
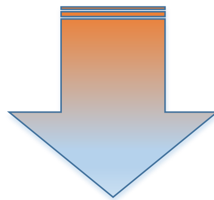


Figure 2.8 — Top: Architecture of XML3D (image: [Sons et al., 2013] © 2013 IEEE). Bottom: Architecture of X3DOM (image: [Behr et al., 2010] © 2010 ACM).

amounts of data. By providing direct access to the GPU, efficient processing can be achieved by encoding data in binary formats that can be uploaded to the GPU with minimal processing in JavaScript, thus saving time and space. The work described in this thesis uses WebGL directly due to its flexibility and low-level powerful access to the GPU that allows rendering of large data at interactive frame rates.

```
1  <!DOCTYPE html>
2  <html>
3          <head>
4              <title> Hello X3DOM! </title>
5              <script src='lib/x3dom.js'> </script>
6              <link rel='stylesheet' type='text/css' href='css/x3dom.css'></link>
7          </head>
8          <body>
9              <x3d width='600px' height='400px'>
10                 <scene>
11                     <shape>
12                         <appearance>
13                         <material diffuseColor='0 1 0'></material>
14                         </appearance>
15                         <box></box>
16                     </shape>
17                     <transform translation='3 0 0'>
18                      <shape>
19                         <appearance>
20                         <material diffuseColor='1 0 0'></material>
21                         </appearance>
22                         <sphere></sphere>
23                     </shape>
24                     </transform>
25                 </scene>
26             </x3d>
27         </body>
28     </html>
```



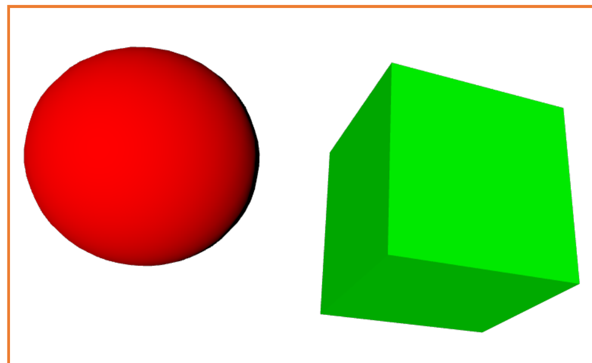
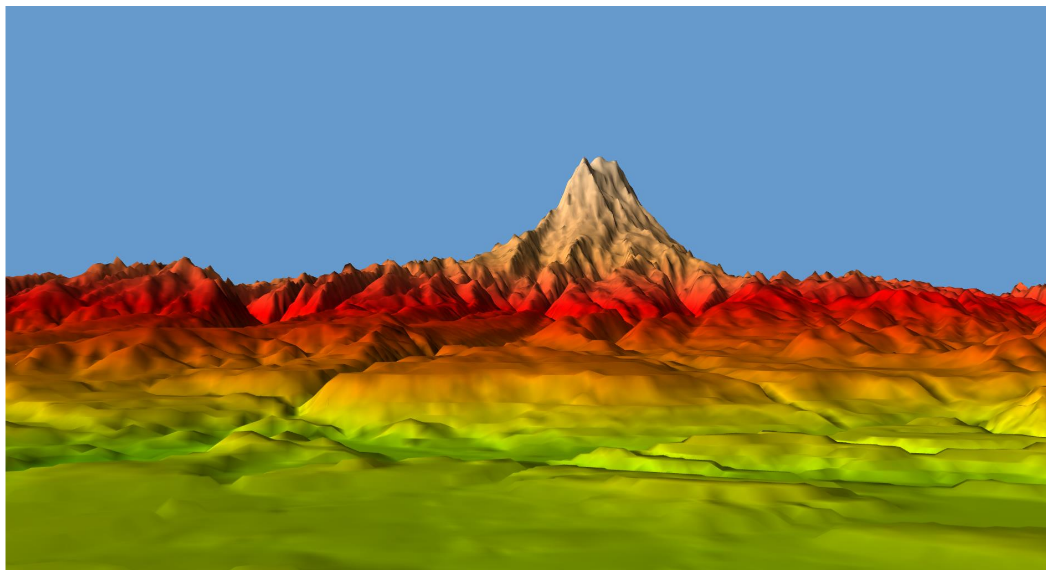


Figure 2.9 — Example of rendering simple 3D shapes using the X3DOM library (*rotated*).



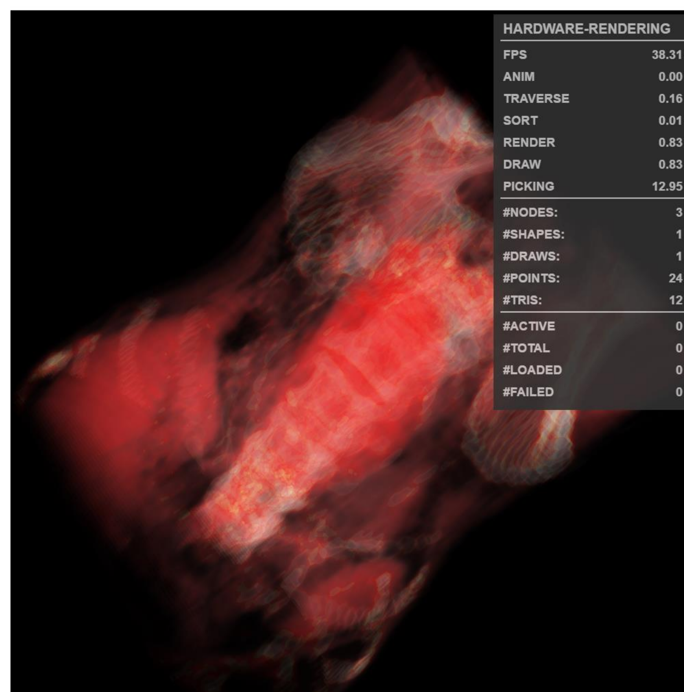


Figure 2.10 — Example visualizations using X3DOM: terrain of Puget sound [X3DOM, 2017a] (Top), volume rendering of medical dataset [X3DOM, 2017b] (Bottom).

CHAPTER 3

Remote Visualization

This chapter discusses previous work in the area of remote visualization with special emphasis on interactive web-based visualization. The work is based on our recent survey paper on web-based visualization [Mwalongo et al., 2016b].

3.1 Motivation

Remote visualization has long been an important area of research motivated by the need to visualize huge amounts of data ranging from simulations over medical imaging to big data from social media or business intelligence. This need arises from insufficient computational resources at the user's side (client). The reason can either be relatively thin clients, like mobile phones, or the sheer amount of data to process.

Even if the client were powerful enough, moving the full data set to the client quickly becomes impractical due to bandwidth, latency or local storage limitations. In certain scenarios, making the raw data available might also present legal or privacy issues. Here, remote visualization becomes a viable solution.

The need for data sharing and collaboration among geographically distributed teams has also been a driving force behind remote visualization. Although there have been rapid technological advances in end user devices (e.g., smart phones, tablets, laptops, and desktops with multiple cores and GPUs), the problem of large data still holds, and in fact, it is growing rapidly because the network

speed and latency improvements do not grow proportionally with processor speeds.

Web-based visualization in particular has received much attention due to its ubiquity across platforms, ranging from desktop computers to smartphones, and its potential as a collaborative platform [Isenberg et al., 2011].

Web-based approaches can allow visualization tools to be deployed across devices from single source code base. This does not only allow easy collaboration and sharing of visualization tools among different teams, but also simplifies software maintenance issues, allowing visualization researchers and domain scientists to focus more on the core issues of their research.

Moreover, this deployment model allows domain scientists to access latest visualization techniques (e.g., by simply refreshing a page). This close collaboration between visualization researchers and domain scientists is also important for the visualization research community to avoid the danger of loosing its users [Lorensen, 2004].

Another advantage of web-based solutions for the client-side is that users (e.g., domain scientists) do not need to install software, they just need a browser. Since browsers are currently deployed in many devices, they give more flexibility for the user to work from any device, anywhere, as long as there is an Internet connection. As many visualization solutions are currently GPU-based, program shaders (which are simple text files) can be hosted on a server together with the data and easily shared by multiple clients. This setup is attractive even in cases where the data is not shared. A user with data on a local machine can simply get the visualization code from the server and visualize the data locally instead of uploading the data to the server, which may be expensive for large datasets or overload the server when many users are uploading their datasets. For large datasets, moving computation (code) to the data is cheaper than moving the data to the computation [Gray et al., 2005].

Despite efforts of previous research, limited bandwidth and network latency are still the main challenges for remote visualization. The continuous growth of data sets and gradual improvement of infrastructure and client devices make remote visualization still an active research area. The current trend is towards exploiting modern computation and networking infrastructure combined with efficient data transmission techniques to enable interactive visualization in the browser. Although compression has been employed for efficient use of network bandwidth, low latencies can best be achieved through client side rendering. Therefore GPU access in the browser through WebGL [Khronos, 2013, 2011b] and HTML5 technologies [W3C, 2017] are an important addition for enabling high-performance interactive rendering in the browser.

3.2 Remote Rendering Approaches

Visualization tools have been developed following a pipeline model first introduced by Haber and McNabb [1990]. Whereas in a traditional visualization, all the stages of the pipeline run in a single machine, in remote visualization, any part of the pipeline can run on a different machine. This is usually in a client-server architecture where part of the pipeline is executed on the server and the other parts on the client.

Strategies for partitioning the visualization pipeline can broadly be divided into three categories depending on where the rendering stage of the pipeline is executed. In the first strategy, the rendering is executed at the server, called *server-side rendering*. In this approach, the images (for static data) or video streams (for dynamic data) are generated at the server and sent to the client for display. The work of Yoo et al. [2010] and Jomier et al. [2011] employ this approach. Many other early techniques followed this approach [Engel et al., 2000, 1999; Engel and Ertl, 1999; Hendin et al., 1998; Trapp and Pagendarm, 1997; Wood et al., 1996]. These approaches use a combination of VRML, Java applets, Flash, or JavaScript on the client side. On the server, some techniques employ hardware-accelerated techniques to achieve better rendering performance.

Historically, server-side rendering was favored, mainly due to limitations in client side computation power. Despite the advantage that this approach has minimum requirements on the client, it can lead to poor performance due to bandwidth and latency issues. This approach is therefore not well suited for interactive visualization.

A second strategy is to render the images at the client, called *client-side rendering*. In this approach the server provides the raw data (sometimes a subset of it) for rendering at the client. Some of the recent work based on this strategy for volume data are by Movania and Feng [2012], and Congote et al. [2011]. Some earlier works also follow this approach [Lluch et al., 2006; Diepstraten et al., 2004; Engel et al., 2000, 1999; Engel and Ertl, 1999]. The advantage of this strategy is that by performing rendering on the client, the interactivity is typically improved as the round-trip latency to the server whenever a viewpoint or other rendering parameters change is avoided. However, this approach demands capable computational resources on the client for interactive rendering. Luckily, recent advances in graphics hardware have brought more computational processing power to commodity hardware including mobile devices, making client-side rendering an attractive approach in both native and web platforms. These advances in GPU-technology have improved the processing power on mobile devices and thus narrowing the gap between mobile and desktop platforms.

The third strategy is to combine both server-side and client-side rendering to benefit from resources on both ends. In this approach rendering is still done on the client, however the server does some preprocessing in order to reduce computation load at the client side. Preprocessing may involve generating acceleration data structures [Mwalongo et al., 2014], generating renderable representations (e.g., geometry and textures) [Limper et al., 2014; Sutter et al., 2014]), or packaging implicit surface parameters in GPU-friendly structures that minimize processing time at the client before rendering [Mwalongo et al., 2015]). For geometric data, streaming and progressive rendering [Limper et al., 2014; Wen et al., 2014; Ponchio and Dellepiane, 2015] with levels of detail are used in order to hide latency and improve interactive user experience. GPU-based rendering helps to improve the rendering performance and interactivity by offloading expensive computations to the GPU that would otherwise be infeasible to be done on the CPU using JavaScript.

The remote visualization problem requires many issues to be addressed from data management, encoding, communication, streaming to efficient rendering techniques [Parulkar et al., 1991; Stevens et al., 2001; Brodlie et al., 2005; Shi, 2011]. Combining server and client side processing provides the benefits of powerful computational resources available at the server and interactive rendering on the client. In order to address the needs of growing datasets and many concurrent users, the server side can benefit from grid and cloud computing resources for complex data management and analysis. Section 3.4 discusses how these technologies can be combined with client-side rendering to enable high-performance interactive visualization for even large datasets and scaling to many concurrent users. We refer to visualization in these environments as grid-based and cloud-based visualization. However, before discussing these two scenarios, we discuss visualization as a web services because the concept of web services is widely applied in these environments.

3.3 Visualization as a Web Service

The World Wide Web Consortium (W3C) defines a web service as *a software system designed to support interoperable machine-to-machine interaction over a network* [W3C, 2004]. The main goal of web services is to allow seamless interoperability between different software applications and tools that need to communicate in order to accomplish a particular task. Since the services interact based on standardized interfaces without consideration of implementation details, software written in different programming languages and running on different platforms can communicate and work together harmoniously without requiring manual integration by the user. This approach is especially important in heterogeneous distributed systems.

Main approaches for applying the concept of web services to visualization with regard to how the pipeline is partitioned can be grouped into three: in the first approach, the entire visualization pipeline is treated as a black box and an application can be viewed as a single service. The user is provided with an interface for specifying data to be visualized and interacting with the resulting visualization. Tableau Online [Tableau, 2013] and TIBCO Spotfire Cloud [TIBCO, 2014] are some of the examples that fall into this category. This approach has the advantage of being simple for the end user. However, it may limit the user from the types of visualizations that can be created.

In the second approach, each of the different stages of the visualization pipeline is implemented as an independent web service. In this approach, the entire visualization pipeline is exposed to the user who can connect different services to create pipelines in order to generate various visualizations. These services can potentially be from different providers, implemented in different languages and running on different platforms. Wang et al. [Wang et al., 2008], Zudilova-Seinstra et al. [Zudilova-Seinstra et al., 2008], and Charters et al. [Charters et al., 2004] follow this approach in the context of grid-based visualization. Although this approach provides the maximum flexibility for the end user, it may suffer from inefficiencies due to communication overheads and excessive data copying between services.

A third approach combines some of the stages of the visualization pipeline into a single service. By combining some stages into a single service, inefficiencies suffered by the first approach due to communication and data movements can be avoided and still retain the flexibility provided by the second approach that can allow users to connect various visualization services that provide more powerful visualizations. Wood et al. [Wood et al., 2008] and d'Auriol [d'Auriol, 2011] follow this approach.

Viewing a visualization application as a single service would provide a good abstraction when considering a visualization as an integral component in the entire data analysis pipeline (e.g., combining simulation, analysis, and visualization) rather than a separate application. On the other hand, mapping each stage of the visualization pipeline to a web service or combining some services into a single service may be more suitable for visualization application developers. Developers may be interested in combining different implementations of visualization techniques that provide different capabilities or exploit special hardware features for each stage of the visualization pipeline to create a new composite visualization service for end users.

Another important aspect is the handling of data movement between web services in a visualization pipeline. Web services can exchange data directly with each other or do so through a centralized data store. Koulouzis et al. [Koulouzis

et al., 2010] compare the performance of these two data movement approaches and show that direct data exchange between services provides better performance compared to doing it through a centralized data store.

On the implementation side, two main approaches are used, SOAP-based web services [Seely, 2001] and RESTful web services [Pautasso, 2014], based on an architectural style first introduced by Fielding [2000]. Current trends [Webber et al., 2010; Wilde and Pautasso, 2011; Richardson et al., 2013] favor RESTful web services rather than SOAP-based services due to their simplicity and easy integration with other web standards. Moreover, by using web standards, RESTful services can exploit the already existing web infrastructure for scalability and performance. Pautasso et al. [2008] give a detailed architectural comparison between these two approaches. Despite the popularity of RESTful web services in the business domain and cloud-based services, this approach has not yet caught much attention in the visualization community. We discuss visualization as a web service in the context of grid and cloud computing environments in Section 3.4.1 and Section 3.4.2 respectively.

Web service approaches allow different tools to be designed in such a way that they can communicate and exchange data in standardized ways, irrespective of their implementation language, platform, or device where they are deployed. This would allow to easily automate the data analysis pipeline (where visualization is just a component). Components need only agree on a communication protocol and keep their internal implementation details to themselves. Moreover, decoupling the interface from implementation details allows each component to be implemented and independently optimized based on its specific task that it performs or special hardware features that it exploits.

Existing work on visualization as a service has focused more on applying the concept of web services to visualization software alone (i.e., interoperability between visualization system components) and less attention has been paid to interoperability between visualization services with other tools in the data analysis pipeline (e.g., simulation and analysis tools/services). The increasingly huge amounts of data and complex analysis required to understand and gain insight from these data makes visualization tools alone inadequate for the task. Hence, multiple approaches need to be combined.

By providing programming language and platform independent interfaces to visualization services, it becomes easy to connect these tools together and potentially automate the creation of data analysis pipelines. This can be achieved through sophisticated visualization workflow engines that can automatically orchestrate different visualization services given only the data to be visualized and some user provided visualization parameters for the desired output visualization. This would allow scientists to focus on their domain research work,

and reduce time spent manually integrating software tools.

3.4 Scaling Server-side Infrastructure

High performance and complex simulations enabled by grid and cloud computing combined with a large number of scientific instruments (e.g., sensors, laser scanners) produce petabytes of data. These huge datasets have led to the use of cloud-based storage systems like Globus Online [Foster, 2011] and Amazon S3 [Amazon, 2010]. Traditional visualization tools designed for a single user and small data start to show their limit for data at this scale. Therefore, it becomes necessary to exploit the available grid and cloud computing resources to solve the problem of large data visualization. A single machine is no longer sufficient to visualize all the data. Similarly, it becomes overwhelming for a single user to understand all the data, thus requiring visualization tools that support collaborative visual analysis.

3.4.1 Grid-based Visualization

Grid computing is a distributed system model that allows pooling of computational and storage resources to provide a high-performance problem-solving computing infrastructure [Foster et al., 2001, 2003]. The computational resources in a grid are distributed similar to a cluster. However, while a cluster is usually built from resources that belong to a single organization, a grid combines resources from multiple organizations that may be geographically distributed [Foster and Kesselman, 1999]. Therefore resources in a grid environment are usually heterogeneous compared to those in a cluster which are generally homogeneous.

Grid-based visualization is motivated by high demands from high-performance and complex simulations that use grid resources for computation and storage of the simulation results. As the simulations and other scientific instruments produce massive data, it becomes infeasible to visualize these data with only locally available computing resources. Therefore, an infrastructure of a similar scale is required for data visualization. As computational and storage resources in the grid are by nature geographically distributed, the challenge for visualization algorithms is therefore how to efficiently use this distributed infrastructure to achieve good performance while at the same time handling network bandwidth and latency issues.

Most of the work in grid-based visualization also exploit the use of web services as a way to ensure interoperability between the heterogeneous computing resources in these environments. As discussed in section 3.3, there are several

approaches for partitioning the visualization pipeline based on web services. Charters et al. [2004] present an architecture for distributed visualization based on web services that maps each stage of the pipeline as proposed by Haber and McNabb [1990] into a separate web service (see Figure 3.1). The architecture allows each service to run on a different node in the grid that can be geographically distributed. This setup can allow each service to exploit local and specific resources for maximum performance. Visualization services exchange data with each other through data transfer services provided by the underlying grid middleware. The visualization pipeline is created and controlled through a pipeline composition tool that orchestrates different web services in order to create a particular visualization.

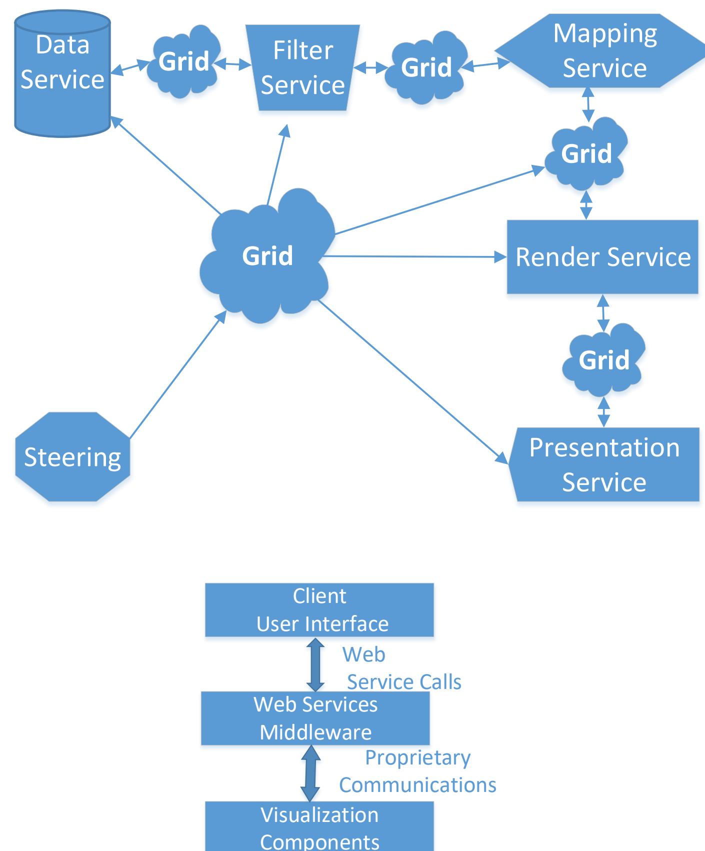


Figure 3.1 — Grid-based visualization pipeline based on web services proposed by Charters et al. (*adapted from*: Charters et al. [2004]) (Top) and by Wood et al. (*adapted from*: Wood et al. [2008])(Bottom).

An implementation of this pipeline uses SOAP-based web services [Snell et al., 2002] built on top of a grid middleware (Globus Toolkit [Foster, 2005]) and

the visualization functionality is provided by the VTK library [Schroeder et al., 2006]. Their implementation is used to demonstrate the visualization of particle simulation of dark matter, X-Ray Crystallography data, and volume data.

A similar approach to web service-based visualization pipeline is advocated by Zudilova-Seinstra et al. [2008], who apply a service-oriented approach to create an interactive visualization framework that supports visualization researchers and domain scientists in collaborative and distributed medical data analysis. Visualization experts can use the framework to create various visualizations that can then be used by medical experts to explore the data.

Wood et al. [2008] present a 3-layered architecture for web service-based visualization. The three layers are client, middleware, and visualization layer. The client layer provides the user interface functionality, the visualization layer is responsible for visualization functionality, and the middleware layer acts as an interface between the client layer and the visualization layer (see Figure 3.1).

Visualization pipelines are created using an XML-based markup language called skML [Duce and Sagar, 2005] that describes input-output relationships. The user can control the visualization pipeline through the middleware layer. Similar to Charters et al. [2004], the architecture is implemented on top of the Globus Toolkit using SOAP-based web services and VTK library for visualization functionality. Web service notification capabilities provided by the underlying grid middleware are used for data exchange between web services.

Kranzlmüller et al. [2004] present a grid visualization kernel (GVK) based on Globus Toolkit [Foster, 2005]. The authors propose a set of visualization services as an extension to grid middleware services that can be used as building blocks for creating visualization applications. To visualize data from a simulation, for example, a visualization request is sent to the GVK. The GVK initializes the visualization pipeline and remains connected with the simulation. When the simulation has new data, it sends the simulation data to the GVK to update the visualization.

To view the visualizations, a client (a user or an application interested in the visualization) sends a visualization request to the GVK. Upon receiving the request, GVK establishes a connection that links the client with the visualization pipeline connected to a simulation whose results the client has chosen to visualize. The kernel allows multiple visualization requests to be connected to a single simulation. One of the main advantages of this architecture is that it decouples the visualization clients from the simulation providing the data source. This decoupling provides much flexibility for linking different simulations and visualization clients.

Koval et al. [2015] build a web-based front-end for visualization of data resulting from a simulation running in a grid computing infrastructure. Data preprocessing on the grid is used to filter out data that is not relevant for the requested visualization in order to reduce the amount of data transferred to the client. Rendering is done on the client in WebGL using the Three.js [Threejs, 2010] library. The motivation for client-side rendering is to minimize latency and improve interactivity.

Koulouzis et al. [2010] apply web-service approach for enabling domain scientists to flexibly create visualization applications from distributed visualization pipelines in the context of medical image analysis. The use of a web-service approach is motivated by the need for interoperability of different tools in the analysis pipeline. Orchestration of the visualization is done through a visualization workflow engine that controls the execution of the pipelines. Since services do not communicate directly, this model of interaction provides more flexibility. The implementation of the visualization is based on the VTK library. Furthermore, the authors experiment with two different models of data communication between web services (i.e., data read, filter, map, and render services.): centralized and distributed communication models. In the centralized model, services communicate through a central data store; in the distributed model a data producing service and a data consuming service communicate directly. In both models, the URIs of the data resources are exchanged between services rather than the actual data for efficiency reasons.

Another difference between the centralized model and distributed model is that, in the centralized model, the workflow engine provides a URI of the central data store where the particular service should upload its results as an input parameter. In the distributed model, each service returns a URI to the work flow engine after completing its task. Other services can then use this URI to get the results. This URI is passed to the next service in the pipeline which can then read the data directly from the provided URI.

In order to benefit from the computational and storage resources of the grid and exploit client resources for interactive grid-based visualization, visualization approaches that combine the resources of the grid and client devices are preferred. Recent advances in client devices including mobile devices with relative powerful GPUs and multi-core CPUs make it feasible to offload expensive preprocessing computations to the grid and perform rendering on the client. This ensures better interactivity compared to approaches that rely on grid-based rendering with image and video streaming. Server-side rendering suffers from network latencies and limited bandwidth in cases where the images or videos generated become large in size compared to the original dataset. In this case transferring only a subset of data that is necessary for generating the required

visualization becomes attractive.

3.4.2 Cloud-based Visualization

Cloud computing is a new model of distributed computing whereby IT resources are delivered and accessed using Internet technologies [Armbrust et al., 2010; Mell and Grance, 2011]. This new model is a result of advances in virtualization, networking, and web technologies [Erl et al., 2013]. Cloud computing provides a flexible and scalable platform that can be exploited for remote and collaborative visualization. Cloud-based visualization can be considered as an evolution of grid-based visualization; they all scale the server part in the remote visualization equation and use distributed computational resources. Although grids and clouds are similar in concept, they differ in the model of management and provisioning of computational and storage resources.

Management of computational and storage resources in the cloud is based on virtualization technology. That is, the cloud management software depends on a virtualization layer controlled by a hypervisor to provide computation and storage resources to applications that run on the cloud through virtual machines. Foster et al. [2008] provide a more detailed comparison between these two paradigms.

Current visualization techniques are offered either as visualization libraries (which are programming language specific and may require significant time to learn them) or as complete visualization applications (providing a challenge to domain scientists to deal with multiple applications and learn new interfaces). This approach of delivering visualization solutions also limits domain scientists' access to latest visualization techniques and therefore denies them the opportunity to benefit from the latest advances in visualization research.

By offering visualization techniques as services and deploying them on the cloud, domain scientists can be relieved from the trouble of dealing directly with visualization applications or visualization libraries and hence concentrate on the core activities of their research. A visualization service hides all the implementation details and the various data formats and provides a simple interface through which its functionality can be accessed. To be able to communicate, the application and the visualization service need only agree on a common communication protocol. Also since the visualization services are programming language agnostic, integration of latest visualization techniques with other tools used by scientists becomes easier. The easy interoperability of visualization services with tools being used by domain scientists and exploiting cloud computing infrastructure as a deployment platform would also greatly accelerate collaborative work among domain scientists themselves and

between visualization researchers and domain scientists who are geographically dispersed.

Web services technology has been developed to address mainly the issue of application interoperability across computing platforms and programming language. As discussed in Section 3.3, several approaches have been proposed to apply web services to the design of visualization applications and different strategies for data exchange between web services.

Similar to grid-based visualization, the application of web services in the cloud is predominant. In contrast to SOAP-based web services implementations in grid-based visualization, RESTful web services implementations are common in cloud services. For example, popular commercial cloud-based visualization services (TIBCO Spotfire [TIBCO, 2014] and Tableau Online [Tableau, 2013]) provide RESTful APIs to their services.

The difference in the way computational and storage resources are managed in the grid and the cloud brings different opportunities and challenges for cloud-based visualization, different from grid-based visualization. Cloud computing works with virtualized resources managed by various cloud middleware (e.g., Open Nebula [Moreno-Vozmediano et al., 2012], OpenStack [OpenStack, 2010], Eucalyptus [Nurmi et al., 2009] for open-source cloud management software and commercial offerings from Google [Google, 2008], Amazon [Amazon, 2006], and Microsoft [Microsoft, 2010]). Because of these virtualized resources, cloud applications (cloud services) run on virtual machines and do not have direct access to the physical hardware.

These cloud management software packages in turn depend on virtualization software technologies (e.g., Microsoft HyperV, Xen, KVM, XenServer, and VmWare). Among these, Xen, XenServer, and KVM are open source. Although most of the computational and storage resources (e.g., CPUs, hard disks and other I/O devices) are well supported, virtualization of GPUs is relatively limited. Thus, it becomes challenging for cloud applications that leverage the power of the GPU for their performance. As most modern visualization techniques rely on the power of the GPU for performance, this means that GPU support in the cloud is necessary for modern visualization applications. This requirement can however have less impact on techniques that use client-side rendering, as in this case, the server side is not required to have GPU capabilities as opposed to the client where rendering is performed. Despite of this, client-side rendering can also still benefit from GPU-based preprocessing computations in the cloud to reduce the computation load on the client and therefore improve rendering performance. This can be achieved through leveraging GPGPU technologies like CUDA and OpenCL.

Currently, GPU-access is provided by only few virtualization software imple-

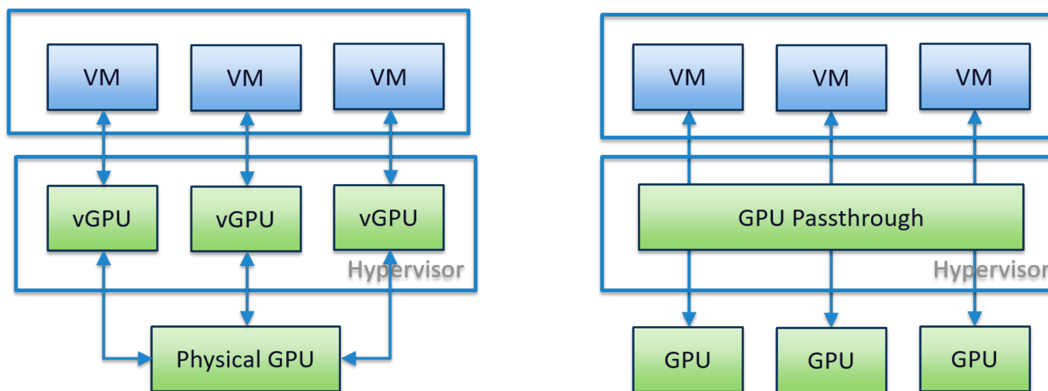


Figure 3.2 — GPU support in the cloud through virtualization software (Left) and GPU passthrough (Right). Virtualization software creates several virtual GPUs from a single GPU that can be shared by multiple concurrent users. However, this approach can perform poorly compared to GPU passthrough due to the addition of a virtualization layer on top of the physical GPU.

mentations through GPU passthrough or virtual GPU (vGPU) approaches (see Figure 3.2). In a GPU passthrough approach, a single dedicated to each virtual machine. This means that there is a one to one mapping between a virtual machine and a GPU. This offers better performance but suffers from scalability. vGPU virtualizes the GPU the same way that the CPU and other I/O devices are virtualized. This allows multiple virtual machines to share a single GPU. This scales better with respect to the number of concurrent users, but can suffer from performance issues because each GPU access from the virtual machine has to pass through the virtualization layer (hypervisor).

Although GPU passthrough capability is supported by GPUs from mutliple vendors, vGPU is only available on specific NVIDIA cards [Herrera, 2013]. Therefore, supporting GPU-based applications in the cloud requires a combination of specific GPUs, virtualization software technologies, and cloud management software. For example, building a private cloud with vGPU support out of open source software can currently only be achieved through OpenStack or CloudStack using XenServer as a hypervisor with NVIDIA GRID GPUs. However, this situation is likely to improve due to research being done on GPU-virtualization techniques [Dowty and Sugerman, 2009; Gupta et al., 2009; Giunta et al., 2010; Yang et al., 2012; Tian et al., 2014; Vu et al., 2014] and the demand for cloud gaming services [Shea et al., 2013; Huang et al., 2013].

Interactivity is crucial for interactive visualization. This implies that cloud-based visualization services with client-side rendering is an ideal combination in order to avoid round-trip network latency delays.

Current techniques, especially cloud-based gaming services use cloud-based rendering with video streaming to the client [OTOY, 2013; NVIDIA, 2015]. Despite the optimization attempts including use of content delivery networks located near the geographic locations of the users, latency still remains a challenge [Choy et al., 2012; Chen et al., 2011; Suselbeck et al., 2009]. Therefore, cloud-based rendering approaches might not be an appropriate approach for interactive visualizations. Moreover, current advances in mobile computing devices providing GPU-based rendering capabilities is narrowing the performance gap between these devices and desktop computers making client side rendering an attractive approach. However, remote rendering approaches may be appropriate in cases where the decision is driven by other factors other than computational resources, for example privacy or security issues.

3.4.3 Design Considerations for Cloud-based Visualization Service

Visualization software as currently designed and developed is typically not capable of exploiting the existing distributed heterogeneous hardware infrastructure and has limited support for typical research environments, which consists of researchers who are distributed across the globe and need tools that can support their collaborative work. The problem facing visualization tools is how to best harness the heterogeneous computing infrastructure, interoperability with other tools that form part of the entire data analysis pipeline, like simulation and analysis/analytics tools.

Depending on the target users (i.e., domain scientists or developers of visualization tools), different approaches for partitioning the visualization pipeline can be adopted. Given its flexibility, a hybrid approach that combines different stages of the visualization pipeline as individual web service and at the same time minimizing communication costs is more ideal. This hybrid approach is more appropriate for developers of visualization tools who would be interested in combining different visualization services in order to create new visualizations.

For example, a cloud-based visualization service could be considered as a composite service formed from three individual services, namely Data Service, Mapping Service, and View Service. The data service would be responsible for interfacing with various data sources and hides the heterogeneity of the various data sources to the visualization pipeline. Other services forming the visualization pipeline can access the data through a uniform interface exposed by this service. Data filtering or search capabilities would also be provided by this service. These two capabilities are crucial especially for big data that cannot be visualized all at once. Users need to be provided with capabilities to

selectively visualize the data for the task at hand. The mapping service would be responsible for mapping the data into renderable representations or visual representation. The visual representation can then be passed to the viewing service for rendering and interaction with the user.

Providing rendering as a capability of a view service rather than a service in its own would be appropriate in order to hide low-level rendering details from users who will be responsible for combining these services into new composite visualization services. All details dealing with rendering and camera manipulations become hidden under a specific instance of a view service. That is, the implementation details of the renderer and camera are hidden within the view service. Only simple interfaces to facilitate parameter manipulation and interaction can be provided. This abstraction can serve well when the underlying implementation technologies for these components change but the user interface should remain unchanged.

Communication between these individual services would be coordinated by a controller service that can integrate these services dynamically and flexibly at run-time to create a single composite visualization service. Other tools that form part of a data analysis workflow can then connect to the service as clients in order to access its functionality. The interaction between the service and its clients can be based on any asynchronous message based communication in order to decouple them in space and time. This decoupling allows them to run on different platforms with computational resources that are more suited for their tasks and hence improve their performance.

By exposing the visualization functionality as a service, it becomes a long running process that once started does not stop, unless intentionally stopped. With this operation mode, it can support a reactive execution model or "push" model rather than a visualization application “pulling” the data to be visualized. Data sources would be “pushing” data to the visualization service. This model of execution would also allow the data sources and the visualization to remain connected in such a way that when the data changes, the visualization can change dynamically to reflect changes in the data. As data continue to grow (Big Data Challenge), this event-driven approach may prove to be more suitable, since data can be visualized as it comes in (as data streams).

With this long running nature of the service, efficient use of resources becomes paramount and this is where the *elasticity* of the cloud in resource management become crucial for scalability. Elasticity is a capability of the cloud to provision computational resources depending on the current workload of the system [Herbst et al., 2013]. It allows the cloud to automatically allocate more resources to a service when the demand is high and releases already allocated resources when the demand becomes low.

3.4.4 Resource and Data Management in the Cloud

Efficient use of resources in the cloud is receiving more research attention focusing on more efficient resource abstraction mechanisms due to high resource demands and inefficiencies in virtual machines as units of computational resource management. Since each virtual machine requires a complete operating system, the I/O overhead and memory requirements for the service running in a virtual machine can be high [Felter et al., 2015]. Current trends towards containers (e.g., Docker Container [Merkel, 2014]), and cloud-based resource management platforms, similar to operating systems, tuned for cloud infrastructure (e.g., Apache Mesos [Hindman et al., 2011]) are more promising for better performance and efficient sharing of computational resources. Both Docker Container and Apache Mesos have GPU support (mainly NVIDIA GPUs).

Another important aspect of cloud-based visualization is the issue of data management. Many existing visualization applications perform data management tasks on their own without the help of databases. However, this approach may not be efficient in the cloud because of the massive amount of data and the potential concurrent access of the visualization services. For example, some researchers are already using cloud computing resources to run their simulations [Kohlhoff et al., 2013; Vecchiola et al., 2009; Evangelinos and Hill, 2008]. These simulations typically produce massive data that may not feasible to be visualized using a single workstation. Since the data, in this case, is already in the cloud, it becomes desirable to also host the visualization services in the cloud in order to avoid moving these massive data. In this setup, visualization applications would benefit from delegating concurrency issues and data management to a separate data layer that is dedicated for data management.

Concurrent access and large data management has, for a long time, been a central issue in database research, however the adoption of databases for management of scientific data in general and visualization in particular has been slow. Complexity of data management and lack of native support for scientific data in most of the databases has been cited as one reason for this slow adoption [Gray et al., 2005]. SciDB [Stonebraker et al., 2013] is a recent database for scientific data that has been introduced to address these issues. Moreover, as the management of computing infrastructure including databases is being outsourced to third parties through computing clouds, the complexity of managing this infrastructure becomes less of a problem for scientists. This is another compelling reason for cloud-visualization services to relieve scientists from managing visualization tools with all the complexity that comes along, and let them focus on doing science. This would not only accelerate discovery but also encourage collaboration.

3.5 Interactive Web-based Visualization

High performance rendering is crucial for interactive visualization in browsers which, in most cases, is achieved through GPU-based rendering. Although GPU-based rendering has been common in the desktop platforms, it has only been recently available in the browser through the introduction of WebGL.

GPUs have mostly been optimized for triangle rendering, meaning that any higher description of a model has to be tessellated to triangles before rendering. Tessellation is normally done in the CPU and then triangles are sent to the GPU for rendering. However, for large models, this usually results in a performance penalty because of the CPU-GPU bandwidth bottleneck. Modern graphics APIs provide more programmable stages (e.g., tessellation shaders) that allow this to be done in the GPU and therefore avoid the bandwidth bottleneck. However, in WebGL only vertex and fragment shaders are available. Any solution that requires tessellation has to be done in JavaScript on the CPU. For large data sets, this becomes an impediment for high performance. It becomes even more problematic for dynamic data that can potentially change in every frame.

Rendering geometry data in WebGL without costly tessellation in the CPU is possible for objects whose geometry can be implicitly defined, e.g., spheres. This geometry can be generated by performing ray-object intersections in the fragment shader. The application just passes the parameters of the implicit function to the GPU, thus minimizing data uploaded to the GPU. For example, in molecular rendering, where spheres are used to model each atom in the molecule, only the radius and the center is sent to the GPU together with other attributes like color.

Therefore, for high performance rendering in the browser, generating the geometry in the GPU using this ray casting technique offers performance benefits compared to polygon rendering. Ray casting offers not only performance benefits but also generates high quality pixel-accurate images. Since ray-object intersections are the most expensive calculations, acceleration structures are usually used to minimize the number of intersection tests. The acceleration structures are usually created on the server as part of a preprocessing step as creating them on the client would be slow [Mwalongo et al., 2014]. This is also explained in details in chapter 4.

Although there have been great improvements in JavaScript performance through just-in-time (JIT) compilation techniques and limited parallel processing capabilities through web workers, as discussed earlier in chapter 2, the performance of JavaScript still lags behind that of native compiled languages. Therefore, in order to gain maximum performance, most web applications follow a client-server architecture where a computation is partitioned between

server and client. In this setup, the server can be used for data storage and high demanding computations that cannot be performed in the client. This means that the data to be visualized has to be transferred to the client before rendering. Network data transfer has to deal with two main challenges: bandwidth and latency. Despite great improvements in network infrastructure, bandwidth is still a challenge due to huge amounts of data generated by modern simulations, social networks, sensors, and scientific instruments.

As the growth of data increasingly outpace the improvements in network bandwidth, compression techniques and data filtering techniques are important for efficient transfer of data from servers to clients. Compression and filtering can also be combined to obtain optimal results. An important note about compression techniques is to balance between optimizing for compression ratio and decoding time. Decoding time is important for interactive rendering. When complex compression techniques that incur expensive decoding procedures are used, the gains obtained in bandwidth savings are lost. These various techniques are discussed in section 3.5.1.

Apart from compression techniques, selectively sending minimum data that is just enough to generate the required visualization is another approach for reducing the amount transferred to the client. This minimal data is usually encoded in a GPU-friendly format (e.g., using array buffers) such that the data can be directly uploaded to the GPU with very little processing on the client.

Latency is another important factor for interactive rendering. Network latency can be influenced by the distance between the client and server locations. As the distance between server and client increases, so does the propagation delay. Expensive computations at the server and client due to, for example, compression and decompression can also increase latency and, thus, affect rendering performance.

Various techniques are employed to tackle this challenge through decoupling the rendering thread and the data transfer thread so that the rendering thread does not get stalled by latencies due to network transfers. In WebGL, this means performing rendering in the main thread and data fetching in the web worker and using transferable objects to transfer the data between the two to avoid the cost of copying the data. The communication between the main thread and the worker thread is asynchronous so the main thread can continue to render the already received data and to update the image as the new data becomes available.

Another techniques that is used to deal with latency is through data streaming and level of detail techniques. The data is encoded in the format that allows progressive transfer of different levels of detail starting with a low resolution model

and continuously updating the model until the high resolution model is downloaded. This technique is mainly applicable for data that can be represented as meshes (see section 3.5.1 for details).

Therefore, interactive web-based visualization depends on optimizing for interactive rendering. This requires techniques that minimize network and CPU-GPU bandwidth together with minimizing latency. For the case of bandwidth this can be achieved through compression techniques that minimize decoding time before rendering. Latency hiding techniques like overlapping rendering with I/O tasks, streaming and progressive rendering techniques, and use of GPU-friendly structures (e.g., array buffers) are crucial in minimizing latency and improving rendering times.

As visualization is executed as a pipeline with several stages, prioritized optimization for rendering is important because no matter how optimized the other stages are, if they in turn have a negative impact on rendering time (e.g., through long decoding time, or CPU-based decoding leading to CPU-GPU bandwidth bottlenecks), the entire pipeline will be as fast as the rendering stage can be. An end-to-end performance optimization strategy is important in order to achieve satisfactory performance. The sensitivity of latency on interactive rendering is what makes client-side rendering important compared to server-side approaches. Although server-side rendering can benefit from powerful computational resources on the server or cloud, they can only be appropriate when interactivity is not important.

3.5.1 Data Encoding and Transfer Techniques

Data encoding and data transfer on the web aims at efficient use of bandwidth, progressive transmission, and minimal decoding time at the client side. These goals are important to ensure high performance rendering in the browser.

Efficient use of bandwidth can be achieved through compression techniques or minimizing the transferred data by preprocessing the data at the server and by avoiding sending unnecessary data to the client [Ponchio and Dellepiane, 2015; Wen et al., 2014]. Commonly used compression schemes that can be automatically handled by browsers are *gzip* and *deflate*. To use these schemes, the browser sends the request to the server with a header *Accept-Econding: gzip,deflate*, indicating to the server that it can handle any of these formats. If the server can encode the data in one of these formats, it sends back the data with the respective encoding. If none of the requested formats are supported by the server, an uncompressed format is sent instead.

These compressions can be combined with preprocessing approaches that select minimal data to be sent in order to get optimal data file sizes that use minimal

bandwidth. The advantage of using these standard compression schemes is that decoding at the client is handled automatically by the browser and, therefore, there is no decoding step in the JavaScript application.

Since the browser provides the uncompressed version of the data to the JavaScript application, the data is uploaded to the GPU in uncompressed form, and therefore it can still suffer from the CPU-GPU bandwidth bottleneck. Moreover, for large dynamic data uploading to the GPU can take significant time and therefore slow down rendering times. Compression techniques that cover the entire pipeline up to the GPU are more efficient because they minimize the data uploaded to the GPU and also save GPU memory. This is important especially for mobile devices.

Since data for rendering is usually fetched from the network, it is important to isolate the rendering thread from network latencies. Latency cannot be completely eliminated but can be tolerated if hidden. This is commonly achieved by doing rendering in the main thread and data fetching in a web worker and using asynchronous communication between the two threads. Rendering can only be done in the main thread because the current WebGL standard does not allow accessing WebGL context in the web worker. To further avoid the cost of copying data between the main thread and the web worker, an efficient transfer mechanism called *transferable objects* is used. This mechanism transfers the ownership of data from sender to receiver.

Another approach for dealing with latency is using progressive data transmission and level of detail (LOD). This approach is common for transmission of mesh data. The idea is to encode the mesh data in different LODs and transmit each of them progressively to the client starting with a low resolution model. At the client, the low resolution model is decoded and rendered. The user can start interacting with the model while more refined representations are progressively being loaded and the image updated to get a more refined representation. Several LODs can be used including discrete, continuous, and view-dependent LODs.

Other techniques combine progressive transmission, view-dependent LODs and compression to optimize the network bandwidth usage [Ponchio and Dellepiane, 2015]. However, they require decoding at the client side that can affect rendering performance and CPU-GPU bandwidth. There are techniques that avoid the decoding step by using chunked binary data representations that can be interpreted at the client as Array Buffers and directly uploaded to the GPU. These approaches can be combined with *quantization techniques* to compress mesh attributes data like position, normals and color to reduce the amount of transferred data.

Dynamic volumetric data can be encoded in the form of compressed textures

combined with a normal compression scheme, for example, deflate [Yang et al., 2015]. Since deflate scheme can be handled by the browser, at the client side the application is presented with already inflated data (i.e., only the compressed texture format). Each frame is treated as compressed video texture that is uploaded directly to the GPU for rendering. This technique takes advantage of video texture support of WebGL and the compressed texture minimizes the bandwidth and GPU storage.

There are other techniques that use lossless image compression techniques like PNG to encode data. However this approach saves only the network bandwidth because the decompression is handled by the browser before presenting the data to the application. Therefore it is uploaded to the GPU in an uncompressed form.

Progressive transmission leverages a feature of the HTTP protocol called *range requests* that allows the client to request a range of bytes of a resource. With this capability encoding formats can pack various LODs in each chunk which then can be progressively decoded at the client [Limper et al., 2014].

As decoding time is paramount for interactive rendering in the browser, any compression technique that does not take that into consideration may be just shifting the problem from a network bandwidth problem to a latency problem. Moreover, doing decoding in JavaScript, no matter how fast it is, can still suffer from the cost of data upload to the GPU and the limited CPU-GPU bandwidth. Although the bandwidth of the PCIe bus is relatively high compared to the network bandwidth, data upload to the GPU can still remain challenging especially on less powerful devices. For static data, a one time upload penalty can be tolerable, but for dynamic data, its cost is significant. Therefore the cost of data upload to the GPU for large dynamic data can not be ignored.

Interactive web-based visualization, therefore, requires a careful balance between computational, communication (bandwidth and latency), and storage costs (memory). While these costs are somehow common also in other visualization environments, communication costs are more pronounced in remote visualization due to network-based communication and data access. Optimizing only for one type of cost may not be optimal for the entire pipeline performance and therefore end-to-end considerations are more important. Data encoding and transfer techniques that use progressive transmission, levels of detail, and GPU-friendly formats that eliminate or keep client side processing to the bare minimum before GPU upload are more promising to achieve high interactive rendering in the browser.

3.5.2 Local Rendering in the Browser

In the local rendering approach, the rendering stage of the visualization pipeline is performed at the client-side (i.e., in the browser). The main motivation for local rendering is to improve interactivity during visualization. Local rendering avoids round trip latencies suffered by server-side rendering approaches. Recent advances in web technologies including JavaScript improved performance and WebGL have made it feasible to achieve high rendering performance in the browser. The discussion in this section focuses only on techniques that utilize modern HTML5 and WebGL standards, as these are the technologies that allow high performance rendering in the browser without plugins.

Many visualization techniques in the browser exploit the capability of the GPU through WebGL or libraries built on top of it (e.g., SpiderGL [Di Benedetto et al., 2010], Three.js [Threejs, 2010], and Babylon.js [BabylonJS, 2013]). Some other techniques use the 2D canvas API or the Scalable Vector Graphics (SVG) through the <*svg*> HML5 element (e.g., D3 [Bostock et al., 2011]). To improve performance, other capabilities like Web Workers (for background processing), Array Buffers (for binary data processing) are used.

The main advantage of local rendering approach is that once the data has been transferred to the client, rendering performance is not affected by network bandwidth and latency issues. For dynamic data, the rendering step can be decoupled from data fetching and pre-processing steps (e.g., through asynchronous communication) to ensure interactivity. By isolating the rendering step from network transfers and pre-processing steps, the user can continue interacting with the data which has already been uploaded to the GPU without being stalled by network data transfers. The main challenge for this approach is that it requires considerably high computing resources on the client (e.g GPU), though this is becoming less a challenge with current technological advances.

Common GPU-based rendering approaches in the browser use polygonal-based or GPU-based ray casting techniques. Polygonal-based techniques use basic triangle rasterization by using the low-level WebGL API directly or any JavaScript library built on top of it. This approach is mainly used for rendering 3D models whose geometry is represented as triangle meshes, for example, CAD models or geometry produced from modeling tools like Maya or Blender.

In some cases where the original data is not a triangle mesh, the model has to be tessellated first before rendering. For example, in visualization of molecular structures, spheres are used to model atoms of a molecule. Therefore, these spheres have to be triangulated first and then uploaded as triangles to the GPU for rendering. Use of triangles as approximations for surfaces benefits from highly optimized rasterization hardware to achieve high rendering rates. In

order to get smooth surfaces, however, a large number of triangles is usually required. This can in turn lead to huge demands on CPU-GPU bandwidth.

In cases where the meshes are fetched from the network, this can have implications on network bandwidth, too. As discussed in section 3.5.1, different compression and transfer techniques are employed to address these challenges. In an effort to achieve interactivity, sometimes a trade-off has to be made between rendering performance and image quality. A surface can be approximated using fewer triangles in order to maintain interactivity, thus compromising the quality of the final image. Sometimes LODs techniques can be used to selectively represent a scene with a coarse or a detailed model depending on how far the object is from the view point or other viewing parameters (view-dependent LOD).

A different rendering technique that achieves high rendering rates and high image quality is GPU-based ray casting. It is an image-based rendering technique that exploits the power of the GPU to accelerate rendering. Since WebGL supports only vertex and fragment processing as programmable stages, GPU-based ray casting is performed in a fragment shader. Images are generated by shooting rays through the center of each pixel to the scene to find objects that intersect the ray using ray-intersection tests. When a ray hits an object along the ray, the closest hit point is taken and shaded according to any shading model, for example, Blinn-Phong. Since rays are shot for each pixel, the resulting images are pixel correct and high quality. To initiate the rendering pipeline, a bounding geometry or proxy geometry is rendered. Other implementations require an acceleration structure and efficient ray traversal techniques to accelerate the ray-object intersection tests as they consume more computation time than other steps of the algorithm.

In platforms where OpenGL or WebGL supports depth writes in the fragment shader, a different implementation technique can be used. In this technique, a simpler proxy geometry is rendered in order to generate the fragments and ray-object intersection is performed in the fragment shader to generate the real object. This approach yields better performance as it does not require creation of an acceleration data structure. Therefore, the time for creating the data structure and the cost of ray traversal in the fragment shader is avoided. This is especially important for dynamic data. These techniques are discussed in more detail in chapter 4 and chapter 5 in the context of macromolecular structure visualization.

Visualization of volume data from medical and other domains uses GPU-based volume ray casting. This technique can use a single-pass [Movania and Feng, 2012] or multi-pass implementation [Congote et al., 2011] depending on the capabilities of the hardware. The technique requires support of 3D textures to

store the volume data. For hardware that does not support 3D textures, 2D slicing or 2D texture atlases can be used to represent the 3D texture. However, this approach requires that trilinear interpolation is computed in the fragment shader which may be inefficient. Using 3D textures benefits from trilinear interpolation performed automatically by the graphics hardware.

3.6 Web-based Visualization Applications

Many scientific domains have employed web-based visualization approaches in order to exploit the ubiquity of the browser and modern HTML5 technologies. These domains include the visualization of particles, volumes, hierarchical and relational data, geospatial data, and 3D models. Moreover, generic toolkits for visualization of various datasets have been developed.

Particle visualization has mainly focused on visualization of molecular data. Molecular visualization in the browser has been popular from the early beginnings of the web. The visualization of molecular data as calotte model is an example for particle-based visualization where each atom is depicted as a sphere. The size of each sphere represents the van-der-Waals radius of the corresponding element. When using polygon-based rendering, these spheres have to be tessellated into triangles that are then rasterized.

Jmol [Jmol, 2009], implemented as a Java applet and its recent re-implementation in JavaScript, JSMol [JSmol, 2013], are some of the most popular tools used for visualizing protein structures. The main limitation of Jmol is that it requires users to install a browser plugin. While JSmol alleviates the plugin problem, it still lacks the capability of handling large molecules. Both Jmol and JSMol use polygon rendering as discussed section 3.5.2. Callieri et al. [2010] also use polygon-based rendering for the visualization of molecular data via the SpiderGL library [Di Benedetto et al., 2010]. Additional recent examples of web-based molecular viewers are NGL Viewer [Rose and Hildebrand, 2015], 3DMol.js [Rego and Koes, 2015], and PV [Biasin, 2013]. They all leverage WebGL for visualization of molecular structures in the browser. The RCSB protein data bank [Berman et al., 2000] web site, for example, uses NGL as one of its integrated protein viewers. Figure 3.3 shows one of the visualizations produced by the NGL Viewer. iView [Li et al., 2014] is another protein viewer that also incorporates virtual reality features. Li et al. [2014] presented a visualization tool for protein-ligand complexes that was built using Three.js [Threejs, 2010]. Beside the aforementioned calotte model, most of these viewers support other commonly used visualizations for molecular data (e.g., molecular surfaces or ball-and-stick models), which are derived from the particle positions. These additional representations are all the same based on a polygonal representation

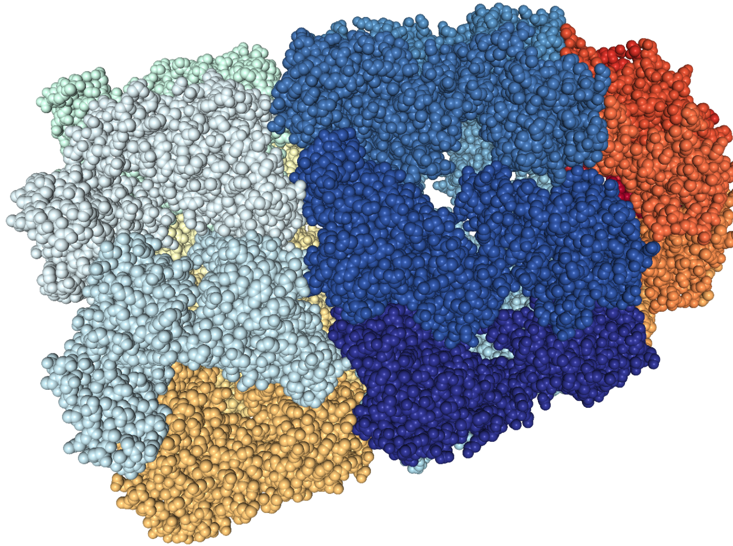


Figure 3.3 — Interactive visualization of the chaperonin complex (PDB ID: 1AON, 58,674 atoms) using the integrated NGL protein viewer [Rose and Hildebrand, 2015] on the RCSB webpage (`https://www.rcsb.org`).

and rendered using the same techniques.

Visualization of dynamic molecular data (*trajectories*) from molecular simulations is also an important feature for domain scientists. This capability has up to recent years been confined to desktop solutions. Chapter 5 discusses visualization of dynamic data in more details. As discussed in section 3.5.2, two main approaches can be used for rendering spheres: GPU-based ray casting and mesh rendering. GPU-based ray casting techniques offer better results in terms of rendering performance and image quality. These techniques are discussed in more details in chapter 4 and chapter 5. Figure 3.4 shows some exemplary images produced by GPU-based ray casting.

Chandler et al. [2015] present a WebGL remote visualization solution for smoothed particle hydrodynamics (SPH) simulation data. The server side preprocesses data by creating an octree data structure for storing a multi-resolution model for each time step. Streaming of data to the client is done for each individual time step. The client initially receives a low-resolution model from the server and then high resolution models are incrementally loaded as the rendering progresses. This is done to avoid long latencies before the first

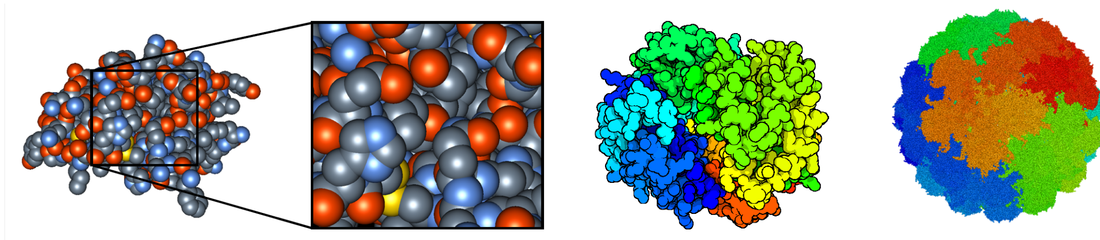


Figure 3.4 — Visualization of molecular structures at interactive frame rates using WebGL-based ray casting techniques. Left: an insulin protein (PDB ID: 1RWE, 823 atoms) and its close-up view with correct sphere-sphere intersections and pixel-accurate spheres [Mwalongo et al., 2014]. Middle: a hemoglobin protein (PDB ID: 4HHB, 4,384 atoms) with toon shading effects generated through deferred shading [Mwalongo et al., 2015]. Right: capsid of a papillomavirus (PDB ID: 3IYJ, 1.3 M atoms).

view of the model is rendered. The received octree data structure is encoded as a texture and uploaded to the GPU during rendering. Since the solution uses a WebGL 1.0 implementation that does not support 3D textures, the textures are packaged as 2D textures and address translation is used to access the data in the shaders. Moreover, fetching and processing of data in the client is done in a web worker to avoid stalling the rendering or UI interactions in the main thread.

Volume visualization has been applied to medical, weather, and hydrodynamics simulation data. Rendering techniques used in these applications are GPU-based ray casting or volume ray marching and texture-based techniques. These techniques typically require 3D texture support in the hardware.

Congote et al. [2011] presented an early work on volume visualization using GPU-based ray marching in WebGL (see chapter 2 for details). They use a multi-pass GPU-based volume ray casting approach to visualize volumetric data from medical imaging and weather radar scans. Their work forms a basis for the implementation of MedX3D in X3DOM—MEDX3DOM [Congote, 2012]. MedX3D [John et al., 2007] is an extension of the X3D standard Brutzman and Daly [2007] to support web-based volume visualization. Other volume visualization systems for large medical data were introduced by Movania and Feng [2012] (see figure 3.5) and Noguera and Jiménez [2012]. Both systems also run efficiently on mobile device browsers. Jiménez et al. [2014] presented a client-server application for the analysis of the 3D fractal dimension of MRI data.

Jacinto et al. [2012] presented a client-server application for medical volume segmentation and rendering. The segmentation uses an image-based approach

Figure 3.5 — Example of medical volume rendering in WebGL using volume ray marching (image: [Movania and Feng, 2012] © 2012 IEEE).

that extracts slices from the original volume data on the server and sends them to the client as JPEG or PNG images, where they can be annotated to trigger the segmentation on the server. Isosurfaces are extracted from the segmented volume and sent to the client for visualization. The server side uses the VTK library and the client side uses the Three.js [Threejs, 2010] library for rendering the isosurface meshes (see figure 3.6). One targeted application is the fast extraction of knee bones for prosthetic design.

A medical visualization application that exploits augmented reality capabilities has been shown by Virag et al. [2014]. The system is designed using a client-server architecture, but also takes advantage of WebRTC for accessing a camera and transmitting the imagery to remotely allow for a second opinion from another physician. The camera feed is also used as input for the JSARToolKit, which takes care of marker tracking. That way, a virtual model of a patient equipped with said markers can be synchronized with the visualization.

Hou et al. [2015] present an image-based technique to visualize medical volumes. They used a server-side GPU-accelerated rendering and stream each generated frame to the client for display. The rendering engine on the server uses CUDA and the VTK library for rendering the data to a framebuffer object (FBO) that is then read back and sent to the client. The advantage of this approach is that it benefits from powerful computational and storage resources on the server side.

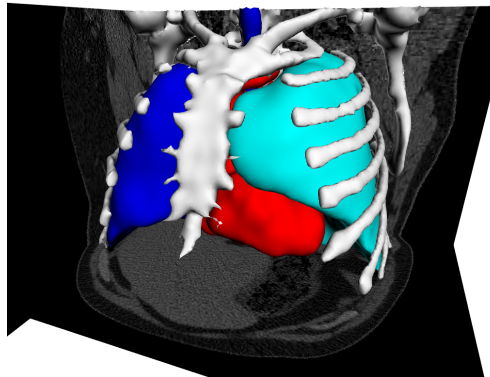


Figure 3.6 — Example of the web-based medical volume rendering presented by Jacinto et al. Jacinto et al. [2012]. The isosurface meshes were extracted on the server using Marching Cubes and rendered on the client using WebGL (image: [Jacinto et al., 2012] © 2012 ACM).

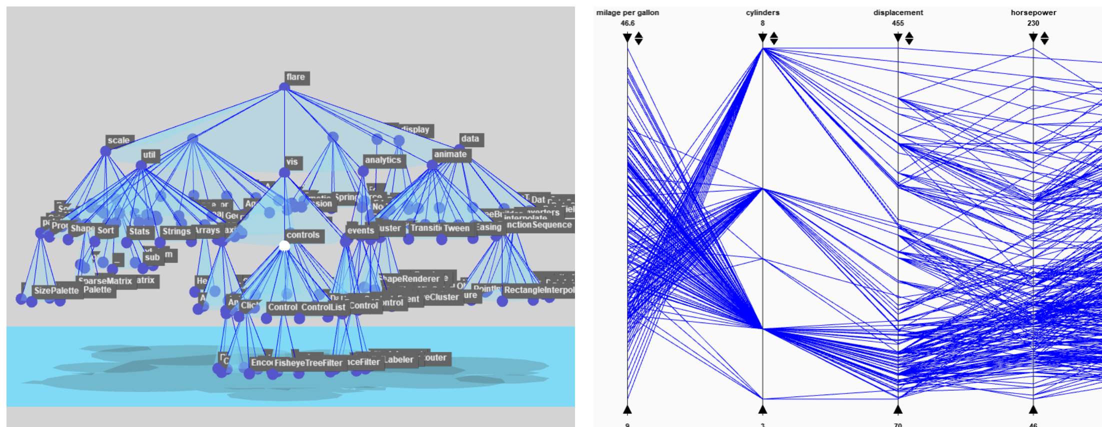


Figure 3.7 — Examples of web-based information visualization using FluidDiagrams [Andrews and Wright, 2014].(image created from examples on the author's homepage `http://digitalwright.net/msc/examples/`, last accessed 2018/07/31).

However, by doing the rendering on the server, smooth interaction on the client is affected due to round trip network latency.

Information visualization techniques leveraging the power of WebGL and HTML5 are presented by Sarikaya and Gleicher [2015] and Liu et al. [2013]. Figure 3.7 shows example information visualizations.

Sarikaya and Gleicher [2015] present a WebGL-based implementation of the Splatterplots technique [Mayorga and Gleicher, 2013] for interactive visualization of large two-dimensional point data sets. Splatterplots address the problem of overdraw in regular scatter plots that arise when visualizing large point data

sets. Computations that were performed on the CPU in the original implementation [Mayorga and Gleicher, 2013] were moved to GPU using *render-to-texture* techniques in order to attain better performance.

Liu et al. [2013] present techniques for visualization of geolocated data in WebGL. They use a client-server architecture where the server preprocesses the data before streaming them to the client for rendering. The preprocessing involves pre-computing data tiles from data cubes and then encoding them as image files. These images are transferred to the client and uploaded to the GPU as textures. On the client side, the authors employ a two-pass approach for rendering: The first pass is a computation step which uploads the data tiles to the GPU and computes summary values. The results are written to a framebuffer object (FBO). The second pass is a rendering pass that uses the textures generated in the first pass to render the final images. The focus of their techniques is to achieve *interactive and perceptual scalability* for visualization of large data sets. In order to achieve this scalability, they use binned aggregation techniques to reduce the amount of data to be visualized and exploit GPU acceleration for parallel computation and rendering. Data reduction techniques combined with the use of data tiles allow efficient visual analysis of large data sets.

Geospatial Visualization includes spatial and spatial-temporal data (4D) and covers various kinds of data like marine data [Resch et al., 2014], raster-maps [Jenny et al., 2016], weather forecasting data [Diehl et al., 2015], and city models [Gaillard et al., 2015]. Some examples of geospatial visualizations are shown in Figure 3.8

Common among all modern web-based geospatial visualizations is the use of WebGL for hardware-accelerated rendering and other modern HTML5 technologies. Some approaches [Resch et al., 2014; Gaillard et al., 2015] use WebGL directly and others [McCann et al., 2014; Kim et al., 2015] use declarative frameworks like X3DOM. X3DOM implements a component from the X3D standard for visualization of geospatial data [Plesch and McCann, 2015].

The overview-and-detail visualization for mesh data by Figueiredo et al. [2014] described in section 3.5.2 was used for the visualization of geomorphological structures of an underground cave in the browser. A 3D mesh representing the surface of the cave was generated from a detailed point cloud and stored on a server at different levels of detail. The overview can be used to navigate and select stalactites or stalagmites and to request a high-resolution representation of the filtered neighborhood of these relevant structures for detail analysis.

Resch et al. [2014] describe a prototypical implementation of a web application for visualization of 3D time-dependent geospatial data to support non-experts

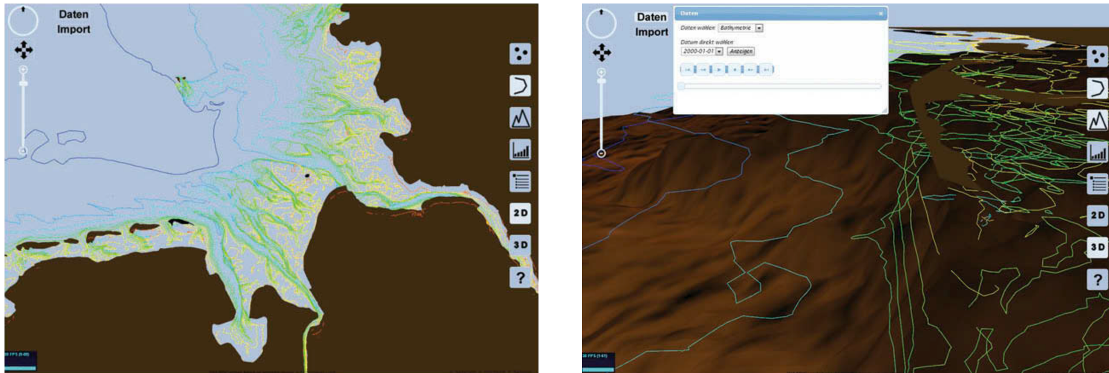


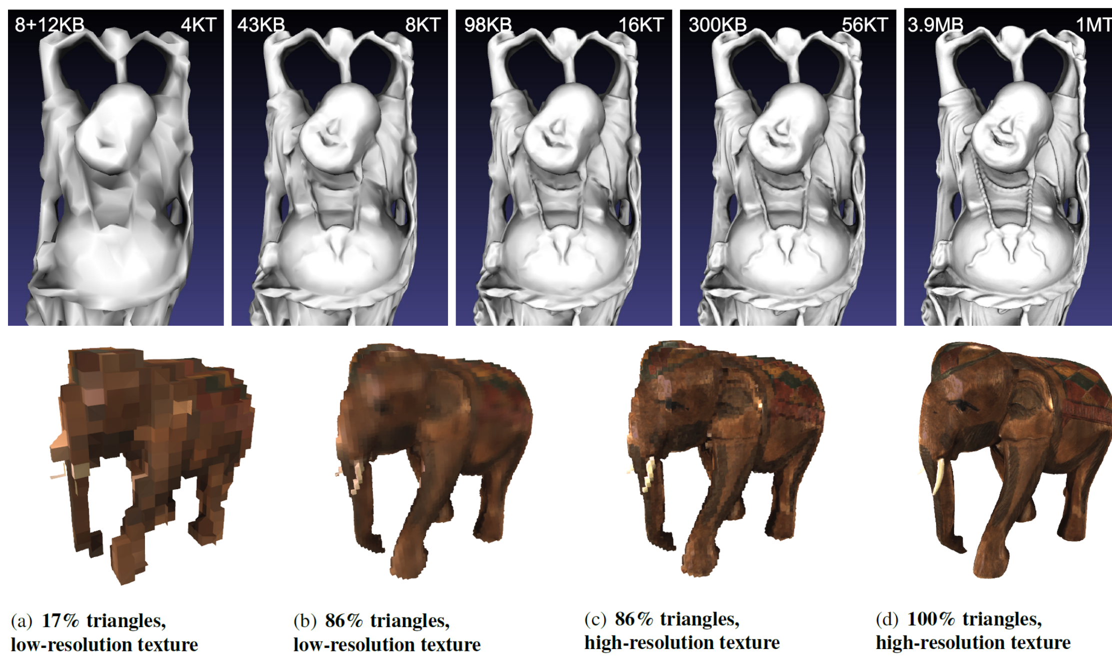
Figure 3.8 — Examples of web-based geospatial visualization applications (image: [Resch et al., 2014]).

in decision making (see figure 3.8). They further discuss various user requirements including usability, representing 4D geo-data for such applications, their implementation aspects and performance. Moreover, the authors highlight challenges of representing spatial-temporal data in web-based visualization tools.

3D Models Visualization involves rendering of 3D meshes that can either be stored locally or streamed from a server. As discussed in section 3.5.1, these applications, employ various level of detail techniques combined with efficient data encoding and transfer techniques for efficient bandwidth usage. Additionally, these techniques can be combined with compression or selective data transfer to further reduce the amount of data that is sent to the client for rendering. Examples of various 3D model visualizations are shown in Figure 3.9

Generic Toolkits exist, covering a range of application domains including information visualization, protein visualization, medical visualization, and geospatial visualization. Most of these toolkits use client-side rendering that employs modern browser-based web technologies like WebGL. They provide high-level functionality on top of WebGL in order to simplify creation of visualizations in certain domains.

FluidDiagrams [Andrews and Wright, 2014] is a prototypical toolkit that offers GPU-accelerated implementations of various information visualization techniques (e.g., bar charts, scatter plots, line charts, hyperbolic browser, parallel coordinates, and cone trees) using the Three.js [Threejs, 2010] library (see figure 3.7). The authors demonstrate that their WebGL-based implementation is capable of scaling to large data sets and achieves higher frame rates compared to those based on SVG (e.g., D3 [Bostock et al., 2011]) or the 2D Canvas API [W3C, 2015].



(a) **17% triangles, low-resolution texture** (b) **86% triangles, low-resolution texture** (c) **86% triangles, high-resolution texture** (d) **100% triangles, high-resolution texture**

Figure 3.9 — Examples of web-based 3D model visualizations (image: Top: [Ponchio and Dellepiane, 2015] © 2015 held by Owner/Author(s). Publication Rights Licensed to ACM. Bottom: [Limper et al., 2014] © 2014 ACM).

Goecks et al. [2013] discuss various server and client web-framework components for simplifying the creation of visual analysis applications for high-throughput genomic data. They further describe how the components are used to create several concrete applications. The components are implemented and integrated into the Galaxy web platform [Goecks et al., 2010].

ManyEyes [Viegas et al., 2007] was a public information visualization website that allows users to upload their data, visualize them, and finally annotate the visualizations while sharing them with others.

Popular commercial cloud-based information visualization solutions include Tableau Online [Tableau, 2013] and TIBCO Spotfire Cloud [TIBCO, 2014]. These solutions provide Software as a Service (SaaS) tools for creating various interactive visualizations using local or online data and allow sharing the visualizations amongst users. These commercial solutions are usually designed to scale to very large data sets and many concurrent users.

The X Toolkit [Haehn et al., 2014b] is a WebGL-based framework for interactive visualization of medical data sets in the browser. The toolkit supports various data formats and visualizations from the neuroimaging domain. Additionally, it supports various data formats for surfaces and volumes. Using this toolkit, Haehn et al. [Haehn et al., 2014a] developed a web-based application for collab-

orative proofreading of brain data that supports 2D and 3D visualizations.

Cesium.js [Analytical Graphics, 2012] is a general library for geospatial data visualization that relies on WebGL for fast rendering. It supports various standard data formats as well as the glTF format [Khronos, 2015] for the exchange of 3D data between client and server.

Jomier et al. [2011] discuss the integration of ParaviewWeb [Jourdain et al., 2010] with a medical and scientific data management system (MIDAS) [Jomier et al., 2009] for visualization of large data sets. The system uses a remote rendering approach where images are rendered on the server and streamed to the client for display. Their integration setup uses a client-server architecture between the data management system and the visualization server. The user selects what data to visualize through a web-based user interface of the data management system. Subsequently, the visualization request is forwarded to the visualization server. Badam and Elmqvist developed Polychrome [Badam and Elmqvist, 2014], an application framework that allows synchronized interaction with web-based visualization. This mainly focuses on the interaction side of visualization and can be used for online collaboration as well as UI synchronization across devices or remote control. Interaction mode extensions to X3D/X3DOM concerning navigation and manipulation for web-based CAD applications are also described by Mouton et al. [2014].

3.7 Conclusion and Challenges

This chapter has provided an overview of previous work on the area of remote visualization focusing on web-based, grid-based, and cloud-based visualization, and web services. Since computing resources employed in remote visualization are distributed and heterogeneous, research on this topic has also attempted to apply the concept of web services to visualization in these computing environments. The chapter has discussed both early approaches using SOAP-based web services and the recent trend towards RESTful web services. Efficient data encoding and transfer techniques for interactive local rendering in the browser have also been discussed. Finally, the chapter has discussed visualization applications in various domains including volume, geospatial, particle, and information visualization.

3.7.1 Conclusion

Motivated by rapidly growing data sets from simulations, sensors, and other digital information sources, remote visualization is gaining a renewed interest

and strives to leverage powerful computation resources to enable scientists and engineers to make sense of the massive data sets.

The main challenges for remote visualization are still bandwidth and latency. Of the two main challenges for remote visualization, bandwidth has received the most attention through various compression techniques (e.g., LZ4 [Collet, 2011] and ZFP [Lindstrom, 2014]) and video encoding techniques. Latency has not been adequately addressed. This makes latency reduction and latency-tolerant techniques an important agenda for research in interactive remote visualization.

There has been much improvement on the computational resources available on the client side. It is now common for mobile devices (e.g., smartphones, tablets, and laptops) to be equipped with multi-core CPUs and graphics cards that can handle computational tasks that a few years ago were restricted to desktop computers. As the trend towards collaborative research and mobile researchers accelerates, visualization techniques that harness the power of mobile devices and server-side technologies (e.g., cloud computing) will become more attractive.

Remote visualization with remote rendering and image/video streaming is based on the assumption that the client machines do not have enough computational power to perform rendering at interactive frame rates. However, this is now changing due to improvements on CPU and GPU technology on mobile devices and networking technology. Hybrid visualization approaches, where expensive preprocessing steps are offloaded to the server or cloud and rendering is performed in the client, are more promising because they exploit the entire spectrum of the available computational resources. Moreover, rendering on the client side addresses the problem of network latency, which is crucial for interactive visualization.

As the data sets continue to grow, even if it becomes possible to visualize all the data, the images generated may become hard to comprehend due to limitations of the human visual system [Healey and Enns, 2012; Kastner and Ungerleider, 2000]. Rather than trying to visualize all the data at once, query-based visualization techniques [Sanderson et al., 2012; Gosink et al., 2008; Stockinger et al., 2005] that extract only a subset of the data relevant for the task at hand will gain importance. By selecting only a subset of the data, less bandwidth is used and computation requirements for local rendering are minimized.

Visualization of large data sets may require data management capabilities similar to those available in databases. Building visualization algorithms on top of a data management and analysis platform would spare the visualization algorithms from data management issues and challenges of the heterogeneity

of the data sources. The visualization layer would focus only on mapping and rendering steps of the visualization pipeline. Data access and filtering can be handled by a separate data service layer that hides the differences of data sources and presents a uniform data access interface to the visualization layer. Scientific databases like SciDB [Stonebraker et al., 2013] that combine data management and analysis capabilities can be exploited for remote visualization of large data sets.

RESTful services [Pautasso, 2014] have become a predominant approach for implementing web services due to their simplicity and for effectively leveraging the web infrastructure for scalability and performance. The introduction of WebGL [Khronos, 2011b, 2013] in the web browsers combined with modern web technologies introduced in HTML5 [W3C, 2017] standard have given the browser the capability to become a preferred platform for deployment of interactive graphics applications. Given the ubiquity of browsers across different devices, visualization tools can harness all the computational resources available to support collaborative visualization for research teams that are geographically distributed. By combining mobile, web, and cloud computing technologies, complex problems can be tackled by bringing together experts across the globe to work on a problem and, thus, accelerate scientific discovery.

3.7.2 Challenges

As discussed in section 3.4, grid and cloud computing are the two platforms for scaling the server side infrastructure to support high demanding computations. With current technological trends, cloud computing is likely to become the dominant server side infrastructure for processing and storage of large amounts of data. Therefore, cloud-based visualization would be the dominant form of remote visualization whereby, data storage and computationally demanding preprocessing is done at the server and the rendering is performed in a browser supporting GPU-based rendering.

The main challenge for cloud-based visualization is limited support for GPU access from the virtual machines. Although the rendering part can be shifted to client devices (the majority of which are currently fitted with GPUs), still GPU access in the cloud is important for visualization techniques that depend on technologies like CUDA and OpenCL for more demanding computations (e.g., preprocessing). As discussed in section 3.4.2, a preferred approach for interactive cloud-based visualization is cloud computation with client-side rendering. Interactive cloud-based visualization can benefit by leveraging GPU acceleration in the cloud the same way cloud gaming is benefiting from GPU acceleration (e.g., rendering and video encoding done on the GPU) [OTOY, 2013; Shea et al., 2013; Chen et al., 2011].

Another challenge is that existing visualization tools were not designed for virtualized and elastic computing infrastructures. For these applications to be deployed in the cloud and to take advantage of its elasticity benefits (ability to scale down and up depending on current workload), redesigning them into the so called *Cloud-Native Applications* [Andrikopoulos et al., 2013] would be required. Moreover, visualization tools will need to be easily integrated with other tools (e.g., simulation and analysis tools) to support seamless analysis workflows and relieve researchers from manual integration of different tools. Service-oriented software techniques can also prove helpful in this regard. Different tools running on different platforms and written in different programming languages should be able to communicate and share their data based on standard communication protocols.

Chapter 4

GPU-based Molecular Data Visualization in the Browser

The visualization of molecular data in the browser has attracted researchers since the beginning of the web. The browser provides a portable platform for deploying applications that run across devices from workstations to smartphones.

Researchers in the field of biochemistry, for example, analyze the behavior of proteins. In order for them to understand the functionality of proteins, the visualization of protein structures is important. Most of the tools developed to support this task were implemenented for the desktop platforms [Humphrey et al., 1996]. Efforts to extend these tools to the browser, until recently, has been challenging due to limited rendering capabilities in the browser as discussed in chapter 3.

One of the popular web-based molecular viewer is Jmol [Jmol, 2009]. This viewer is implemented as a Java Applet. Despite its popularity, its performance has been hampered by lack of support in modern browsers. To solve this problem, a re-implementation based purely on JavaScript, called JSMol [JSmol, 2013] has been introduced. However, as mentioned in chapter 3, it can only support visualization of small molecules.

The advent of WebGL has led to research interest in GPU-based molecular visualization tools that exploit the capabilities provided by the modern browsers. Given that the GPU is optimized for rendering triangles, most existing web-based molecular viewers use polygon rendering techniques to visualize molecu-

lar structures [Di Benedetto et al., 2010; Pettit and Marioni, 2013; Li et al., 2014; Rose and Hildebrand, 2015].

One of the models for visualizing molecular structures is the space-fill model. In this model, each atom of the molecule is represented as a sphere where the center of an atom maps to the center of the sphere and the atomic radius (Van-der Walls radius of an element for that particular atom) is mapped to the radius of the sphere.

Since each atom is represented as a sphere, classically these spheres have to be tessellated into triangles before they can be rendered. This essentially limits the visualization to molecules with only a small number of atoms because in order to obtain high quality images, a large number of triangles is required to obtain a smooth surface. As mentioned in chapter 3, this is challenging especially in JavaScript because its performance is usually slow compared to compiled languages and therefore tessellating thousands of spheres would be slow. The resulting data would also create a communication bottleneck between the CPU and the GPU, depending on the dataset size. Moreover, polygon rendering techniques do not generate high-quality images compared to those obtained by performing ray-object intersection using the implicit surface equation of the sphere.

This chapter describes an implementation of the techniques for visualization of molecular structures in the web browser using GPU-based ray casting. The work discussed in this chapter has been published in a paper [Mwalongo et al., 2014]. Figure 4.1 shows some visualizations rendered using techniques described in this chapter.

4.1 GPU-based Ray Casting

GPU-based ray casting is considered as the state-of-the-art technique for the visualization of molecular structures. It enables high-quality images and allows interactive rendering of molecules with very large number of atoms [Grottel et al., 2015]. This is achieved by leveraging the power of the GPU that can run thousands of threads in parallel.

GPU-based ray casting renders the model by performing ray-sphere intersection tests for each atom in a fragment shader. A proxy geometry that covers the whole object is rendered and the implicit description of the object is sent to the shader. For a sphere, for example, only the center and radius are needed. In the fragment shader, a ray from the camera position through the current fragment is constructed and intersected with the object, similar to ray tracing without secondary rays. Local lighting (e.g., Blinn-Phong-shading) can be used to render the object. This method, first introduced by Gumhold [2003] and Klein

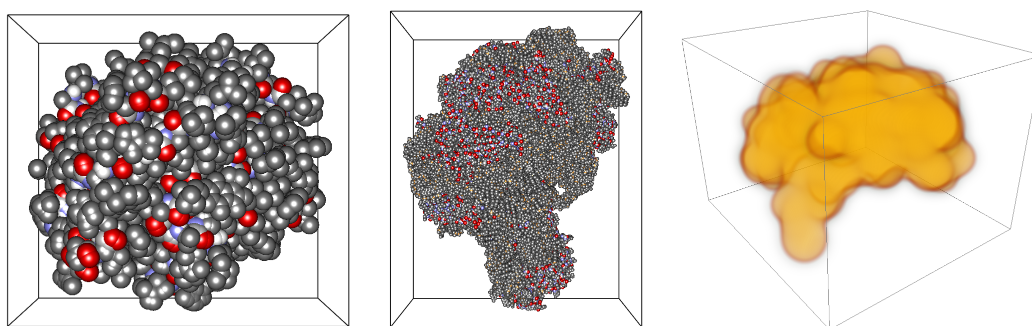


Figure 4.1 — Visualization of spacefilling model for proteins using our WebGL glyph ray casting method. Left: A lactamase (PDB ID: 1M40; 5,712 atoms), middle: a large ribosomal subunit (PDB ID: 1N8R; 90,418 atoms). The implementation of the sphere ray casting described in this paper is similar to volume ray marching. The image to the right shows a WebGL volume rendering of the approximate electron density distribution of an insulin protein (PDB ID: 1RWE, 823 atoms; volume resolution: 99×74×95 voxels).

and Ertl [2004], has been used for rendering molecular models (e.g., [Reina and Ertl, 2005; Sigg et al., 2006]) and is also used in popular molecular visualization tools like VMD [Humphrey et al., 1996].

Moreover, by using the compact, implicit description of the geometry for rendering on the GPU, less data has to be transferred from the CPU to the GPU, thus minimizing the bandwidth usage. This is critical in the browser environment whereby in order to obtain much performance, more computations have to be pushed to the GPU and less data has to be uploaded to the GPU so that the CPU code has more time handling user interaction events.

Performing ray-sphere intersection is computationally expensive and, therefore, benefits from acceleration structures to speed up the computation. There are several acceleration structures that can be used, each with different performance characteristics [Wald et al., 2007].

Generating these structures is done in a preprocessing step. This step can introduce significant computational overhead, especially for dynamic data in which case the acceleration structure has to be generated in each frame. However, for static data, as it is in our case, this overhead is incurred only once.

In our implementation we use a 3D grid data structure due to its low computational overhead in creating the data structure and traversal of the data structure during rendering in the fragment shader.

A different technique that does not require an acceleration structure uses a proxy geometry for each sphere. The center of the sphere and its radius are sent

to the GPU for each atom and the proxy geometry for each sphere is rasterized to generate fragments that cover the bounding box of each atom in screen space. In the fragment shader, a ray is cast for each fragment and a ray-sphere intersection is performed.

Since behind each proxy geometry there is a single sphere, no traversal is required in the fragment shader. Thus, this approach reduces the overhead of traversing the grid acceleration structure. However, to get correct intersections, the fragment shader requires the capability of writing the fragment depth (*gl_FragDepth*). This capability is not part of the WebGL 1.0 [Khronos, 2011b] standard, but is provided as an extension. However, at the time of implementation, this extension was not yet supported by the browsers, so a technique first proposed by Lindow et al. [2012] was adopted to suit the WebGL constraints. The ability to replace the depth value in the fragment shader is now part of the standard in WebGL 2.0.

4.2 Acceleration Data Structures

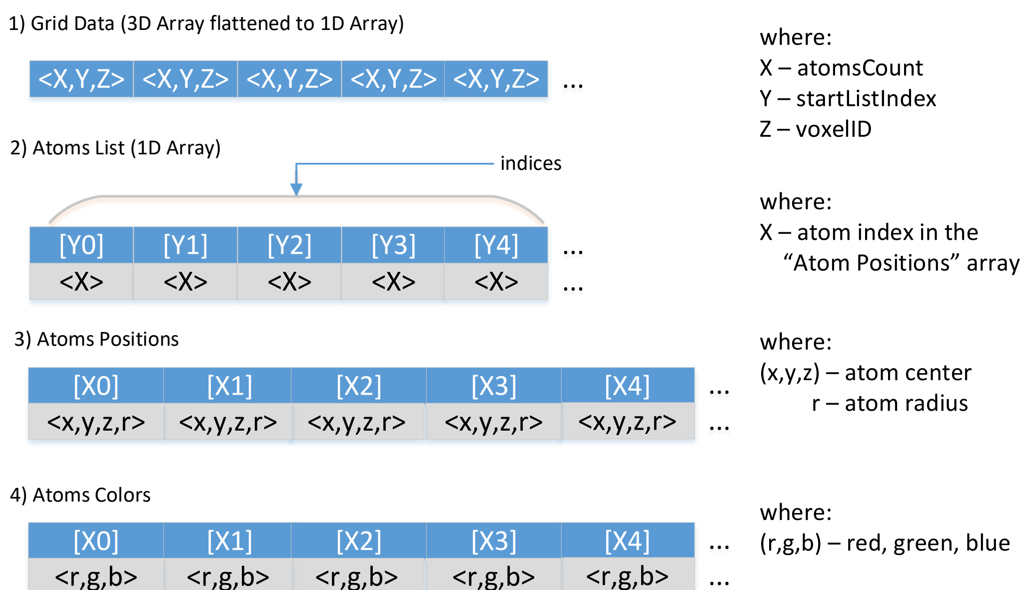


Figure 4.2 — Data structures for the grid that is used for ray casting the spheres. The atoms (position in $\mathbb{R}^3$ and radius) and their colors (RGB) are also stored on the GPU and referenced by the grid [Mwalongo et al., 2014].

As mentioned above, the acceleration data structure is usually built in a preprocessing step. The structures used are as shown in Figure 4.2. In our prototypical

implementation, which is described in the next section, the data structures are created on the server and sent to the client in JSON format for rendering. Once the data is received on the client, it is decoded, packed as textures and then uploaded to the GPU for rendering.

4.3 Implementation

The visualization processing pipeline is implemented using a client-server architecture. The server performs the preprocessing steps and the client performs the rendering and handles user interactions. The choice of a client-server architecture allows to offload costly computations to the server, and reduces the computation load on the client. Performing the preprocessing on the client was also implemented in order to compare the performance with that of performing the preprocessing on the server. Performing the preprocessing on the server performed better than the alternatives (see section 4.4). Figure 4.3 and Figure 4.4 show an overview of the implementations of the two approaches.

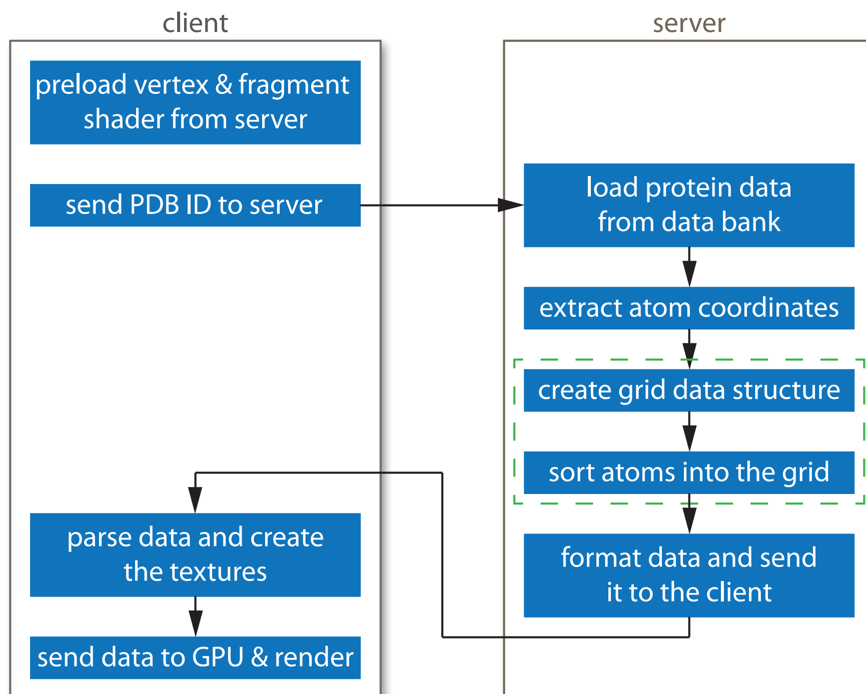


Figure 4.3 — Server-side preprocessing.

The server creates a 3D uniform grid structure by sorting the atoms in each cell of the grid. We use a compact grid data structure as proposed by Lagae and Dutré [2008]. The data structure uses two static arrays for storing a concatenated list of atoms in each cell and another array for the grid itself. Each cell stores

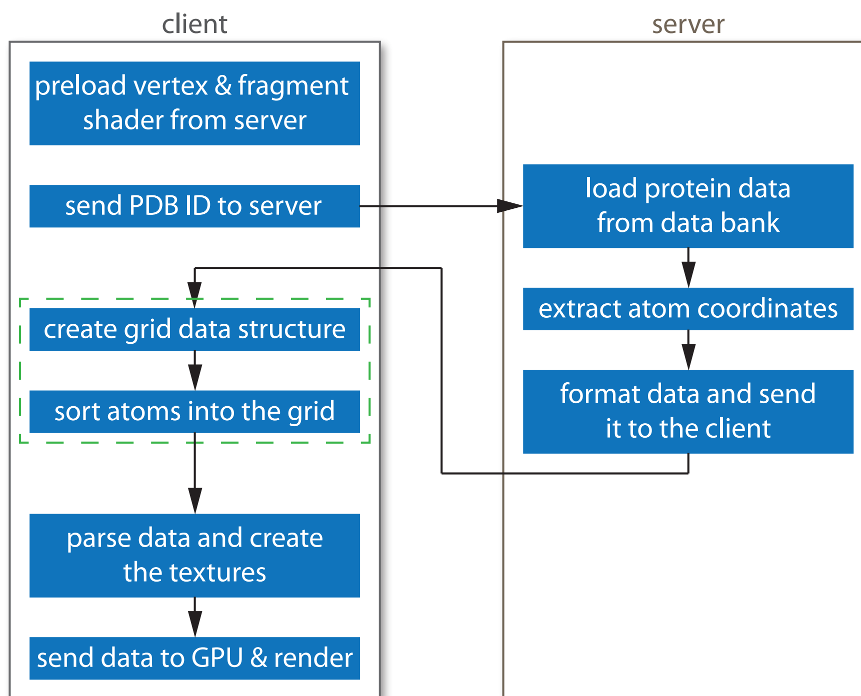


Figure 4.4 — Client-side preprocessing. The dashed green box highlights the part of the computation that is shifted between the client and the server.

the starting index of the list of atoms and the total number of the atoms that overlap it. They used a special *sentinel* value to indicate where the list for each voxel ends. Since we store the number of atoms in each cell, we do not have to use such a sentinel to indicate where one list ends in the concatenated array. The number of atoms in each cell is used to iterate through the atom list during the traversal stage.

Two approaches can be used to sort the atoms in each cell of the grid. The first approach uses sphere-box intersection test [Larsson et al., 2007]. This algorithm computes a sphere-box intersection test with each cell for every atom. The atom is added to a particular cell only if it intersects it. However, this method has the disadvantage of performing many sphere-cell intersection tests and can affect the performance significantly especially for large molecules.

The second approach uses a bounding-box method [Pharr and Humphreys, 2010]. This method first finds the bounding box of each atom and then the atom is added to every cell of the grid that is covered by the extents of the bounding box. Although this algorithm can add more atoms even in cells where the atom does not overlap the cell, it avoids many cell-sphere intersection tests.

Each cell of the grid stores the number of atoms that overlap that cell and

the start index of the first atom in the list of atoms for that cell. The atom positions are parsed from a PDB file that can be automatically downloaded from the RCSB Protein Data Bank [Berman et al., 2000] or from a file stored locally. In addition to sorting the atoms in the grid data structure, the server also assigns the color to each atom depending on the chosen coloring mode. In our implementation, we use the CPK coloring scheme [Koltun, 1965], which assigns a specific color to each element. Because WebGL 1.0 does not support 3D textures, the data is linearized and packed into an array and sent to the client including the bounding box of the molecule.

The client decodes the received data and creates textures that are uploaded to the GPU. In WebGL 1.0, only 2D textures are supported. Therefore 1D and 3D textures are packed as 2D textures and address translation is used to access the correct values from the textures in the fragment shader (see Figure 4.5). 3D textures have now become a part of the standard in WebGL 2.0 [Khronos, 2013].

```
Voxel getVoxel(ivec3 pos)
{
  //pos is in i,j,k voxel coordinates
   Voxel voxel;
   voxel.voxelCoord = pos;
   voxel.pos = voxelToPos(pos); //convert from voxel coordinates to world coordinates
   int idx = threeTo1D(voxel.voxelCoord,numVoxels); //get linear index for the i,j,k coordinate
   vec2 texCoord = oneTo2D(idx,uGridTexSize); //get texture coordinates for the 2D texture ( 0.0 ...1.0)
   vec3 voxelData = texture2D(uGridTexture,texCoord).rgb;  //retrieve the voxel data;//r=count,g=startIndex,b=voxelID
   voxel.count = int(voxelData.r); // count is the number of atoms in the voxel
   voxel.startIndex= int(voxelData.g); //start index in the AtomsList
   voxel.voxelID = voxelData.b;
   return voxel;
}
```

Figure 4.5 — Fragment shader code for accessing voxel data from the grid data structure. Because WebGL 1.0 does not support 3D textures, the data is packed as 2D textures (image source: Mwalongo et al. [2014]).

All the data required for rendering is stored as 2D textures, as this is the only type supported by WebGL. Grid data is stored as 2D `GL_RGB` texture, the atoms list is stored as 2D `GL_LUMINANCE` texture, atom positions and radii are stored in a single 2D `GL_RGBA` texture, and colors of atoms are stored as a 2D `GL_RGB` texture.

To initiate the rendering pipeline, the front-faces of the bounding box of the molecule are rendered. The grid traversal and ray-sphere intersection tests are performed in the fragment shader. For each fragment, a ray originating from the eye position through the center of the current fragment position is generated and each cell is visited using the traversal algorithm introduced by Amanatides and Woo [1987]. If the cell has no atoms, the traversal proceeds to the next cell in the same direction, otherwise a ray-sphere intersection is performed for all the atoms that are found in that cell. The nearest intersection point from the ray

Table 4.1 — Performance measurements for the client-side preprocessing and the server-side preprocessing. All times are given in milliseconds (ms). The grid density was set to 5 for all tests.

		Client-side			Server-side			WebGL	OpenGL	
PDB ID	**#Atoms**	**Preproc.**	**Mem.**	**Loading**	**Preproc.**	**Mem.**	**Loading**	**Rendering**	**Rendering**	**Grid Res.**
1CCN	327	17	37 kB	8	2	65 kB	10	427 fps	452 fps	11×13×10
1OGZ	944	35	49 kB	12	2	185 kB	20	291 fps	376 fps	18×14×17
1M40	5 712	248	291 kB	49	7	1.7 MB	108	122 fps	156 fps	28×26×37
1AF6	10 050	309	526 kB	68	12	2.3 MB	140	126 fps	159 fps	38×39×32
1N8R	90 418	2 240	5.4 MB	345	65	20 MB	1 800	30 fps	47 fps	65×82×82

origin is kept along the way. The traversal is terminated when it reaches the end of the bounding box.

To optimize performance, the ray can be terminated early once the nearest intersection point is found. The intersection point is shaded using Blinn-Phong shading model. If the traversal reaches the end of the bounding box without hitting any atom, then the current fragment is discarded.

4.4 Results and Discussion

We measured the rendering performance as well as the times for preprocessing and data transfer using several protein data sets from the RCSB Protein Data Bank [Berman et al., 2000].

The test platform was a Windows 7 PC with an Intel Core i7-2600 (3.4 GHz), 8 GB RAM and an NVIDIA GeForce GTX 560 Ti. We used the Tomcat 7 web server and the Mozilla Firefox web browser (version 27.0.1 using WebGL 1.0 without ANGLE binding and WebGL GLSL ES 1.0). The canvas size was set to 768×768 pixels for all measurements.

Table 4.1 shows the results of our performance measurements for client-side preprocessing as well as server-side preprocessing. *Preproc.* is the time required to sort all the atoms in the grid, excluding the time to calculate the grid parameters. *Mem.* denotes the amount of memory that is transferred from the server to the client. *Loading* is the time for downloading the data used in creating the textures from the server to the browser of the client (the server and the client have the same hardware and are connected to a local gigabit Ethernet network). The time does not account for the loading of other files making up the page, like the shaders, which are all loaded in the browser when the page is first accessed. *WebGL Rendering* is the time for rendering the protein in the GPU. We calculate this time by forcing the draw calls to block (using *glFinish()*) and measure the elapsed time for the draw calls. *Grid Res.* gives the grid resolution, that is, the number of cells in the x, y, and z dimension.

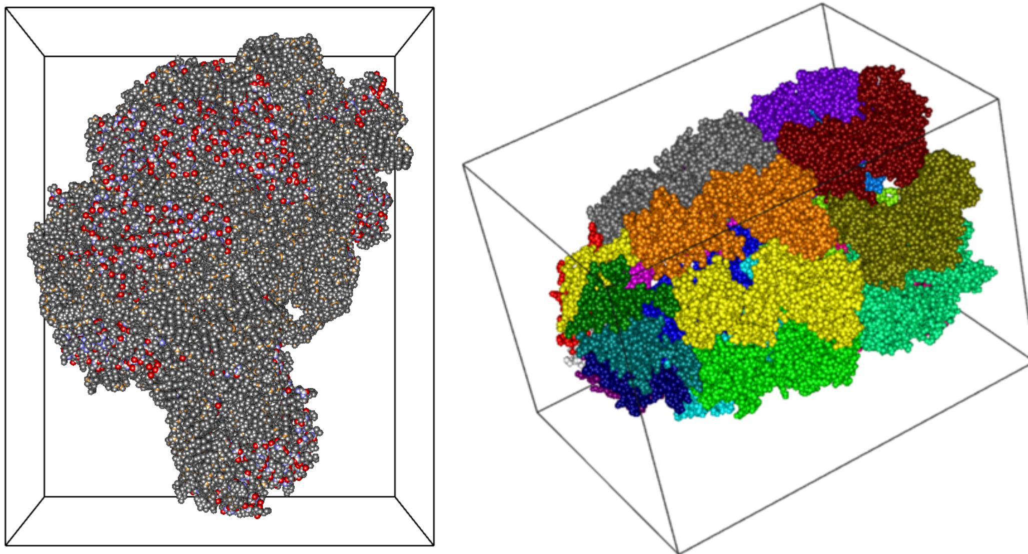


Figure 4.6 — Left: Structure of large ribosomal subunit in complex with virginiamycin M (PDB ID: 1N8R) with 90 418 atoms (image source: Mwalongo et al. [2014]). Right: Visualization of the crystal structure of the asymmetric GroEL-GroES-(ADP)7 chaperonin complex (PDB ID: 1AON) with 58 674 atoms.

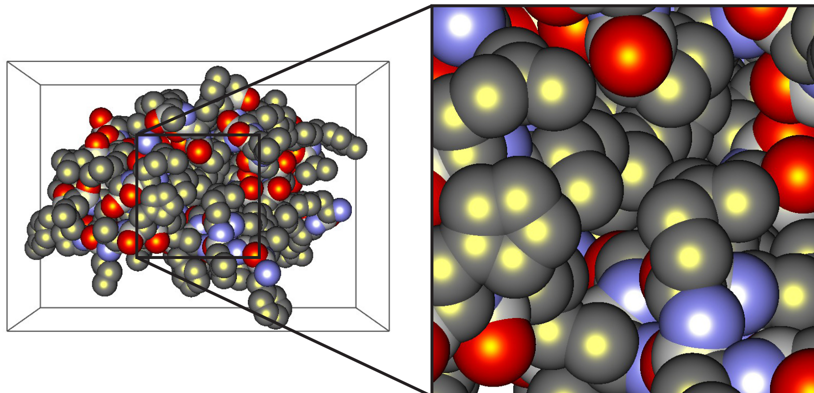


Figure 4.7 — Visualization of an insulin protein (PDB ID: 1RWE, 823 atoms). The cutout to the right shows a close-up view. Our shader-based WebGL ray casting generates pixel-accurate spheres and correct sphere-sphere intersections (image source: Mwalongo et al. [2014]).

The performance results given in Table 4.1 show that it is feasible to achieve high-quality images at interactive frame rates even for molecules with a large number of atoms (see Figure 4.6). A close-up view of the visualiazion of a small protein shown in Figure 4.7 highlights the visual quality of the images generated by the WebGL-based ray casting described in this chapter.

Another observation of the results in Table 4.1 shows that the grid resolution influences the performance of the rendering speed. High grid resolution results in large number of cells that can increase the grid traversal time in the fragment shader. On the other hand, low grid resolution results in fewer cells but can result into more atoms in a single cell, thus increasing the computation time for ray-sphere intersection in each cell. Therefore, choosing the right grid resolution is important in order to achieve better performance.

The grid resolution is influenced by the grid density, which is defined as the ratio of cells and primitives, given as $\rho = M/N$ where M is the number of cells and N is the total number of primitives (in our case spheres). Several studies (see e.g., [Haines, 2001], [Shirley, 2002], and [Wald et al., 2006]) have shown that a number between 2 to 10 gives good performance. In our prototypical implementation we used 5 as the grid density, which gave good results in our measurements. Increasing this value increases the grid resolution and, hence, the number of grid cells. Consequently, it would result in higher times to traverse the grid and more data to transfer (due to the duplication of atoms in adjacent grid cells). In contrast, lower density values would result in larger grid cells that contain more atoms in a single cell. In this case, more unnecessary sphere intersection tests per cell would have to be computed in the fragment shader, which would also lower the performance. Moreover, the WebGL security model [Khronos, 2011c] discourages long running shaders as a mechanism to prevent itself from denial of service attacks. Therefore browsers usually terminate long running shaders, leading to lost WebGL context.

In order to evaluate the WebGL rendering speed, we implemented the sphere ray casting algorithm as a desktop application using C++ and OpenGL 2.0 with GLSL 1.3 shaders, which is a basis for WebGL 1.0. The frame rates can be found in the *OpenGL Rendering* column in Table 4.1. The performance of the WebGL rendering is comparable to the OpenGL rendering, which is a promising result for interactive web-based visualization.

As mentioned in section 4.3, we implemented two approaches involving server-side and client-side preprocessing. The server side was implemented using Java, while the client side was implemented using JavaScript. Table 4.1 allows us to compare the performance of the two approaches. The results show that performing preprocessing on the server side gives considerably better performance compared to the one performed on the client side. As the algorithmic implementation is the same, this difference in performance may be attributed to the difference in language efficiency as Java in general performs better than JavaScript.

On the aspect of data transer, time taken for sending data from the server to the client is higher when the preprocessing is done on the server compared to

when it is done on the client. This is expected since the preprocessing stage generates additional data structures (i.e., the grid data structure and the atoms list), which add to the size of the data sent to the client. Moreover, higher data transfer times can be attributed to the additional processing done by the server to generate the data structures—data transfer times include preprocessing time by the server. For our tests in a local network environment, the combined times for preprocessing and downloading of the data to the client was lower if server-side preprocessing is used (see Table 4.1). If the network connection is slow, however, the data transfer times can of course predominate for large data.

Nevertheless, we argue that the server-side preprocessing is superior to the client-side one, not only for our intended use case, but also because it was faster in all our tests. Server-side preprocessing approach allows us to guarantee that large molecules can be visualized because the faster server application executes the heavy computations. The server might even offload these computations to a dedicated compute cluster or cloud. In addition, our solution has only moderate hardware requirements on the client side, which makes it even feasible for mobile devices with less computational power than a desktop PC. Additionally, the server-side preprocessing can involve special hardware capabilities like CUDA computations that would not be possible on a client using a JavaScript application.

As mentioned in Section 4.1, the sphere ray casting method we use is conceptually similar to modern, GPU-based volume ray marching as, for example, presented by Krüger and Westermann [2003]. Congote et al. [2011] described how to implement volume ray marching using WebGL 1.0, which also poses difficulties since there is, for example, no native support for 3D textures. The 3D volume has to be stored in one or more 2D textures—similar to the grid in our implementation—and the trilinear interpolation of the voxel values has to be implemented in the fragment shader. We included a volume rendering similar to the one described by Congote et al. [2011] into our WebGL visualization. Figure 4.1 shows a screenshot of our volume rendering. Volumetric representations of a molecule can, for example, show the electron density distribution or can be used to extract a smooth molecular surface. The volume rendering can be combined with geometric models such as the triangle rendering of the Cartoon representation shown in Figure 4.8, which are rendered to a Framebuffer Object prior to the ray marching. The required volume data sets can, for example, be generated using the *VolMap Tool* in VMD [Humphrey et al., 1996], which can also be used to generate triangulated models such as the aforementioned Cartoon representation.

This chapter has presented techniques that demonstrate the feasibility of imple-

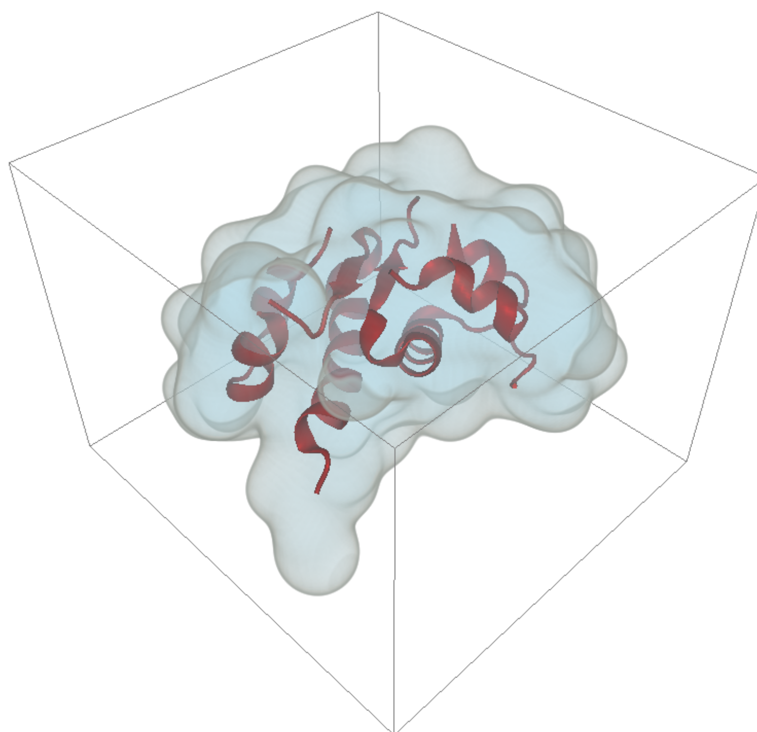


Figure 4.8 — Volume rendering of an insulin protein (PDB ID: 1RWE, cf. Fig. 4.7) combined with a triangulated Cartoon representation. The volume data set is the same as shown in Figure 4.1 using a different transfer function (image source: [Mwalongo et al., 2014]).

menting GPU-based ray casting for molecular visualization using WebGL 1.0. The results show that large protein structures can be visualized at interactive frame rates compared to classical rendering using triangulated spheres. To achieve this, costly preprocessing computations need to be offloaded to a server and generation of the geometry for visualization has to be done on the GPU. Generating the geometry on the GPU using GPU-based ray casting technique does not only allow less data to be sent to the GPU but also generates high quality images compared to techniques that rely on polygonal surface approximations.

In order to efficiently use the bandwidth between server and client, efficient data formats are required. In this implementation, JSON was used because it is more efficient compared to popular formats like XML and it is also easy to parse the data on the client using built-in browser functions. For more optimizations, binary formats using *typed arrays* are preferred because they are not only bandwidth efficient but also do not incur much decoding penalty on the client before they can be uploaded to the GPU. By sending data encoded

in the format that is close to the format required by WebGL buffers, the data can immediately be uploaded to the GPU with minimal processing required. Additionally, using *WebSocket* [Fette and Melnikov, 2011] as the application communication protocol can improve latency, since it provides a full-duplex communication channel compared to the HTTP protocol [Fielding, 2000], which provides only a half-duplex communication channel. These optimizations are used in a technique that is discussed in chapter 5.

CHAPTER 5

GPU-based Remote Visualization of Dynamic Molecular Data on the Web

Visualization of dynamic data from molecular dynamics simulations is crucial for understanding many functional aspects of molecules. By visualizing data resulting from these simulations, scientists can gain insight into the structure and dynamics of molecular mechanisms at atomic level. In order to get accurate results, simulation models require many time steps [Frenkel and Smit, 2001]. The data resulting from these simulations contain, at least, the position of each atom at every time step of the simulation and is normally written to a file that is usually called a *trajectory*. However, in order to reduce storage requirements, the data are usually not written after every time step. After the simulation completes, this trajectory file can then be loaded for analysis using a visualization tool like VMD [Humphrey et al., 1996] or MegaMol [Grottel et al., 2015].

Advances in computing technology have made it possible to simulate very large and complex molecular systems with longer simulation time. These simulations usually require powerful computational resources involving clusters of computers. Consequently, the massive data produced become infeasible to be moved to a different machine for visualization. In order to visualize these large data, a different approach is needed that does not require moving the raw data at once from the storage server to the client.

As discussed in chapter 3, the browser has become a preferred platform for deployment of visualization tools because of its ubiquity and advances in web technologies including GPU-access in the browser through JavaScript. Despite these advances, existing web-based molecular viewers (e.g., [Jmol, 2009; JSmol, 2013; Pettit and Marioni, 2013; Li et al., 2014]) are still limited to static data and molecules with small number of atoms. Although Jmol [Jmol, 2009] can support playing animations and movies created from a number of supported trajectory files, it suffers from security issues because of its plugin-based technology. JSmol [JSmol, 2013] alleviates the problem of plugins but it cannot handle the visualization of large molecules and molecular trajectories because most processing is performed on the CPU using JavaScript rather than on the GPU.

Moreover, all these tools use triangulated spheres for rendering, which does not scale well when visualizing molecules with large number of atoms as discussed in chapter 4. This approach becomes even more challenging when used for dynamic data, which can potentially change in every frame. Since the spheres have to be tessellated before rendering, many triangles are needed for a molecule with a large number of atoms, This can affect the rendering performance because of large data uploads to the GPU. Moreover, to obtain quality images with polygon-based rendering, fine tessellation is required which results into more triangles being generated. This does not only cause heavy computations to be performed on the CPU but also causes CPU-GPU bandwidth problems. Therefore, in order to be able to build interactive visualization tools that support large dynamic data, rendering techniques that avoid heavy computations on the CPU and send less data to the GPU are required. Browser-based visualiazion of large dynamic molecular data also requires efficient data transfer between server and client.

This chapter discusses an implementation of a web application that supports interactive visualization of large dynamic molecular data in the browser. Techniques described in this chapter exploit modern web standards like web socket, web workers, typed arrays, and GPU-based ray casting to enable visualization of large dynamic molecular at interactive frame rates. In order to save bandwidth, quantization techniques are applied to reduce the amount of data that is transferred to the client. The results discussed in this chapter were published in a paper at the Web3D conference [Mwalongo et al., 2015] and was subsequently extended as a journal paper [Mwalongo et al., 2016a].

5.1 Overview

The application follows a client-server architecture as shown in Figure 5.1. The client initiates the interaction by sending a request URI to the server. The server responds by sending HTML, CSS, JavaScript, and WebGL shader files to the client. These files constitute the client-side of the application. Other parameters like data to be visualized and visualization options like coloring mode can also be encoded in a URI and sent to the server anytime during interaction.

After receiving the files, the client establishes a WebSocket connection with the server. This connection is used for transferring the raw data for visualization. The format for the transferred data is as shown in Figure 5.2. The data format can encode both triangulated surfaces and parameters for implicitly defined surfaces, for example, atom centers and radii. When only one type of the data is present, the field for the number of elements on the missing type is set to zero. The format is encoded so as to ensure that there is minimal processing done on the client before the data are uploaded to the GPU. By ensuring less computation in JavaScript on the client side, performance is improved. This is crucial for dynamic data that can potentially change in every frame.

5.2 Implementation

As discussed in section 5.1, the application follows a client-server architecture. The server side consists mainly of a WebSocket server and a data management component based on an existing visualization framework called *MegaMol* [Grottel et al., 2015]. *MegaMol* was extended by adding a WebSocket server component on top of its data management component in order to support data transfer to the client in the browser using the WebSocket protocol.

The WebSocket server includes a HTTP server that responds to initial requests that results into the files forming part of the client side of the application to be sent to the client. Client-side application includes HTML5, CSS, JavaScript, and WebGL shader files. One of the JavaScript files sent to the client contains code for establishing a WebSocket connection back to the server. Once the connection has been successfully established, the data transfer between client and server begins. Data requests from the client are received by the WebSocket server component, which then forwards them to the data management component.

The data management component loads the frames into memory from a trajectory file and encodes each single frame in a binary byte array data format as shown in Figure 5.2 and sends it to the client for rendering. Each frame consists of data for a single time step of a simulation.

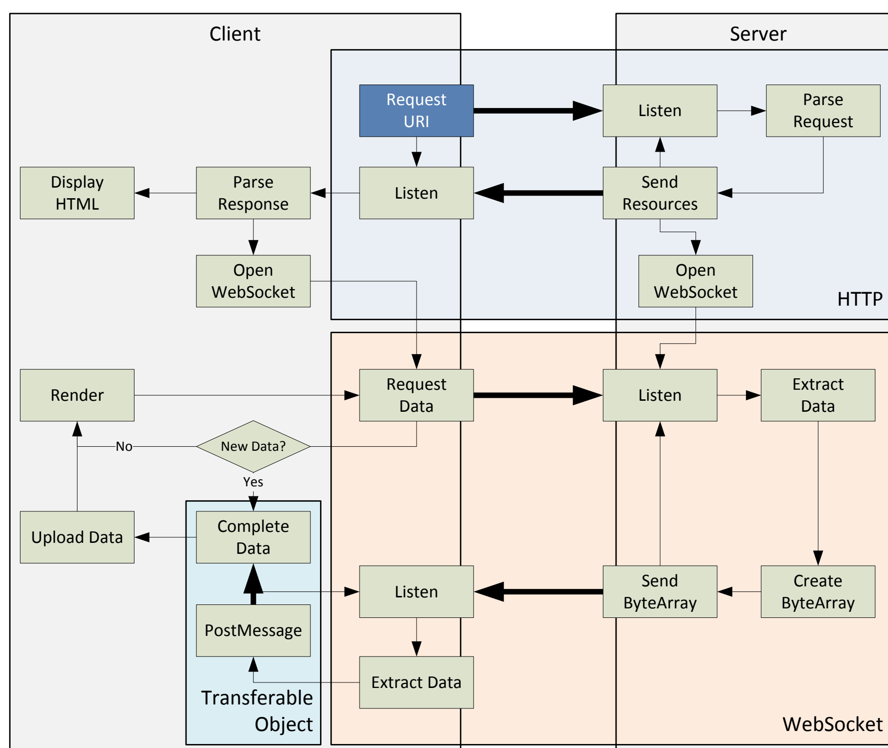
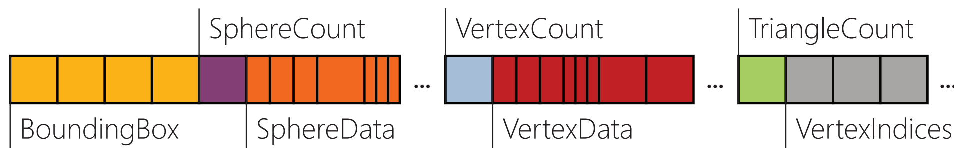


Figure 5.1 — Our application architecture. Thin arrows represent control flow within server or client, respectively. Bold arrows represent data flow between memory spaces according to the protocol/interface illustrated by the colored boxes (HTTP, WebSocket, Transferable Object). The entry point for the user is the request URI in the client.

The rendering stage of the visualization pipeline is implemented on the client in order to ensure interactivity. Only the data that is necessary for the kind of rendering requested by the client is sent to the client. Requests for new frames of data from the server are controlled by the client. The client always notifies the server of its readiness to receive new data once it has issued a draw call to the GPU; this way the client is not overloaded by data from the server, and also ensures that all frames are rendered.

In order to decouple the rendering thread and the data fetching thread, the fetching of data is implemented in a web worker. This separation is important for performance reasons, given that JavaScript is generally single-threaded. Therefore, to allow the user interface to remain responsive, all long-running code paths need to run outside the main thread. Network communication is an example for such a long-running path that would stall the main thread while waiting for data transmission to complete. The communication between client and server using WebSocket happens on the web worker on the client side.



BoundingBox: 4 floats (center XYZ, longest edge);
SphereCount: 1 unsigned int;
SphereData: SphereCount × 13 bytes: 3 unsigned shorts + 1 float + 3 unsigned bytes
(center XYZ, radius, color RGB);
VertexCount: 1 unsigned int;
VertexData: VertexCount × 17 bytes: 3 unsigned shorts + 3 unsigned bytes + 2 floats
(position XYZ, color RGB, normal $n_X n_Y$);
TriangleCount: 1 unsigned int;
VertexIndices: TriangleCount × 3 unsigned int.

Figure 5.2 — Memory layout for the data buffer sent by the server. Each square represents a block of four bytes (e.g., a floating point value or an unsigned integer). Half-squares represent two bytes (unsigned shorts), quarter-squares a single byte.

Once the data is received from the server in the web worker, they are posted to the main thread for rendering. Since web workers and the main thread do not share memory and communicate only by passing messages, copying large amounts of data between them can lead to significant penalty on performance and memory. Therefore a feature called *transferable objects* is used for data transfer between the web worker and the main thread. This capability allows the ownership of the memory to be transferred instead of copying the data. This decoupled and asynchronous communication between rendering and data fetching is crucial because it allows the rendering thread to continue rendering the data that has already been uploaded to the GPU without being stalled by the network transfer delays. Moreover, the user can continue interacting with the visualization while new data is being transferred.

5.2.1 Data Encoding and Quantization

The data encoding format as shown in Figure 5.2 encodes both implicit and explicit geometry. The data for implicit geometry, in our case spheres, consists of the bounding box, number of spheres, and sphere data comprising of sphere centers and their radii, and colors. The bounding box is required for the camera setup at the client.

Our explicit geometry is a set of triangles whose encoded data consists of

the number of triangles, vertices, indices, and the vertex data themselves that consist of position, color, and normal of each of each vertex. An important consideration for data encoding is to ensure efficient data transfer through the Internet and less computational burden on the client. This is the reason why we use a binary byte array format representation for efficient data transfer and ease of decoding in the client before uploading the data to the GPU for rendering.

In order to further save bandwidth and reduce transfer times we employ data quantization in addition to using an efficient binary format for encoding the data. Atom positions are encoded using 16 bits instead of the usual 32 bits. Therefore, rather than using 12 bytes per atom, we use only 6 bytes per atom, thus cutting down the memory by half. This is possible because the position data can be encoded at lower precision without affecting the image quality.

The quantization is done by normalizing the atom position with respect to the bounding box of the molecule and scaling the results within the range of unsigned short integers. The scaled values need to be clamped to fit the integer range from the original floating point values. This of course introduces a small quantization error whose magnitude depends on the size of the bounding box. However, this error is negligible and does not affect our results.

The radii are stored as single precision floating point values and the color information is reduced to a single byte per channel. This reduces the number of bytes to only 13 from 28 bytes when simply using single precision floating point values for all elements.

Similar to the atom data, vertex positions are also quantized to relative coordinates inside the bounding box, while color precision is reduced to single bytes per channel. Additionally, the third component of the normal is omitted and reconstructed from the first two, reducing vertex data in total from 36 bytes to only 17 bytes. Since the transferred data contain triangulated surfaces and atom data that are mapped to describe implicit surface of a sphere for each atom, the client side implements two rendering approaches: GPU-based ray casting for atom data and normal triangle rasterization for the triangle meshes.

5.2.2 WebGL GPU-based Ray Casting

GPU-based ray casting is used to render implicit surface of the spheres. As explained in chapter 4, this rendering technique has the advantage of producing high quality images and reduces the amount of data transferred to the GPU because only the parameters defining the implicit sphere are uploaded, thus saving the CPU-GPU bandwidth. The implementation described in chapter 4 maintained compatibility with WebGL 1.0 while the implementation in this

chapter uses a feature of WebGL 2.0, available as an extension in WebGL 1.0, to write the fragment depth (*gl_FragDepth*) to the depth buffer in order to get perspective-correct depth values for the spheres. This implementation is also simpler compared to the one used in the previous chapter. Moreover, this technique is suitable for dynamic data because it does not require any acceleration structure that may cause preprocessing overhead as discussed in chapter 4. Using acceleration structures would require that the structures be generated for every frame, thus increasing computational time.

Before rendering, the data format as described in section 5.2.1 has to be decoded. Since the data is heterogeneous, there are two options that can be used to decode the data. One option is to create different typed array views to a single *ArrayBuffer* object representing the raw binary data. Each view starts at a different byte offset where the particular data type begins. Another option is to use the *DataView* object. However, due to quantization, the layout in the raw data becomes non-uniform, making decoding the data using these objects slow. This slowness could probably be due to differences in byte-alignment between the raw data and the typed array views. We, therefore, directly copied the raw bytes from the raw binary data into an *ArrayBuffer* object using bitwise operators.

To render the spheres using GPU-based ray casting [Gumhold, 2003], proxy-geometry data for the atoms are stored in a single vertex buffer object (VBO). A rectangular proxy-geometry is used and, therefore, each atom requires two triangles. In the vertex shader, a screen-space bounding box of each sphere is calculated for generation of fragments during rasterization step. This is done in order to avoid expensive computations in the fragment shader for all the fragments that are not covered by the sphere. That means the rasterizer generates fragments covering only the projected bounding box of the sphere.

In the fragment shader, a ray is cast for each fragment and a ray-sphere intersection test is performed. If the ray hits a point on the surface of the sphere, then that point is shaded and becomes the color of the fragment. If the ray does not intersect a sphere, then the fragment is discarded. Additionally, the perspective correct depth value is written to the *gl_FragDepth*. This requires an extension *gl FragDepthEXT* in WebGL 1.0, but in WebGL 2.0 it is part of the core standard.

5.2.3 Triangle Rendering

For rendering triangles, there are two ways that they can be defined: using only vertex positions or vertices together with an index buffer that defines the triangles and references the vertices. When triangles are defined using only vertex positions, the *TriangleCount* field is set to zero. Therefore, each triangle is

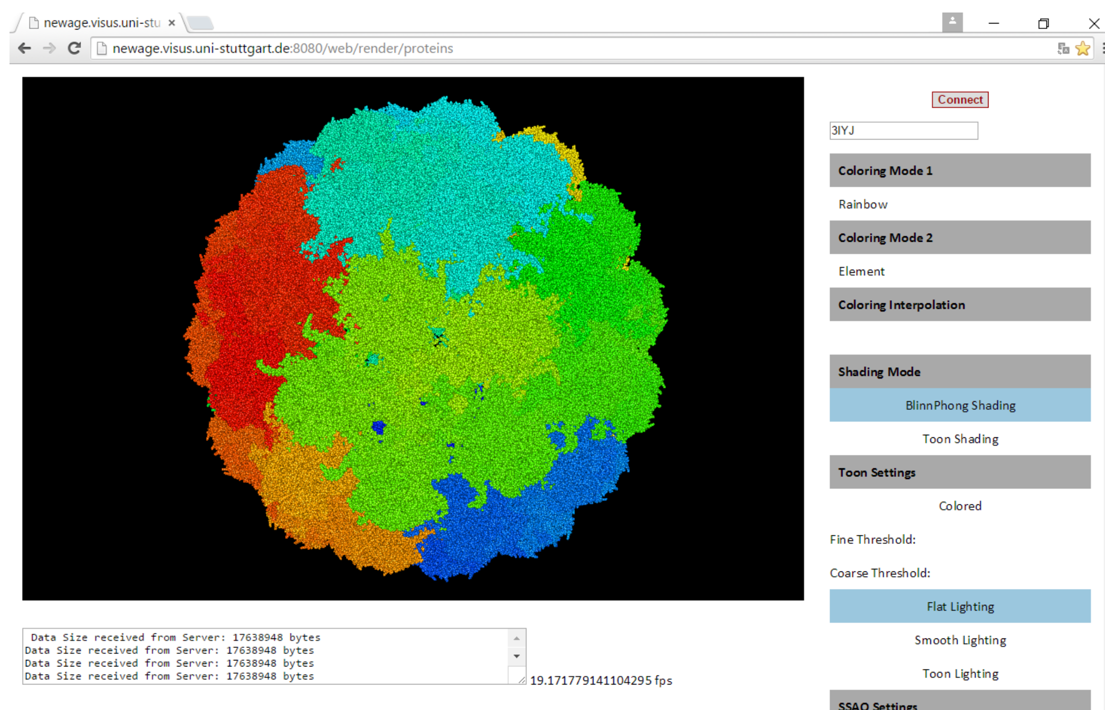


Figure 5.3 — The visualization of the capsid of a papillomavirus (PDB ID: 3IYJ) consisting of 1.3 million atoms. On the right hand the user interface for adjusting the rendering modes and parameters is shown.

constructed using three consecutive vertices and the value for *TriangleCount* is found by dividing the number of vertices by three.

In the case where the triangles are defined using vertices together with an index buffer that references the vertices, the field of *TriangleCount* is given explicitly. Multiplying this field by three gives the value of *IndexCount*. This means that three vertex indices are stored for each triangle. This way of defining triangles has the advantage of storing each vertex only once, thus reducing the data size.

Both triangle formats can directly be rendered by WebGL using `drawArrays` or `drawElements`. Since each vertex in the raw data has a 17 bytes after quantization, padding is done to insert additional bytes between vertices in order to meet WebGL requirements of having 4-byte alignment vertices in the vertex buffer. Aside from adding additional padding between vertices before sending the data to the GPU, no further processing is required in JavaScript, as the index data is simply sliced from the server response as a `Uint32Array` sub-array, while the position, color, and normal data are sliced from the server response as a `Uint8Array` sub-array. Both arrays are uploaded to the GPU as separate buffer objects. Figure 5.4 shows a combination of raycast spheres and a triangle

mesh.

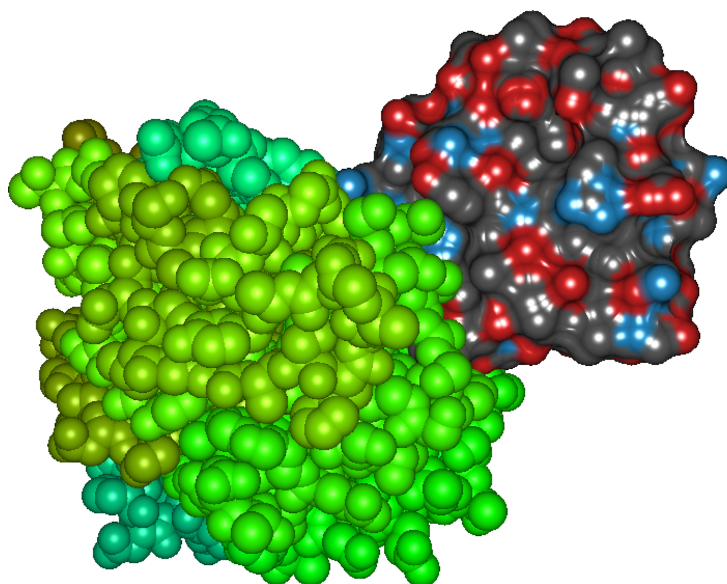


Figure 5.4 — GPU-based sphere ray casting combined with triangle rendering. The image shows a ligand (triangle surface) which is interacting with a receptor (spheres).

5.2.4 Deferred Shading

Deferred shading allows separation of geometry processing and shading calculations into different passes [Saito and Takahashi, 1990]. This technique decouples scene complexity from shading computations by ensuring that shading calculations are performed only for visible fragments. We use this technique to implement toon shading effects [Saito and Takahashi, 1990] for illustrative rendering and screen-space ambient occlusion (SSAO) [Kajalin, 2009] for enhancing depth perception of molecular structures [Tarini et al., 2006] (see Figure 5.5).

In the first pass, which is a geometry pass, the geometry data is rendering and the data required for shading calculations is written to a G-buffer—a framebuffer object with multiple textures attachments. In our case, the data written includes the camera-space position, depth value, albedo color, and camera-space normals. At the time of the implementations we used the `WEBGL_draw_buffers` extension to enable writing to multiple render targets in a fragment shader with a single draw call. This improves performance because it avoids using multiple passes to write each render target separately. This extension is part of the core specification in WebGL 2.0 [Khronos, 2013]. In the shading pass, a screen-filling

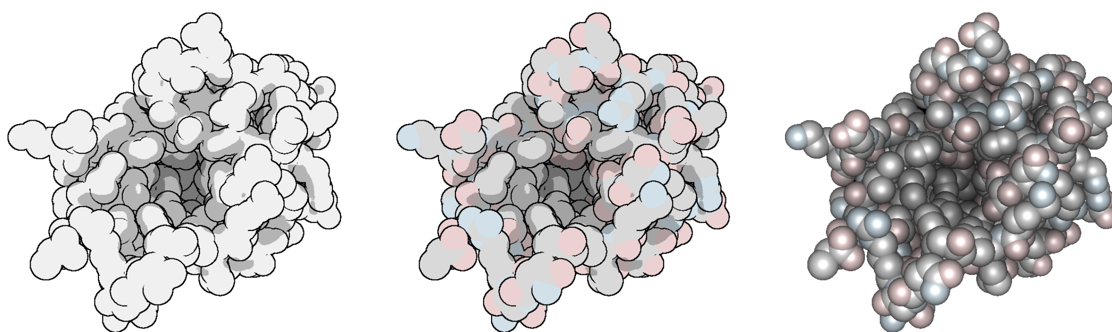


Figure 5.5 — Different postprocessing effects via deferred shading. From left to right: toon shading (silhouettes and quantized screen space ambient occlusion) without colors, toon shading with colors, local lighting and screen space ambient occlusion. The ambient occlusion emphasizes protrusions and depressions (image source: [Mwalongo et al., 2016a]).

quad is rendered, and shading is done in the fragment shader using surface information written in the G-Buffer during the geometry pass as input.

To implement toon shading, an additional render pass computes the depth gradient from the linear depth texture stored in the G-Buffer. This is required because shading calculations for toon shading effects rely on the gradient of the depth texture. This additional render pass takes the linear depth texture from the G-Buffer as input and writes the depth gradient values to another texture. This texture is then used by another shading pass for drawing depth-dependent silhouettes around objects [Saito and Takahashi, 1990].

Screen-Space Ambient Occlusion (SSAO) computation requires three additional passes per rendered frame. The first pass computes per-pixel ambient occlusion factors. This is done by first randomly rotating and orienting a precomputed and randomly generated hemispherical sampling kernel along the surface normal for each pixel. Then for each sample for the respective value stored in the depth texture of the G-Buffer depth values are compared to find samples contributing to an occlusion factor for the given pixel. Depth information and surface normals are taken from the G-Buffer. The two last passes filter these results using a separated Gaussian blur in order to smooth the image by removing noise in the image by the randomly generated sampling kernel in the first path.

The main limitations of implementing these two techniques is the limited framebuffer layouts in WebGL that may result into more GPU memory being used than necessary. For example, although a single value is computed as an ambient occlusion factor for SSAO computations, this value is stored in an RGB texture. Another limitation of WebGL is the lack of support for non-static

Table 5.1 — The systems used for performance measurement including their specifications. The clients were both running Chrome (v.41.0) using ANGLE binding with D3D11 enabled. The WiFi speed of the Laptop was rated at 65 MBit during testing.

System	CPU	RAM	GPU	OS	Network
Server	Intel Core i7-2600 (3.4 GHz)	8 GB	GeForce GTX 560 Ti	Windows 7 Pro x64	GBit LAN
Laptop	Intel Core i7-3520M (2.90 GHz)	8 GB	GeForce GT 640M LE	Windows 8.1 Pro x64	WiFi/GBit LAN
PC	Intel Core i7-2600 (3.40 GHz)	16 GB	GeForce GTX 660 Ti	Windows 8.1 Enterprise x64	GBit LAN

loop control variables. This makes it challenging, for example, to dynamically control the quality of SSAO and the image smoothing by manipulating the radius and the size of the sampling kernel in real-time.

5.3 Results and Discussion

Performance test results focused on transfer times and rendering rates because these are the important parameters for interactive remote visualization. Therefore, we evaluated how fast the data can be transferred from the server to the client and whether rendering speed could maintain interactivity for large dynamic data sets. Molecular datasets used for the tests are from the protein data bank [Berman et al., 2000] and from project partners. The canvas size for all test runs was set to 1280 × 720 pixels. The rendering of the largest molecule of our test dataset (1.3 million atoms) is shown in Figure 5.3.

The test environment included a desktop machine as a server and laptop and a desktop PC as clients in LAN (Gigabit LAN) and Wi-Fi (for Laptop) network environments as shown in Table 5.1. Our tests measured transfer and rendering times for data with quantization and without quantization. These tests were done for both clients, i.e., the PC and Laptop client. The results for these tests for data with quantization and without quantization are shown in Table 5.2.

Rendering performance results of the PC and Laptop client for data without quantization are shown in Table 5.3. This table shows results for two graphics cards: the Intel integrated (HD4000) graphics card and the Nvidia GPU (in parentheses). The rendering times of these two clients for data with quantization are shown in Table 5.4. In both cases, rendering was done using GPU-based sphere ray casting.

Results in Table 5.2 show a significant difference in transfer times from server to client between LAN and WiFi connections. This can be attributed to the difference in bandwidths of the two connections. Having significant differences in the transfer times between LAN and WiFi also suggests that the application is bandwidth-limited. *Transfer time* is taken as the time from when the WebSocket

Table 5.2 — Performance results showing *Transfer times* for the *Laptop* client machine for various molecules in both LAN and WiFi network environments. All transfer times are averages over 100 transfers given in milliseconds (ms). Values in parentheses show results for optimized data layout.

			LAN	WiFi	
Molecule Name	**PDB ID**	**#Atoms**	**Transfer(ms)**	**Transfer(ms)**	**Memory**
peptide simulation	—	100	6.16 (6.83)	7.5 (8.39)	2.8 (1.3) kB
lipase simulation	2VEO (added H_2)	6,626	15.94 (8.05)	45.11 (22.9)	185.5 (86.2) kB
lipase + solvent sim.	2VEO, solvent	38,789	55.89 (12.49)	234.66 (96.31)	1.09 (0.5) MB
mottle virus sim.	—	214,440	234.94 (45.6)	1,212 (407.47)	6.0 (2.8) MB
simian virus	1SVA (× 60)	958,980	1,039 (191.1)	5,598 (2,242)	26.8 (12.5)MB
papillomavirus	3IYJ (× 60)	1,356,840	2,121.9 (284.4)	6,976 (3,179)	38.0 (17.6)MB

Table 5.3 — Rendering performance of the GPU ray casting in frames per second for various molecules using the *PC* and *Laptop* client machines (frame rates given include the data upload to the GPU.). Note that *Rendering* frame rates for the *Laptop* client are given in frames per second (fps) for the Intel integrated (HD4000) graphics as well as for the Nvidia GPU (in parentheses). The largest data set could not be visualized using the Intel GPU. We noticed that, in contrast to the PC client and the Intel GPU, the performance using the Nvidia GPU drops after a time when the visualization is running. This is probably due to thermal limitations of the laptop chassis that causes the GPU to throttle its clock speed.

			PC	Laptop		
Molecule Name	**PDB ID**	**#Atoms**	**LAN**	**LAN**	**WiFi**	**Memory**
peptide simulation	—	100	59.24	59.0 (60.5)	59.0 (60.4)	2.8 kB
lipase simulation	2VEO (added H_2)	6,626	58.99	37.5 (60.3)	37.97 (60.1)	185.5 kB
lipase + solvent sim.	2VEO, solvent	38,789	58.17	15.17 (51.8)	15.33 (59.9)	1.09 MB
mottle virus sim.	—	214,440	44.88	2.52 (10.8)	2.50 (40.0)	6.0 MB
simian virus	1SVA (× 60)	958,980	41.33	1.31 (4.3)	1.31 (15.6)	26.8 MB
papillomavirus	3IYJ (× 60)	1,356,840	38.62	- (3.6)	- (14.9)	38.0 MB

object in the client sent the request to the server up to when the data is entirely received by the client for processing.

Despite the transfer times showing a significance difference, the same is not reflected in the rendering frame rates when using LAN and WiFi connections. This shows that the decoupling of the data fetching thread and the rendering thread works as expected. By running the data fetching thread in a web worker and rendering in the main thread, data transfer times do not stall rendering. The main thread continues to render the already received data while new data is being fetched from the server in the background.

A significant improvement in frame rates was observed when the client is

disconnected from the server and consequently receives no buffer updates. This suggests that the drop in the frame rates as data size increases is attributed to client processing and data upload to the GPU.

In addition to using efficient data encoding format for data transfer in order to save bandwidth, data compression could have been employed to further alleviate the bandwidth problem. However, the additional time required for compression and decompression could outweigh the benefits obtained from reduced transfer time and lower memory footprint.

To find out the effect of quantization, transfer times were measured for data with and without quantization. From the results (see Table 5.2), we note that quantization had a significant impact on transfer times, by reducing the amount of data transferred by almost 50 %. Quantization has an advantage compared to normal compression schemes because the decoding can be done on the GPU, thus significantly reducing decoding time. Even when done on the CPU in JavaScript, the decoding time is minimal.

Despite our promising results, bottlenecks for interactive visualization of dynamic data sets lie not only in the still abstracted memory management in JavaScript but rather in the communication cost between server and client, and between the CPU and GPU. Moreover, extensions in web workers and WebGL to allow data upload to the GPU from Web Workers could prove useful. Current specifications and implementations do not allow uploading data to the GPU or accessing WebGL functionality from a web worker. Although the use of *Transferable objects* partly solves the problem of data transfer between a web worker and the main thread, still this can prove challenging for larger datasets. This can be seen in the drop of rendering frame rates (see Table 5.3) as the data size increases, despite the rendering thread and data fetching being decoupled. If data could be uploaded to the GPU directly from a web worker before rendering, the main thread could be responsible only for user interactions and handling of rendering commands.

As discussed in section 5.1, our application uses client-side rendering. A possible alternative to our client-side rendering would be to transport the images rendered at the server side. This approach, in contrast to our solution, is extremely sensitive to bandwidth fluctuations. One needs to consider the memory footprint even of compressed images in relation to the transfer of the molecule data. For example, a 1280x720 image compressed as JPEG at 50% quality requires 40KB. In order to get smooth interaction, at least 15 of these images need to be sent to the client per second. This would require 600K/s of constant and reliable bandwidth. Although our larger data sets exceed such size, one has to consider the overhead for image compression and decompression as well as the fact that for greater interactivity, we can just pause the requests

Table 5.4 — Rendering performance of the GPU ray casting in frames per second of various molecules using the *PC* and *Laptop* client machines with the optimized data layout (frame rates given include the data upload to the GPU.) Note that the shown *Rendering* frame rates for the *Laptop* client are for the Nvidia GPU.

			PC	Laptop		
Molecule Name	**PDB ID**	**#Atoms**	**LAN**	**LAN**	**WiFi**	**Memory**
peptide simulation	—	100	59.95	59.80	59.84	1.3 kB
lipase simulation	2VEO (added H_2)	6,626	59.95	59.80	59.80	86.2 kB
lipase + solvent sim.	2VEO, solvent	38,789	59.95	59.48	58.07	504.3 kB
mottle virus sim.	—	214,440	44.40	43.02	40.43	2.8 MB
simian virus	1SVA (× 60)	958,980	19.93	20.50	24.82	12.5 MB
papillomavirus	3IYJ (× 60)	1,356,840	7.41	13.32	19.17	17.6 MB

for new data, which directly results in interactive frame rates. For example, papillomavirus rendered up to about 53% faster when requests for new data were paused; such an option is not available for image-based solutions.

Additionally, interaction is sensitive to the total round trip time between request for an image with an updated viewpoint and the resulting rendering. Interactively rotating a data set with significant lag can prove extremely challenging. While this solution only requires a client capable of decompressing the image data, the server needs to be equipped with a dedicated GPU. In cloud deployment scenarios, this is a challenge by itself since there are very few specialized and extremely costly GPUs that allow for virtualization, reducing the number of concurrent cloud instances per chassis drastically.

On the aspect of scalability, a client-side rendering method would scale better than an image-streaming method because it does not create huge demands in bandwidth and requires no server-side rendering infrastructure. In fact, client-side rendering is central to the scalability of the web itself. It would have required massive server infrastructure to host a web page that is accessed by many users if the web page content was rendered on the server.

From the bandwidth requirements for dynamic data given in Table 5.2, we can derive network requirements for our system. For example, in a GBit LAN environment, there are about seven data updates per second for one user or one data update per second for seven concurrent users per server for our largest data set. If the data is static, this scales to seven new users per second, each one receiving the data just once. Afterwards, each client can render the received data interactively, in contrast to an image streaming method that would constantly require transmission of new images from the server.

5.4 Conclusion

By leveraging the GPU, dynamic large molecules with more than one million atoms can be rendered at interactive frame rates. These results demonstrate that it is feasible to visualize large dynamic molecular data in the browser by using WebGL GPU-based ray casting rendering technique combined with modern web technologies in the browser. These client-side technologies make it possible to render large datasets on the client while offloading demanding computations to the server.

Moreover, interactive rendering frame rates require efficient binary data encoding format that minimizes the amount of data transferred, decoupling of rendering and data fetching threads through web workers, and leveraging transferable objects for zero-copy between main thread and web worker. Processing of binary data in the browser is made possible through typed arrays that are part of the JavaScript language and supported by all modern browsers. A web worker ensures that the main thread that is responsible for user interaction and rendering is not stalled because of other long running computations. In order to reduce round trip times to the server, and hence, reduce latency, full-duplex and persistent connection between server and client is required. WebSocket provides such a capability that allows both server and client to exchange data in both directions at the same time.

Therefore, interactive visualization in the browser is achievable by leveraging a combination of modern web standards in the browser combined with GPU-based ray casting techniques for rendering. Our results indicate that, although browser technologies and support of WebGL in modern browsers are still evolving, their potential as a preferred platform for highly interactive collaborative visualization systems is high. These web-based tools would allow scientists at arbitrary locations to concurrently visually analyze the same data and help accelerate the pace of scientific discovery. The application discussed in this chapter can also be employed for remote simulation monitoring on mobile devices. A scientist could connect a visualization tool running in a browser to a simulation executing on a powerful server and monitor its execution in real time. This would also provide immediate feedback compared to the traditional approach where the visualization is run as a post-processing step.

The ability to run the same visualization applications across platforms ranging from desktop PCs to smartphones from a single code base also simplifies software maintenance and upgrades. Once a new version is uploaded to the server, all clients have an immediate access to the new software. Additionally this could also act as a driving force for scientists to share data and tools and encourage collaboration across the globe.

CHAPTER 6

Web-based Visualization of Bricked Volumetric Data with Levels of Detail

Visualization of volume data is important for understanding the internal structure of objects. These data can come from sources like sensors, medical scanners or computer simulations. An example of volume data is medical datasets obtained from MRI or CT scanners. Improvements in volume data acquisition techniques and simulation technologies has led to huge growth in the size of volumetric data. Remote visualization of these data, especially in web browsers, poses a challenge in terms of bandwidth, latency, and computational power at the client side. Although availability of GPU-capabilities in the browser through WebGL allows the use of hardware-accelerated volume ray marching for client-side rendering in the browser, network bandwidth and latency are still the main bottlenecks, especially for large data. The bandwidth problem can be partially remedied using compression. Most of these techniques require that the whole data is transferred to the client at once. However, decompression of large data sets in the client can introduce additional latency. Thus, compression has to be applied with caution [Limper et al., 2013].

Image or video streaming is another popular approach for remote rendering, where the server renders images and sends them to the client. The rendering process can be optimized through adaptive sampling and compression for efficient transfer of individual frames rendered by a server [Frey et al., 2015].

The amount of data transferred to the client in remote rendering approaches can be reduced by employing *volumetric depth images* [Frey et al., 2013]. While this is attractive for clients with limited rendering capabilities, it still requires high bandwidth and low latency in order to ensure interactive visualization. Furthermore, the server has to offer this type of rendering. Consequently, this approach does not scale well for higher numbers of users.

Multi-resolution rendering and bricking techniques for visualizing large data sets have been introduced on desktops and workstations [Engel et al., 2006; Beyer et al., 2008, 2015]. For example, multi-resolution techniques have been employed to reduce interaction latency by allowing the user to view a low-resolution model during interaction and render a high-resolution model when there is no user interaction [Engel et al., 2006]. Bricking techniques have mainly been used for addressing the problem of visualizing large volume data by allowing individual bricks to be streamed to the GPU from memory or local disk [Beyer et al., 2015]. However, these techniques have not been fully exploited in the browser.

Increasing capabilities in web technologies and availability of volume data on the web, necessitates exploration of techniques for visualization of large volume data on the web. Web-based techniques are attractive due to the ubiquity of the browser across devices and platforms. Moreover, web-based techniques can easily support collaborative visualization for teams that are geographically distributed across the globe. Visualization of large volumetric data requires a combination of techniques that allow efficient data transfer between client and server and minimal processing on the client. Additionally, latency needs to be minimized in order to ensure interactive visualization.

As discussed in chapter 3, current streaming and progressive data transfer techniques in the browser [Ponchio and Dellepiane, 2015; Lavoué et al., 2014; Sutter et al., 2014; Limper et al., 2014] have focused mainly on mesh data but have not considered volume data.

Previous work on GPU-based volume rendering using WebGL has mainly focused on techniques that allow visualization of volumetric data without the use of 3D textures, since these techniques have been implemented using WebGL 1.0, which has no support for 3D textures. However, WebGL 2.0 supports 3D textures. Congote et al. [2011] implemented a GPU-based volume ray marching in WebGL to visualize medical volumes and weather radar volumetric data sets at interactive frame rates. Due to lack of support for 3D textures in WebGL 1.0, volumetric data is stored using a 2D texture atlas. Noguera and Jimenéz [2012] also address the lack of 3D textures in WebGL 1.0 and in mobile devices supporting OpenGL ES 2.0 by using 2D texture mosaics and combine it with multi-texture support for storing large volume data by utilizing

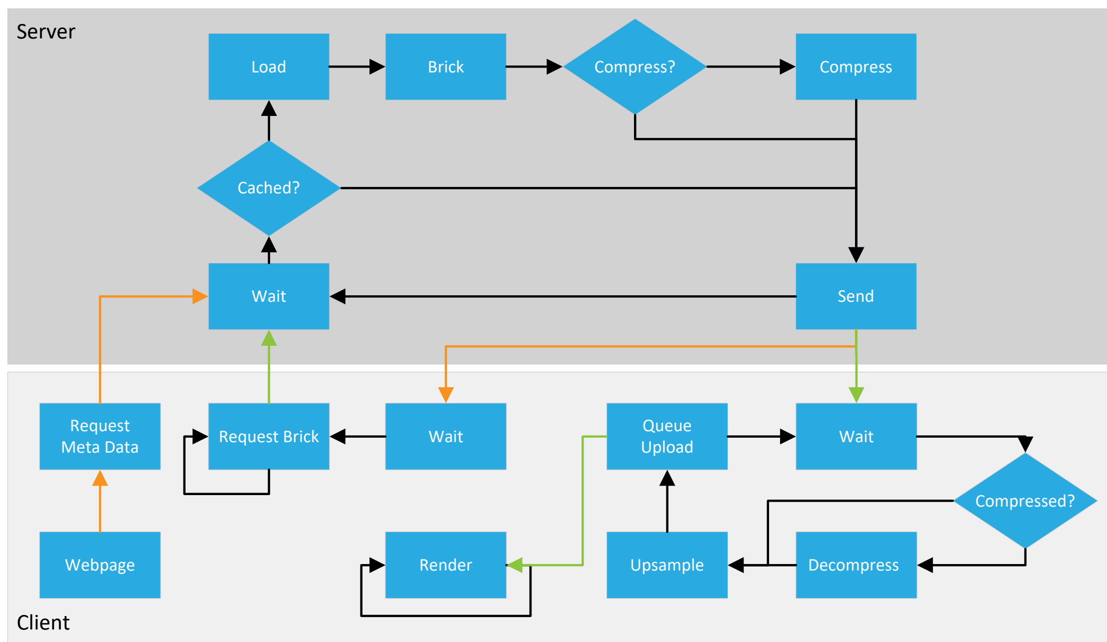


Figure 6.1 — Algorithmic pipeline of our client - server architecture for the bricked volume rendering.

all available texture units on the GPU. Movania and Feng [2012] implemented a single-pass GPU-based ray caster in WebGL for visualization of medical data sets. Yang et al. [2015] presented a specialized compression technique for time-varying volumetric data that combines S3TC texture compression with *deflate* compression for efficient transmission of volumetric data to the browser. On the client side, the compressed data is inflated and uploaded to the GPU as video textures, which are directly supported by WebGL. None of these previous approaches uses multi-resolution volumes or bricking to address network latency and bandwidth issues.

This chapter discusses a GPU-based volume ray marching technique that uses a progressive data transfer utilizing multiple levels of detail and bricking for interactive volume rendering of remote data sets. The technique combines multi-resolution volumes and bricking to address the problem of interaction latency and efficient data transfer in a network environment using WebGL 2.0. The results discussed in this chapter were published in a paper at the VMV conference [Mwalongo et al., 2018]

6.1 Algorithmic Pipeline

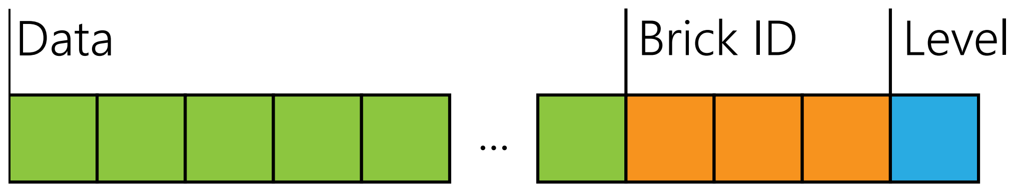
The algorithmic pipeline follows a client-server architecture. The server is responsible for storing the volumetric data, generating the bricks of the different levels of detail, and encoding the brick data before transferring them to the client. The client is responsible for decoding the brick data and rendering. An overview of the pipeline is shown in Figure 6.1.

The application allows the user to visualize volumetric data stored on the server by providing a browser-based interface. First, a hierarchical multi-resolution volume is computed on the server, where each level has half the resolution of the previous level. Each level of detail is divided into uniformly sized bricks, leading to short per-brick transfer times even for low network bandwidth. By using a bricked multi-resolution volume representation, the user can start viewing the lowest-resolution volume while the volume is being progressively refined using higher-resolution bricks, which are asynchronously streamed from the server in a background thread.

The interaction begins with the client requesting meta data about the volume to be visualized via a web page. The server, upon receiving this request, first checks whether the data set is already cached. If not, the data set is loaded, a MIP-map-like level of detail pyramid is generated, and each level is bricked into equally sized subvolumes. Optionally, the bricks are compressed to reduce the memory footprint. Afterwards—or if the data was already loaded—the volume meta data is sent to the client. These meta data include the original volume resolution, the brick resolution, and the number of levels of detail.

The client can now use this information to allocate texture memory for the whole volume. It also uses the meta data to generate requests for the bricks from the server. The initially allocated texture is updated with the brick data as it is received from the server. Receiving and decoding the brick data is done asynchronously in a separate thread to avoid stalling the main thread.

The volume texture is rendered continuously as soon as the first brick is received, processed, and uploaded. That is, the progressive refinement of the volume is visible as soon as it is available on the GPU. Since the volume texture always has the full resolution, the bricked structure of the data does not affect the rendering.



Data: $(brickRes_x \times brickRes_y \times brickRes_z)$ unsigned byte
Brick ID: 3 unsigned bytes (id_x, id_y, id_z)
Level: 1 unsigned byte

Figure 6.2 — Memory layout for brick data serialization between server and client.

6.2 Implementation

The server side implementation uses *Node.js* runtime[1] and the client side uses JavaScript and WebGL 2.0. Using the same language in both client and server simplified the prototype implementation. However, for efficiency reasons, the server part can be implemented in any high-performance language like C++.

After receiving the data from the server, the client decodes it using the *Dataview* object in order to extract the volume header and the actual volume data from the serialized format shown in Figure 6.2. The extracted data is used to create a 3D volume texture that is uploaded to the GPU for rendering. Volume rendering is initialized by first rendering a unit cube with back face culling enabled. The vertex shader performs modelview and projection transformations while the volume ray marching is done in the fragment shader.

6.2.1 Server-side Brick Generation

The brick generation is performed on the fly when any data of a particular volume data set is requested for the first time by a client. After loading the volume data set, the first step is to compute how many levels of detail have to be generated. Similar to a classical 2D texture MIP map, each level has half the resolution in each dimension as the previous level until the lowest level is reached, which consists of just one brick. The voxel values of the lower levels are computed by averaging the corresponding 2×2×2 voxel values of the previous level. Each level is divided into fixed-size cubic bricks. We typically use a brick resolution of 32×32×32 or 64×64×64 voxels, which results in a reasonably small memory footprint per brick but also does not lead to too many HTTP requests by the client for the entirety of the bricks. Once the volume is loaded from disk

[1] https://nodejs.org/ (last accessed 04/15/2018).

and bricked, it is cached in memory and subsequent requests will directly get the data of a particular brick.

The additional memory for storing the lower-resolution bricks is very low. Since the resolution is divided by a factor of two in each dimension, the total memory can be computed as $\Sigma_{i=0}^{\infty} \left(1/8\right)^i = 1 + 1/7$. That is, the additional memory for all lower levels equals $1/7$ of the memory needed for the full volume.

The bricked volume data could also be precomputed and stored to disk to eliminate the processing time when loading a new volume data set. However, since the cost for preprocessing is only incurred once as explained above, precomputing the bricks and storing them to disk was not considered for implementation in the prototype.

6.2.2 Data Encoding and Transfer

The data format for sending the brick data from the server to the client is shown in Figure 6.2. The serialized message starts with the actual data of the brick, followed by the brick indices—i.e., the (x,y,z)-coordinates of the brick in the current level—and the level of detail to which the requested brick belongs. Since the bricks always have the same resolution, independent of the current level, the messages always have the same size. All values are stored as byte values (UINT8), that is, the current implementation can address 16k×16k×16k volumes if 64×64×64 voxel bricks are used. The information about the brick indices and level are needed since the receiving thread in the client cannot get this information from the main thread. That way, incoming brick data can be processed without stalling the rendering thread.

In order to keep the data that has to be transferred from the server to the client as small as possible, we optionally use compression. The above mentioned serialized brick data is compressed using the Snappy.JS library[2]. If compression is used, the brick size is set to 64×64×64 voxels by default. Otherwise, the amount of data to be transferred for each brick request would be very low.

6.2.3 Client-side Data Processing

After the client has requested a new data set and it was loaded and bricked by the server, the client receives meta data about the data set (i.e., the voxel resolution of the data set, the voxel resolution of the bricks, the number of levels etc.). The client uses this information to allocate the required 3D texture storage, which has the same resolution as the original volume data. We use immutable textures, which are created using `texStorage3D()`, since these are

[2] Snappy.JS `https://github.com/zhipeng-jia/snappyjs` (last accessed 04/14/2018).

more efficient than mutable textures as per WebGL 2.0 specification [Khronos, 2013]. After this, the volume bricks can be requested.

Each time a brick is received from the server, the corresponding texture memory has to be updated with the new brick data. If the data was compressed for transfer, it is of course first decompressed again. Since each brick has a different footprint in the full volume depending on which level of the volume it is in, the brick data has to be upsampled to the resolution of the original volume before being sent to the GPU. During the upsampling, the information about the level of the brick and the brick index is used to compute the corresponding global voxel offsets in the full-resolution 3D texture. The data is upsampled by duplicating the voxels by a factor that is computed as the ratio between the resolution of the original volume data and the resolution of the volume at the corresponding level of detail of the brick. Receiving and upsampling each brick are performed by a web worker [W3C, 2015]. The transfer of data between this worker and the main thread is done using transferable objects to avoid copying of the data. This has already been discussed in chapter 5 for particle rendering. Finally, the upsampled brick data is used to update the volume texture using `texSubImage3D()`.

6.2.4 Prioritization of Volume Bricks

The initial bricking described in section 6.2.1 allows the client to request the data level by level, starting with the lowest resolution (just one brick) to get an initial image very fast and adding higher-resolution data level by level until the full resolution is reached. However, volume data often has regions that are more important than others. That is, it can be worthwhile to transfer the bricks out of order, so that more important bricks are available earlier in the client than less important ones. Thus, important regions in the data set are available for rendering sooner in full resolution.

Since it is hard to find a measure of importance that applies to all kinds of data sets, we decided to use entropy as a measure of importance in our prototype. That is, we compute the entropy of each brick and assign it as the importance value of that brick. The reasoning behind this is that bricks with a higher disorder and more diverse voxel values contain more information. Our algorithmic pipeline would of course also work with other, arbitrary measures of importance.

After assigning an importance value to each brick, a list consisting of pairs of importance values and the corresponding brick indices—that is, the level and (x,y,z)-coordinate of each brick—is generated. This list is sorted by the importance values in ascending order. The client can now request this sorted list and use it to prioritize the bricks. That is, the client can request the bricks

in the order of their importance. The only exception from this rule is the lowest-resolution level consisting of just one brick. Since this brick should always be transferred first to the client to give a first impression of the whole data set, it is assigned the maximum importance value.

Note that the scheme described above can lead to bricks being requested out of level order, that is, a brick from a lower level (with higher resolution) can be received earlier than the corresponding brick with higher level (lower-resolution), which overlaps with the first brick. In this case, it has to be ensured that the lower-resolution brick does not overwrite the higher-resolution brick (see Figure 6.3). We handle these cases by storing all bricks in an octree-like data structure. If a brick is loaded and upsampled, we have to check recursively whether one or more of its eight child nodes already contain brick data. If this is the case, the respective areas are updated with the available high-resolution bricks prior to uploading the volume to the GPU. To improve the performance of this algorithm, we propagate the availability of high-resolution bricks to lower levels, so we do not have to traverse the whole tree. The importance-based prioritization of brick requests is of course only optional.

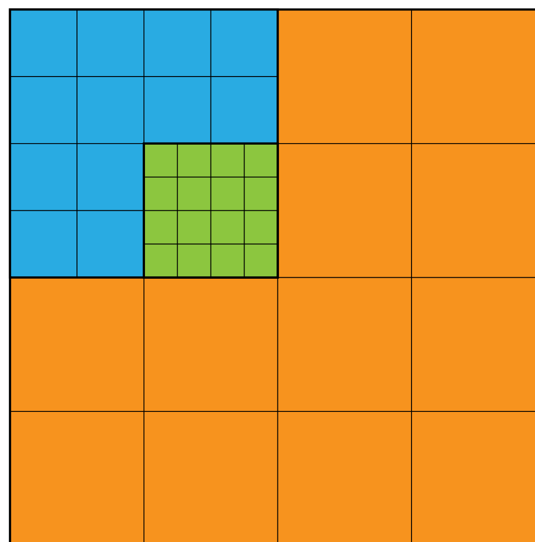


Figure 6.3 — Effects of importance-based prioritization of bricks. The lowest-resolution brick (orange) always has the highest importance value and is requested first. If the green brick has a higher importance than the blue one, it is requested earlier. Since the blue brick is at a higher level and overlaps the data of the already available green brick, the corresponding voxels in the 3D texture used for rendering must not be overwritten by the values of the blue brick. Note that the orange and blue voxels are upsampled to the original resolution of the volume (green brick).

6.2.5 Client-side Rendering using WebGL

As mentioned at the beginning of this chapter, WebGL 2.0 [Khronos, 2013] supports 3D textures and complex fragment shaders that allow for dynamic loops. For the rendering, we implemented a basic 3D texture volume ray marching [Engel et al., 2006] in WebGL. Only the front faces of the bounding box of the volume are rendered in order to initiate the volume rendering. The actual ray marching is performed in the fragment shader.

The user can switch between maximum intensity projection or a 1D transfer function. All transfer functions used to generate images for this chapter were constructed using the Inviwo visualization framework [Sundén et al., 2015] and stored locally on the client as *PNG* images. We use a `FileReader` object of the File API [W3C, 2013] to read the transfer function data from the local disk and upload it to the GPU as a texture.

As observable in Figure 6.4, the aggregation of densities over resolution levels cannot be properly performed via simple averaging. This has been investigated in detail by Sicat et al. [Sicat et al., 2014]. They proposed to encode high-frequency data in the lower resolutions via probability distribution functions to obtain a scale-consistent rendering. Since this severely impacts the memory requirements, the volume is represented via a Gaussian mixture model (GMM) instead. The downside of this method is that the GMM fitting is extremely costly and the rendering requires a density histogram instead of discrete density values per voxel. Since the refinement of the volume converges rather quickly using our bricked data transfer, we decided to use the straightforward averaging in our prototype. Actually, the artifacts arising from lower-level data make it easy for the user to spot locations where full-resolution data is still missing.

6.3 Results and Discussion

We measured transfer with and without decompression times on multiple machines and various data sets. To account for different network connections, we employed a workstation (custom built), a laptop (Microsoft Surface Book), and a typical smartphone (Samsung Galaxy S7). We used the hazelnuts data set (512×512×512 voxels, 128 MB; shown in Figure 6.5) for this test and varied the size of the transferred bricks from 512^3 (one brick) down to 64^3 (585 bricks over all levels of detail). The results and the hardware specifications of our test systems can be seen in Figure 6.6. Several interesting effects can be observed in the resulting graphs. One, for local wired network, compression does not improve the overall time. Although the Snappy compression results in roughly half the data set sizes, the time required for decompression roughly equals the time saved from shorter transmission. However, the improvements for

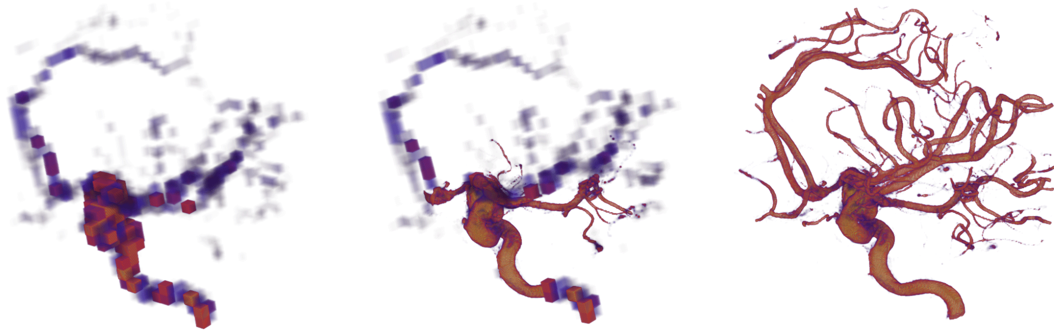


Figure 6.4 — Different snapshots of the bricked volume rendering method showing the aneurism data set at different levels of detail. We use multiple levels of detail of the volumetric data, which are divided into bricks and progressively transferred to the client for rendering. The leftmost image shows the lowest resolution while the rightmost image shows the final image with the original resolution of the data set. The volume ray marching was implemented in WebGL 2.0 and uses a simple 1D transfer function.

compressed data are significant for slower network speeds (WiFi and LTE). The overall time for data transmission is reduced to less than 50% in all cases and decompression times are negligible compared to transfer times. As expected, the additional data needed for the levels of detail is so low that it does not affect total transfer times significantly. That is, the total transfer time does not depend on the chosen bricking for the two wireless connections. The variance of the measurements in these cases can be attributed to constantly changing connection quality, which we also observed in the connection properties. This is due to taking the measurements in a real-world office environment. For example, in the WiFi case, we observed speeds between 39 Mbit/s and 117 Mbit/s (average was 72 Mbit/s). It is interesting to notice, however, that for the wired connection the single brick and the smallest bricks behave differently. The slightly higher transfer time for the single brick can be attributed to the fact that only a single transfer is performed, while all other cases queue and execute several HTTP requests that will be served concurrently. This means that we could still improve the transfer of a whole volume even without levels of detail by bricking it all the same. We hypothesize that the sudden increase of transfer time for the smallest brick must stem from some inherent overhead in the HTTP requests that cannot be mitigated by the low latency of wired networking.

Since the main target of our method are networked connections with lower speed, not only the total transfer time, but also the per-brick transfer times are important. These directly influence the time a user has to wait until the visualization is available as well as the time it takes for each refinement of the available data. As mentioned above, the total transfer time does not vary

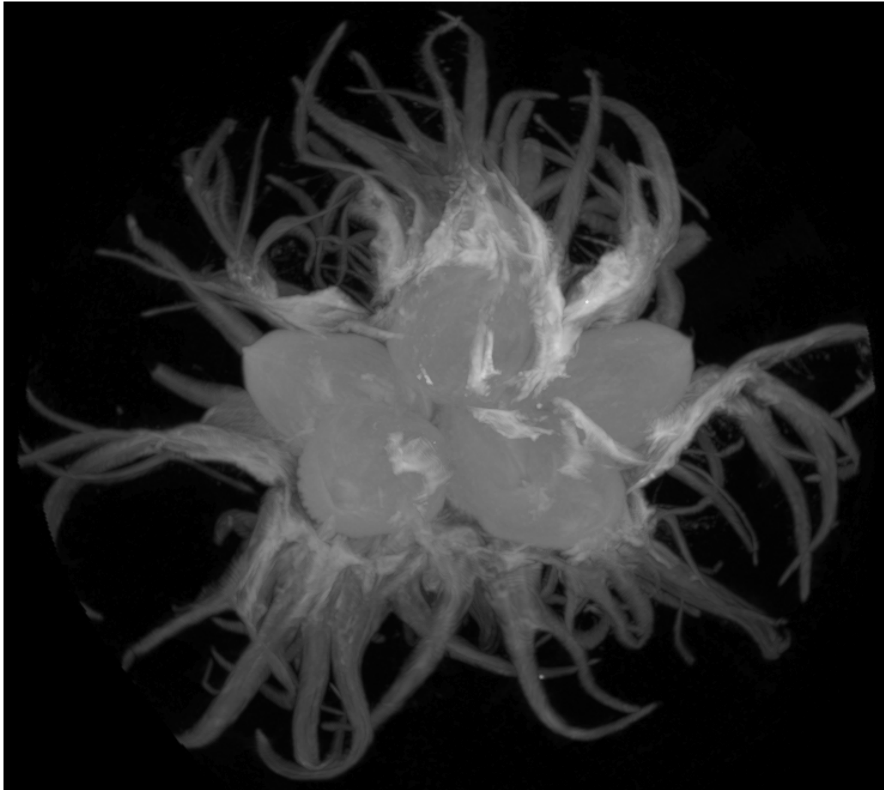


Figure 6.5 — Visualization of an hazelnuts volume data set using maximum intensity projection.

much over brick sizes, so the trade-off for having a quick response with lower-resolution data is more than justified. Especially on a smartphone, where the absolute transfer times are very high, quick response times are essential. Based on our measurements, we therefore recommend the smallest brick size (64^3 voxels, compressed), which took about 20ms to transfer, in this case. For the WiFi connection, a brick size of 128^3 can be chosen to obtain comparable results (19ms).

The rendering performance for the different client systems is shown in Figure 6.7. Note that we use a logarithmic scale for the frames per second (FPS) values. In addition to the hazelnuts, we also tested the engine data (256×256×128 voxels, 8 MB; see Figure 6.8) and the aneurism data (256^3 voxels, 16 MB; see Figure 6.4). For the frame rate measurements, we zoomed into the volume data sets until they reached maximum screen coverage while still being fully visible. Performance was measured after data set transmission was completed, although the background transfer and texture updates did not affect performance noticeably. The overall behavior of the different hardware is on par with expectations: while our simple volume ray marching reaches very high frame rates on the current

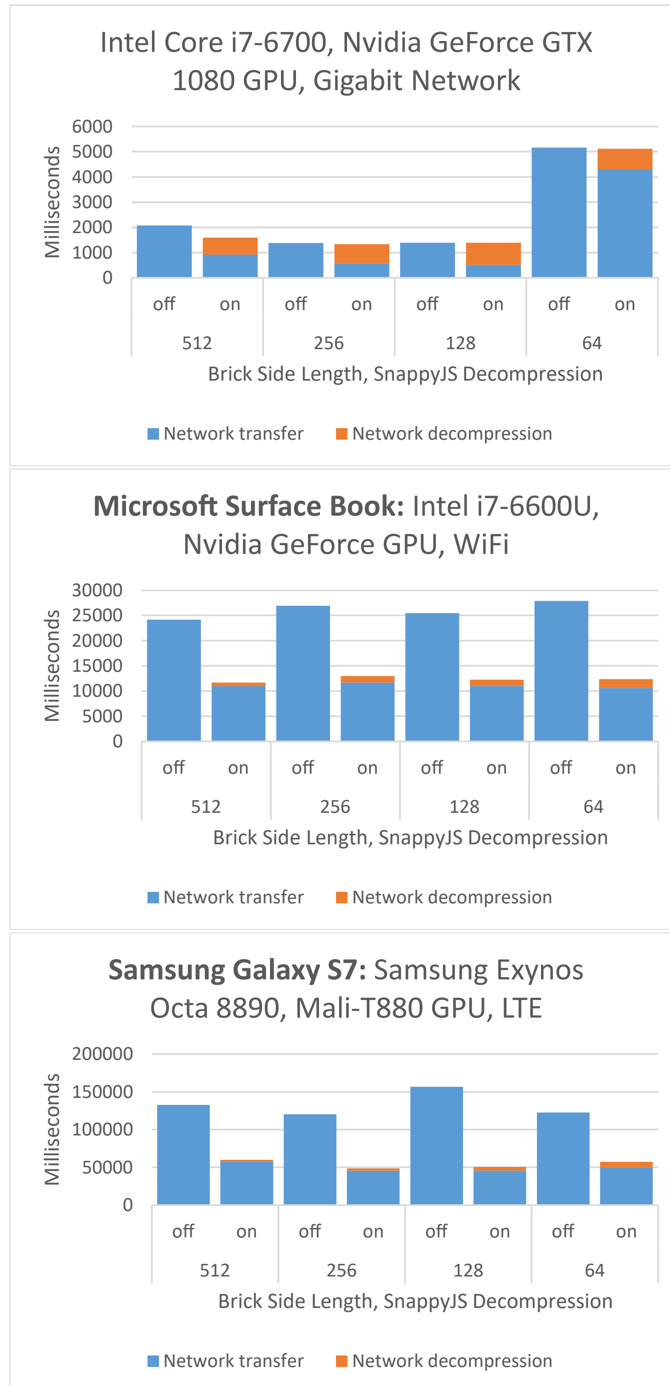


Figure 6.6 — Transfer and decompression times for the Hazelnut data set. The machines used for testing are the same as in Figure 6.7 and off means no compression.

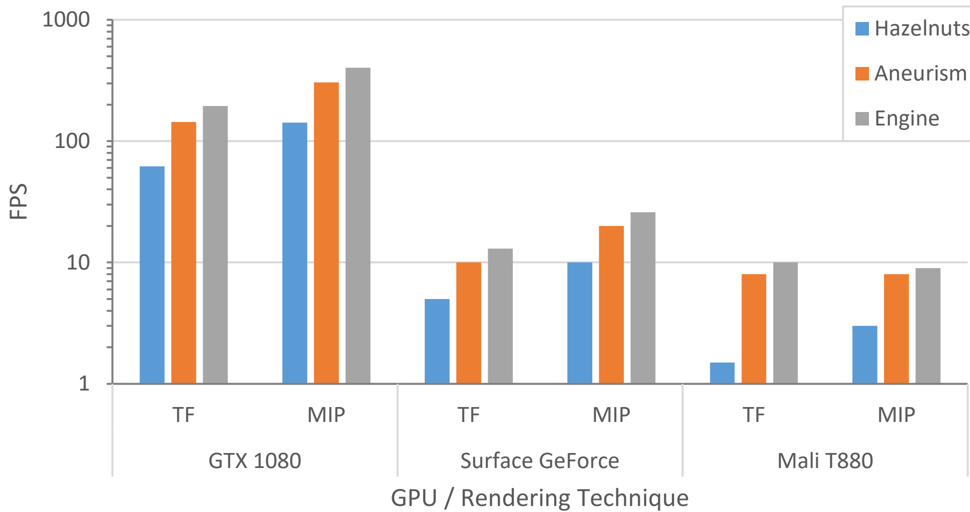


Figure 6.7 — Performance of the WebGL volume renderer for different data sets and rendering techniques: Maximum Intensity Projection (MIP) and volume ray marching using a Transfer Function (TF). A canvas size of 1024×768 was used. Frame rates were measured using the *stats.js* library (`https://github.com/mrdoob/stats.js/`).

desktop hardware, smaller (and lower-power) devices still struggle with this visualization technique. Especially the smartphone does not reach interactive frame rates for the largest data set. On both PC systems, MIP is twice as fast as using a transfer function. Interestingly, the performance of the smartphone does not change significantly if using a transfer function. This means that the dependent texture lookups have a much lower cost on the Mali architecture. The hazelnuts data set has such low performance that the performance difference is beyond measurement accuracy (1 vs. 2 FPS).

The prioritization scheme described in section 6.2.4 works well in conjunction with the transfer of levels of detail. Figure 6.4 shows the effect of the entropy-based prioritization of brick requests. In the intermediate snapshots, the central region of the data set is available in full resolution very early, since these bricks have the highest importance. As observable in the image, entropy is a very effective measure of importance for this data set, since the aneurism is available in high resolution first. Depending on the transfer function, this is not always the case. If it is known beforehand, the server could compute the entropy with respect to this classification. However, this is not applicable in an exploratory application case, where the user interactively designs the transfer function.

While the aneurism example shows that the entropy-based importance is an acceptable generic approach, more specific importance measures can be devised

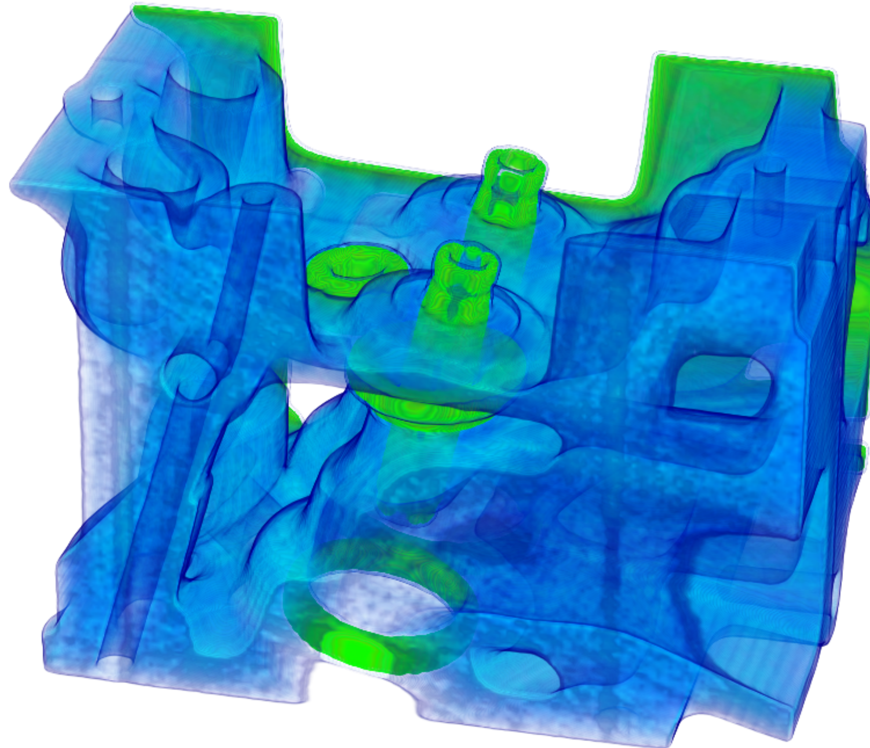


Figure 6.8 — Visualization of the engine volume data set using a transfer function.

depending on the data sets and application domain. For known data sets, expert annotations could be used to derive importance. An alternative could be to use machine-learning-based classification, which is already routinely used in medical imaging [Litjens et al., 2017], to identify important spots in the data. It would also be possible to use a weighted sum of different importance measures and allow an expert user to adjust the weighting in accordance with the task at hand.

6.4 Summary and Conclusion

Minimizing latency and efficient use of network bandwidth is important for interactive remote visualization in the browser. The work discussed in this chapter has demonstrated the feasibility of visualizing volumetric data sets at interactive frame rates using multiple levels of detail and bricking techniques in the browser to address the problem of interaction latency and efficient use of bandwidth. Combining levels of detail, bricking, and importance-based streaming of compressed bricks to the client provides a base for visualization of large volumetric data that is only limited by texture memory of the GPU.

Figure 6.9 — Visualization of the flower data set with 1024^3 voxels using a transfer function. Our unoptimized WebGL 2.0 volume renderer reaches ~2 fps on a Nvidia GeForce GTX 1080, which would be the expected frame rate for a similar desktop volume renderer.

By transferring individual bricks starting with the lowest level of resolution, this technique allows the user to start interacting with the volume without waiting for the entire volume to be transferred to the client. Moreover, this minimizes latency and network bandwidth requirements. Although the use of compression can improve transfer times, especially for slow networks as discussed in section 6.3, its costs in terms of compression and decompression time should be carefully weighed. Since the main focus for this work was latency and bandwidth issues, acceleration strategies like early ray termination and empty space skipping were not implemented. However, these techniques could improve the rendering of large datasets like the flower in Figure 6.9. This is left as future work.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

The work presented in this thesis has demonstrated the feasibility of interactive web-based visualization of large scientific data sets including both static and dynamic data without the use of plugins. This has been achieved through exploiting modern web technologies like HTML5 and WebGL that enable implementation of GPU-accelerated rendering techniques that were until recently only confined to desktop platforms.

Interactive rendering in the browser requires efficient data encoding and network data transfer techniques in order to minimize bandwidth usage and decoding time on the client side. Data encoding formats that exploit typed arrays and are GPU-friendly and allow the data to be uploaded to the GPU with minimal processing on the CPU side.

The results have shown that GPU-based ray casting rendering techniques are key to achieving interactive visualization in the browser. By generating geometry data on the GPU from implicitly defined surfaces, much work is pushed to the hardware thus avoiding doing heavy computations on the CPU that would affect performance. This also allows less data to be uploaded to the GPU as only the parameters of the implicit surface are transferred from the server to the client and down to the GPU. This saving in bandwidth and avoiding doing computations in the CPU makes rendering large data sets feasible. This is especially critical for dynamic data that can potentially change in every frame. Keeping computations on the CPU side to the minimum is important because

JavaScript is usually slow compared to compiled languages like C++.

Although heavy computations can be offloaded to a background process through web workers, this helps only to keep the user interface responsive but data upload remains a challenge because it has to be done on the main thread that handles user interactions and rendering. This is challenging given the single-threaded nature of JavaScript.

Another advantage of GPU-based ray casting rendering is high image quality. Since ray-object intersections are evaluated per pixel, the produced images are accurate (at the pixel level) and mathematically correct because the surface points are sampled from a mathematical function definition of the surface. To get similar image quality using polygon rendering requires huge number of triangles to approximate smooth surfaces. This would in turn increase storage and bandwidth requirements and affect rendering rates.

The problem of data upload to the GPU is critical for rendering dynamic data. Rendering performance could be improved by exploiting web workers in the browser for data upload to the GPU. Unfortunately, the WebGL standard does not allow access of the WebGL context in the web worker; making it impossible to do any WebGL related task on the web worker. This limitation makes it difficult to decouple rendering and data upload tasks.

As discussed in chapter 5, data upload takes significant processing time and affects rendering rates for dynamic data where new data has to be uploaded to the GPU whenever there are data changes. Decoupling the rendering and data upload tasks could allow one of these tasks to be performed in the web worker. For example, if data could be uploaded to the GPU from a web worker and rendering done on the main thread, the main thread could continue rendering the old data and be notified to use the new data once the upload is complete. Current efforts on standardization of *Off screen canvas* [WHATWG, 2018] could help remove this limitation.

Efficient data encoding techniques are important for efficient use of network and CPU-GPU bandwidth. Traditional solutions for bandwidth problems are through compression. However, this approach may not always work well in the browser because of the decompression time required to decompress the data. Moreover, when decompression is done on the CPU, the CPU-GPU bandwidth remains a bottleneck for rendering performance. Therefore, data encoding techniques that require minimal processing on the client become important. For example, as discussed in chapter 5, use of array buffers and quantization techniques allows for efficient use of bandwidth and minimum processing requirements on the CPU.

Hiding latency is another factor that is important in order to achieve interactive rendering in the browser. Network latency is in large part addressed by ren-

dering on the local client rather than on the server. This is important because it avoids round trip network latency for any rendering parameter changes. However, data fetching and data updates still have to be communicated through the network. Additionally, communication between data fetching and rendering threads need to be asynchronous in order to avoid one thread stalling the other.

Latency can also be minimized using bricked multi-resolution volume data represention combined with progressive importance-based data transfer as discussed in chapter 6. This allows the user to have the glimpse of the visualization without waiting for the entire volume data to be fully loaded. Again as discussed in chapter 5 and chapter 6, asychronous data fetching using web workers plays an important role in client-side rendering in the browser because it avoids stalling the main thread responsible for rendering.

The HTTP protocol is a request-response protocol. The server replies only in response to a client initiated request. Once all the resources associated with a particular request have been sent to the client, the connection is closed. This causes a new connection to be established for every new request to the server. However, establishing a new connection for every request and one-way communication between client and server introduces communication latency and inefficient use of network resources. WebSocket addresses this inefficiency by providing a bidirectional persistent communication channel between client and server. Similar to HTTP, the request to establish a communication channel is initiated by the client as a normal HTTP request but with header fields—*Upgrade* whose value is set to "websocket" and *Connection* whose value is set to "Upgrade". This indicates to the server that the client wants to upgrade the HTTP connection to a web socket connection. Once the connection is established the client and server can exchange data at any time.

This bidirectional persistent communication is important for visualization of dynamic data that require constant data updates because it reduces latency by avoiding the cost of establishing a new connection for every data update. Moreover, the server can send the data to the client without the client requesting it first.

Another important technique for hiding network latency is to overlap data fetching with rendering. This is achieved by fetching data asynchronously in the background using a web worker so that rendering is not stalled by data fetches. The rendering thread continues to render the already received data and then gets updated when new data arrives. Since web workers and main thread do not share memory and communicate only through message passing, using typed arrays to encode data is also important because this allows data exchange with zero copy through *transferable objects* technique.

7.2 Outlook

The browser has great potential to become a preferred deployment platform for visualization tools and services. Its ubiquity across operating systems and devices, from smartphones to desktops is attractive for the increasingly mobile and collaborative research work. Combing this with the advances in networking, cloud computing, and mobile GPU technologies, it is more likely that future visualization tools will be deployed in the cloud and consumed on various devices through the browser. The cloud would become a shared virtual research space where domain scientists and visualization researchers can share their data and visualization tools to gain insight and accelerate scientific discovery. As discussed in chapter 3, many visualization tools are already on the web nowadays providing a basis for cloud-based visualization.

7.2.1 Visualization as a Cloud Service

The increasingly growing data sets due to the proliferation of digital devices producing massive data sets have created a demand for cloud storage and computational resources. A single machine is no longer sufficient for storage and computational needs of these data sets. Since the data is already stored in the cloud, it becomes natural to store the tools that process the data in the cloud too. Moreover, a cloud service can also provide the necessary compute power for users with low-end devices like tablets and smartphones.

The traditional approach to visualization as a post-processing step is becoming impractical because of the inability to download the entire data from the cloud and visualize it on the client. Therefore, a better approach would be to host the visualization tools as services in the cloud and use client-side GPU-based rendering for visualizing the data to support the analysis task at hand using data streaming techniques. Since not all generated data may be useful, visualization tools can be combined with analytics engines that can detect interesting patterns in the data and send to the client only the interesting data for rendering. This can be achieved through a data base management system that integrates data management functionality with data analysis tools like SciDB [Stonebraker et al., 2013].

Another important aspect is the need for visualizing data generated in real-time. This would require a visualization service that is continuously running and reacting to constant data updates in a streaming computational model. The data are visualized on the fly as they are being generated because storing them first would require huge storage. Moreover, for data that are produced in real-time, visualizing them immediately is important for real-time decision making. In cases of emergency situations where visualization can play an important role by

helping rescue teams to understand an evolving situation, making fast decisions can save lives. This could be helpful, for example, in flooding or earthquake situations. A simulation model coupled with real data from affected areas can feed data to the visualization service that can help to understand the situation better for proper predictions and rescue planning and management.

The web service approach to building software has been introduced to address the problem of software interoperability. RESTful web services have gained attention and popularity due to their simplicity and well integration with other web technologies. By building on top of existing web infrastructure, RESTful web services benefit from reusing the existing infrastructure for better performance and scalability. For example, the GET method of the HTTP protocol is a read only operation and is highly optimized for performance through different levels of caching from in browser caching to regional level caching servers.

Exploiting web services for visualization tools would allow seamless integration between visualization and data analysis tools through standardized interfaces. In the era of big data, visualization tools alone may not be sufficient to help make sense of the data; a combination of tools is required.

7.2.2 Collaborative Web-based Visualization

The increasing trend towards geographically distributed and collaborative research teams, requires visualization tools that can support this kind of collaborative work. Cloud computing combined with web technologies provide an ideal environment for deployment of such visualization tools. Current technological trends especially on mobile GPUs and web technologies suggest that this trend will continue. At the same time massive data will continue to be produced at increasing speeds in real-time from simulations, sensors and other digital instruments. In order to gain insights from these data, teams of scientists, who may be geographically distributed, would benefit from visualization tools that can allow them to work together in real-time while exploring the same data.

For example, a global team of health experts, could use a geospatial collaborative visualization tool to study and share their insights about the spread of a disease outbreak in real-time. This would allow them to make decisions faster than they would if each person was working in isolation. Another collaboration that would benefit from these tools is that of domain scientists and visualization researchers. This would allow developers of visualization tools and users of these tools to work together in close collaboration and help to accelerate scientific discovery. Collaborative web-based visualizations would be ideal

in these scenarios because all these users may be using different devices and running different operating systems.

The work presented in this thesis can serve as a basis for further research to tackle these demanding challenges. Interactive visualization of large, real-time data is a challenge of scale in computation, storage, and communication. Scalable web-based visualization algorithms in all these three aspects will be crucial to handle these challenges effectively.

Bibliography

J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In , editor, *EG 1987-Technical Papers*, pages 3–10. Eurographics Association, 1987. 65

Amazon. Amazon Web Services (AWS)- Cloud Computing Services, 2006. [Online]. Available: https://aws.amazon.com/, (last accessed 2018/08/31). 34

Amazon. Amazon S3, 2010. [Online]. Available: https://aws.amazon.com/s3/, (last accessed 2018/08/31). 29

Analytical Graphics. Cesium - WebGL Virtual Globe and Map Engine, 2012. [Online]. Available: https://cesiumjs.org/, (last accessed 2016/04/15). 54

K. Andrews and B. Wright. FluidDiagrams: Web-Based Information Visualisation using JavaScript and WebGL. In N. Elmqvist, M. Hlawitschka, and J. Kennedy, editors, *EuroVis - Short Papers*. The Eurographics Association, 2014. 50, 52

V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. How to adapt applications for the Cloud environment. *Computing*, 95(6):493–535, 2013. 57

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010. 33

BabylonJS. BabylonJS - 3D engine based on WebGL/Web Audio and JavaScript, 2013. [Online]. Available: http://www.babylonjs.com/, (last accessed 2018/08/31). 19, 44

S. K. Badam and N. Elmqvist. PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages 109–118, New York, NY, USA, 2014. ACM. 54

J. Behr, Y. Jung, J. Keil, T. Drevensek, M. Zoellner, P. Eschler, and D. Fellner. A Scalable Architecture for the HTML5/X3D Integration Model X3DOM. In *Proceedings of the 15th International Conference on Web 3D Technology*, Web3D '10, pages 185–194, New York, NY, USA, 2010. ACM. 19, 20

H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 2000. 3, 10, 46, 65, 66, 83

J. Beyer, M. Hadwiger, T. Möller, and L. Fritz. Smooth Mixed-resolution GPU Volume Rendering. In *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'08, pages 163–170, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. 90

J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum*, 34(8):13–37, 2015. 90

M. Biasin. pv - WebGL protein viewer, 2013. [Online]. Available: https://github.com/biasmv/pv, (last accessed 2016/01/27). 46

M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *IEEE Trans. Vis. Comput. Graphics*, 17(12):2301–2309, 2011. 19, 44, 52

K. Brodlie, J. Brooke, M. Chen, D. Chisnall, A. Fewings, C. Hughes, N. W. John, M. W. Jones, M. Riding, and N. Roard. Visual Supercomputing: Technologies, Applications and Challenges. *Computer Graphics Forum*, 24(2):217–245, 2005. 26

D. Brutzman and L. Daly. *X3D: Extensible 3D Graphics for Web Authors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. 48

M. Callieri, R. M. Andrei, M. Di Benedetto, M. Zoppè, and R. Scopigno. Visualization Methods for Molecular Studies on the Web Platform. In *Proceedings of the 15th International Conference on Web 3D Technology*, Web3D '10, pages 117–126, New York, NY, USA, 2010. ACM. 46

N. Carlini, A. P. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 97–111, Bellevue, WA, 2012. USENIX. 2

J. Chandler, H. Obermaier, and K. I. Joy. WebGL-Enabled Remote Visualization of Smoothed Particle Hydrodynamics Simulations. In E. Bertini, J. Kennedy, and E. Puppo, editors, *Eurographics Conference on Visualization (EuroVis) - Short Papers*. The Eurographics Association, 2015. 47

S. M. Charters, N. S. Holliman, and M. Munro. Visualization on the grid: A Web Service Approach. In *Proceedings UK eScience third All-Hands Meeting*, pages 202–209, 2004. 27, 30, 31

K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. 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. 36, 56

S. Choy, B. Wong, G. Simon, and C. Rosenberg. The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-user Latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, NetGames '12, pages 2:1–2:6, Piscataway, NJ, USA, 2012. IEEE Press. 36

Y. Collet. LZ4 - Extremely Fast Compression, 2011. [Online]. Available: `http://cyan4973.github.io/lz4/`, (last accessed 2016/04/15). 55

J. Congote. MedX3DOM: MedX3D for X3DOM. In *Proceedings of the 17th International Conference on 3D Web Technology*, pages 179–179. ACM, 2012. 48

J. Congote, A. Segura, L. Kabongo, A. Moreno, J. Posada, and O. 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. 25, 45, 48, 69, 90

B. J. d'Auriol. Serviceable Visualizations. *The Journal of Supercomputing*, 61(3): 1089–1115, 2011. 27

M. Di Benedetto, F. Ponchio, F. Ganovelli, and R. Scopigno. SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW. In *Web3D 2010. 15th Conference on 3D Web technology*, 2010. 44, 46, 60

A. Diehl, L. Pelorosso, C. Delrieux, C. Saulo, J. Ruiz, M. E. Gröller, and S. Bruckner. Visual Analysis of Spatio-Temporal Data: Applications in Weather Forecasting. *Computer Graphics Forum*, 34(3):381–390, May 2015. 51

J. Diepstraten, M. Gorke, and T. Ertl. Remote Line Rendering for Mobile Devices. In *Computer Graphics International, 2004. Proceedings*, pages 454–461, June 2004. 25

M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009. 35

D. A. Duce and M. Sagar. skML a Markup Language for Distributed Collaborative Visualization. In L. M. Lever and M. McDerby, editors, *EG UK Theory and Practice of Computer Graphics*. The Eurographics Association, 2005. 31

K. Engel and T. Ertl. Texture-based Volume Visualization for Multiple Users on the World Wide Web. In M. Gervautz, D. Schmalstieg, and A. Hildebrand, editors, *Virtual Environments*, Eurographics, pages 115–124. Springer Vienna, 1999. 25

K. Engel, O. Sommer, C. Ernst, and T. Ertl. Remote 3D Visualization using Image-streaming Techniques. In *In ISIMADE - 11 TH Internationl Conference on Systems Research, Informatics and Cybernetics*, pages 91–96, 1999. 25

K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In W. de Leeuw and R. van Liere, editors, *Data Visualization 2000*, Eurographics, pages 167–177. Springer Vienna, 2000. 25

K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-salama, and D. Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 11, 90, 97

T. Erl, R. Puttini, and Z. Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013. 33

C. Evangelinos and C. N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA*, 2008. 38

A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Blat. 3D Graphics on the Web: A Survey. *Computers & Graphics*, 41:43 – 61, 2014. 19

W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated Performance Comparison of Virtual Machines and Linux Containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015. 38

I. Fette and A. Melnikov. The WebSocket Protocol, 2011. [Online]. Available: http://www.rfc-editor.org/info/rfc6455, (last accessed 2018/07/31). 17, 71

R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000, AAI9980887. 28, 71

M. Figueiredo, J. I. Rodrigues, I. Silvestre, and C. Veiga-Pires. Web3D Visualization of High Detail and Complex 3D-mesh Caves Models. In *2014 18th International Conference on Information Visualisation (IV)*, pages 275–280, July 2014. 51

I. Foster. Globus Toolkit Version 4: Software for Service-oriented Systems. In *Proceedings of the 2005 IFIP International Conference on Network and Parallel Computing*, NPC'05, pages 2–13, Berlin, Heidelberg, 2005. Springer-Verlag. 30, 31

I. Foster. Globus Online: Accelerating and Democratizing Science through Cloud-Based Services. *Internet Computing, IEEE*, 15(3):70–73, May 2011. 29

I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 29

I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Aug. 2001. 29

I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. *The Physiology of the Grid*, chapter 8, pages 217–249. Wiley-Blackwel, 05 2003. 29

I. Foster, Z. Yong, I. Raicu, and L. Shiyong. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, Nov 2008. 33

D. Frenkel and B. Smit. *Understanding Molecular Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2001. 73

S. Frey, F. Sadlo, and T. Ertl. Explorable Volumetric Depth Images from Raycasting. In *26th SIBGRAPI - Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 123–130, 2013. 90

S. Frey, F. Sadlo, and T. Ertl. Balanced Sampling and Compression for Remote Visualization. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, SA '15, pages 1:1–1:4, New York, NY, USA, 2015. ACM. 89

J. Gaillard, A. Vienne, R. Baume, F. Pedrinis, A. Peytavie, and G. Gesquière. Urban Data Visualisation in a Web Browser. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 81–88, New York, NY, USA, 2015. ACM. 51

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. *SIGPLAN Not.*, 44(6):465–478, June 2009. 2

G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In P. D'Ambra, M. Guarracino, and D. Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 379–391. Springer Berlin Heidelberg, 2010. 35

A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994. 8

J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: A Comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences. *Genome Biology*, 11(8):R86, 2010. 53

J. Goecks, C. Eberhard, T. Too, A. Nekrutenko, and J. Taylor. Web-based Visual Analysis for High-throughput Genomics. *BMC Genomics*, 14(1):1–11, 2013. 53

Google. Google Cloud, 2008. [Online]. Available: `https://cloud.google.com/`, (last accessed 2018/08/31). 34

L. J. Gosink, J. C. Anderson, E. W. Bethel, and K. I. Joy. Query-Driven Visualization of Time-Varying Adaptive Mesh Refinement Data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1715–1722, Nov 2008. 55

J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4): 34–41, Dec. 2005. 24, 38

S. Grottel, M. Krone, C. Muller, G. Reina, and T. Ertl. MegaMol—A Prototyping Framework for Particle-Based Visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 21(2):201–214, 2015. 3, 10, 60, 73, 75

S. Gumhold. Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings of the Vision, Modeling, and Visualization Conference 2003 (VMV 2003)*, pages 245–252, 2003. 60, 79

V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM. 35

R. B. Haber and D. A. McNabb. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In *Visualization in Scientific Computing*, pages 74–93. IEEE Computer Society Press, 1990. 8, 25, 30

D. Haehn, S. Knowles-Barley, M. Roberts, J. Beyer, N. Kasthuri, J. W. Lichtman, and H. Pfister. Design and evaluation of interactive proofreading tools for connectomics. *Visualization and Computer Graphics, IEEE Transactions on*, 20 (12):2466–2475, 2014a. 53

D. Haehn, N. Rannou, B. Ahtam, E. Grant, and R. Pienaar. Neuroimaging in the Browser using the X Toolkit. *Frontiers in Neuroinformatics*, (101), 2014b. 53

E. Haines. Ray Tracing Roundtable Report. *Ray Tracing News*, 14(1), 2001. 68

C. G. Healey and J. T. Enns. Attention and Visual Memory in Visualization and Computer Graphics. *Visualization and Computer Graphics, IEEE Transactions on*, 18(7):1170–1188, July 2012. 55

O. Hendin, N. W. John, and O. Shocet. Medical Volume Rendering Over the WWW Using VRML and Java. In Westwood, editor, *Medicine Meets Virtual Reality*, pages 34–40, Amsterdam, 1998. IOS Press and Ohmsha. 25

N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX. 37

A. Herrera. NVIDIA GRID:Graphics Accelerated VDI with the Visual Performance of a Workstation), 2013. [Online]. Available: http://www.nvidia.com/content/grid/vdi-whitepaper.pdf, (last accessed 2018/09/01). 35

S. Heule, D. Rifkin, A. Russo, and D. Stefan. The Most Dangerous Code in the Browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association. 2

B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association. 38

X. Hou, J. Sun, and J. Zhang. A Web-based Solution for 3D Medical Image Visualization. In *Proceedings of the SPIE*, volume 9418, page 8 pp., 2015. 49

C.-Y. Huang, C.-H. Hsu, Y.-C. Chang, and K.-T. Chen. GamingAnywhere: An Open Cloud Gaming System. In *Proceedings of the 4th ACM Multimedia Systems Conference*, MMSys '13, pages 36–47, New York, NY, USA, 2013. ACM. 35

W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996. 10, 59, 61, 69, 73

P. Isenberg, N. Elmqvist, J. Scholtz, D. Cernea, K.-L. Ma, and H. Hagen. Collaborative Visualization: Definition, Challenges, and Research Agenda. *Information Visualization*, 10(4):310–326, Oct. 2011. 11, 24

H. Jacinto, R. Kéchichian, M. Desvignes, R. Prost, and S. Valette. A Web Interface for 3D Visualization and Interactive Segmentation of Medical Images. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 51–58, New York, NY, USA, 2012. ACM. 48, 50

B. Jenny, B. Šavrič, and J. Liem. Real-time Raster Projection for Web Maps. *International Journal of Digital Earth*, 9(3):215–229, 2016. 51

S. Jeon and J. Choi. Reuse of JIT Compiled Code in JavaScript Engine. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1840–1842, New York, NY, USA, 2012. ACM. 2

J. Jiménez, A. López, J. Cruz, F. Esteban, J. Navas, P. Villoslada, and J. R. de Miras. A Web platform for the Interactive Visualization and Analysis of the 3D Fractal Dimension of {MRI} Data. *Journal of Biomedical Informatics*, 51: 176 – 190, 2014. 48

Jmol. Jmol: An Open-source Java Viewer for Chemical Structures in 3D, 2009. [Online]. Available: `http://www.jmol.org/`, (last accessed 2018/08/31). 10, 46, 59, 74

N. John, M. Aratow, J. Couch, D. Evestedt, A. Hudson, N. Polys, R. Puk, A. Ray, K. Victor, and Q. Wang. MedX3D: Standards Enabled Desktop Medical 3D. *Studies in health technology and informatics*, 132:189–194, 2007. 48

J. Jomier, S. R. Aylward, C. Marion, J. Lee, and M. Styner. A Digital Archiving System and Distributed Server-side Processing of Large Datasets. In *Proceedings of the SPIE 7264, Medical Imaging 2009: Advanced PACS-based Imaging Informatics and Therapeutic Applications*, volume 7264, 02 2009. 54

J. Jomier, S. Jourdain, U. Ayachit, and C. 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. 25, 54

S. Jourdain, U. Ayachit, and B. Geveci. ParaViewWeb, A web framework for 3D Visualization and Data Processing. *IADIS International Conference on Web Virtual Reality and Three-Dimensional Worlds*, 07 2010. 54

JSmol. JSmol: JavaScript-Based Molecular Viewer From Jmol , 2013. [Online]. Available: `http://sourceforge.net/projects/jsmol/`, (last accessed 2018/08/31). 3, 9, 46, 59, 74

V. Kajalin. Screen-Space Ambient Occlusion. In W. Engel, editor, *ShaderX7 : Advanced Rendering Techniques*, pages 413–424. Charles River Media, 2009. 81

S. Kastner and L. G. Ungerleider. Mechanisms of Visual Attention in the Human Cortex. *Annual Review of Neuroscience*, 23:315–341, 2000. 55

J. Kessenich, G. Sellers, and D. Shreiner. *OpenGL®Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V*. Addison-Wesley Professional, 9 edition, 2016. 15

Khronos. Typed Array Specification, 2011a. [Online]. Available: http://www.khronos.org/registry/typedarray/specs/latest/, (last accessed 2016/04/13). 17

Khronos. WebGL 1.0 Specification. http://www.khronos.org/registry/webgl/specs/latest/1.0/, 2011b, (last accessed 2014/02/21). 2, 13, 16, 19, 24, 56, 62

Khronos. WebGL Security, 2011c. [Online]. Available: https://www.khronos.org/webgl/security/, (last accessed 2015/03/13). 68

Khronos. WebGL 2.0 Specification, 2013. [Online]. Available: http://www.khronos.org/registry/webgl/specs/latest/2.0/, (last accessed 2014/02/24). 2, 16, 24, 56, 65, 81, 95, 97

Khronos. glTF 1.0 Specification, 2015. [Online]. Available: https://github.com/KhronosGroup/glTF/tree/master/specification, (last accessed 2016/01/28). 54

J.-S. Kim, N. Polys, and P. Sforza. Preparing and Evaluating Geospatial Data Models using X3D Encodings for Web 3D Geovisualization Services. In *Proceedings of the 20th International Conference on 3D Web Technology*, pages 55–63. ACM, 2015. 51

T. Klein and T. Ertl. Illustrating Magnetic Field Lines using a Discrete Particle Model. In *International Workshop on Vision, Modeling, and Visualization*, pages 387–394, 2004. 61

D. Koch and N. Capens. The ANGLE Project: Implementing OpenGL ES 2.0 on Direct3D. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, pages 543–570. CRC Press, July 2012, http://www.openglinsights.com/. 15

K. J. Kohlhoff, D. Shukla, M. Lawrenz, G. R. Bowman, D. E. Konerding, D. Belov, R. B. Altman, and V. S. Pande. Cloud-based Simulations on Google Exacycle Reveal Ligand Modulation of GPCR Activation Pathways. *Nature Chemistry*, 6(1):15–21, Dec. 2013. 38

W. L. Koltun. Space Filling Atomic Units and Connectors for Molecular Models, 1965, US Patent 3,170,246. 65

S. Koulouzis, E. Zudilova-Seinstra, and A. Belloum. Data Transport Between Visualization Web Services for Medical Image Analysis. *Procedia Computer Science*, 1(1):1727 – 1736, 2010, {ICCS} 2010. 27, 32

Y. Koval, H. Mendrul, A. Salnikov, I. Sliusar, and O. Sudakov. Interactive Dynamical Visualization of Big data Arrays in Grid. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015 IEEE 8th International Conference on*, volume 1, pages 153–156, Sept 2015. 32

B. Kozlíková, M. Krone, M. Falk, N. Lindow, M. Baaden, D. Baum, I. Viola, J. Parulek, and H.-C. Hege. Visualization of Biomolecular Structures: State of the Art Revisited. *Computer Graphics Forum*, 36(8):178–204, 2016. 10

D. Kranzlmüller, P. Heinzlreiter, H. Rosmanith, and J. Volkert. Grid-Enabled Visualization with GVK. In F. Fernandez Rivera, M. Bubak, A. Gomez Tato, and R. Doallo, editors, *Grid Computing*, volume 2970 of *Lecture Notes in Computer Science*, pages 139–146. Springer Berlin Heidelberg, 2004. 31

J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003*, pages 38–44, 2003. 69

A. Labour, M. Papakipos, S. Okasaka, and J. Timanus. Safe Browser Plugins using Native Code Modules, Jan. 8 2013. [Online]. Available: https://www.google.com/patents/US8352967, US Patent 8,352,967 (last accessed 2018/09/01). 2

A. Lagae and P. Dutré. Compact, Fast and Robust Grids for Ray Tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, June 2008. 63

T. Larsson, T. Akenine-Möller, and E. Lengyel. On Faster Sphere-Box Overlap Testing. *J. Graphics Tools*, 12(1):3–8, 2007. 64

G. Lavoué, L. Chevalier, and F. Dupont. Progressive Streaming of Compressed 3D Graphics in a Web Browser, 2014. [Online]. Available: http://liris.cnrs.fr/~glavoue/travaux/conference/SigTalk2014.pdf, (last accessed 2018/09/01). 90

H. Li, K. S. Leung, T. Nakane, and M. H. Wong. iview: An Interactive WebGL Visualizer for Protein-Ligand Complex. *BMC Bioinformatics*, 15:56, 2014. 3, 46, 60, 74

M. Limper, S. Wagner, C. Stein, Y. Jung, and A. Stork. Fast Delivery of 3D Web Content: A Case Study. In *Proceedings of the 18th International Conference on 3D Web Technology*, Web3D '13, pages 11–17, New York, NY, USA, 2013. ACM. 89

M. Limper, M. Thöner, J. Behr, and D. W. Fellner. SRC - a Streamable Format for Generalized Web-based 3D Data Transmission. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, Web3D '14, pages 35–43, New York, NY, USA, 2014. ACM. 26, 43, 53, 90

N. Lindow, D. Baum, and H.-C. Hege. Interactive Rendering of Materials and Biological Structures on Atomic and Nanoscopic Scale. *Computer Graphics Forum*, 31(3):1325–1334, 2012. 62

P. Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014. 55

G. J. S. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. W. M. van der Laak, B. van Ginneken, and C. I. Sánchez. A Survey on Deep Learning in Medical Image Analysis. *Medical Image Analysis*, 42:60 – 88, 2017. 102

Z. Liu, B. Jiang, and J. Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum*, 32(3pt4):421–430, 2013. 50, 51

J. Lluch, R. Gaitán, M. Escrivá, and E. Camahort. Multiresolution 3D Rendering on Mobile Devices. In V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 287–294. Springer Berlin Heidelberg, 2006. 25

B. Lorensen. On the Death of Visualization. In *Position Papers NIH/NSF Proc. Fall 2004 Workshop Visualization Research Challenges*, 2004. 24

K. Matsuda and R. Lea. *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*. Addison-Wesley, 2013. 13

A. Mayorga and M. Gleicher. Splatterplots: Overcoming Overdraw in Scatter Plots. *IEEE Transactions on Visualization and Computer Graphics*, 19(9):1526–1538, Sept. 2013. 50, 51

M. McCann, B. Yoo, and D. Brutzman. Integration of X3D Geospatial in a Data Driven Web Application. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, pages 145–145. ACM, 2014. 51

P. M. Mell and T. Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, United States, 2011. 33

D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), Mar. 2014. 38

Microsoft. Microsoft Azure Cloud, 2010. [Online]. Available: `https://azure.microsoft.com/`, (last accessed 2018/09/01). 34

R. Moreno-Vozmediano, R. Montero, and I. Llorente. IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. *Computer*, 45(12):65–72, Dec 2012. 34

C. Mouton, S. Parfouru, C. Jeulin, C. Dutertre, J.-L. Goblet, T. Paviot, S. Lamouri, M. Limper, C. Stein, J. Behr, and Y. Jung. Enhancing the Plant Layout Design Process Using X3DOM and a Scalable Web3D Service Architecture. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, Web3D '14, pages 125–132, 2014. 54

M. M. Movania and L. Feng. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 381–388, June 2012. 25, 45, 48, 49, 91

F. Mwalongo, M. Krone, G. Karch, M. Becher, G. Reina, and T. Ertl. Visualization of Molecular Structures Using State-of-the-art Techniques in WebGL. In *Proceedings of the Nineteenth International ACM Conference on 3DWeb Technologies*, Web3D '14, pages 133–141, New York, NY, USA, 2014. ACM. 5, 9, 26, 39, 48, 60, 62, 65, 67, 70

F. Mwalongo, M. Krone, M. Becher, G. Reina, and T. Ertl. Remote Visualization of Dynamic Molecular Data Using WebGL. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 115–122, New York, NY, USA, 2015. ACM. 5, 9, 26, 48, 74

F. Mwalongo, M. Krone, M. Becher, G. Reina, and T. Ertl. GPU-based Remote Visualization of Dynamic Molecular Data on the Web. *Graphical Models*, 88: 57–65, 2016a. 5, 74, 82

F. Mwalongo, M. Krone, G. Reina, and T. Ertl. State of the art report in web-based visualization. *Computer Graphics Forum*, 35(3):553–575, 2016b. 4, 23

F. Mwalongo, M. Krone, G. Reina, and T. Ertl. Web-based Volume Rendering using Progressive Importance-based Data Transfer. In Beck, Fabian and Dachsbacher, Carsten and Sadlo, Filip, editor, *Vision, Modeling and Visualization*, VMV18, pages 147–154. Eurographics Association, 2018. 6, 91

J. M. Noguera and J.-R. Jiménez. Visualization of Very Large 3D Volumes on Mobile Devices and WebGL. In *20th WSCG International Conference on Computer Graphics, Visualization and Computer Vision 2012*, 2012. 48, 90

D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 124–131, May 2009. 34

NVIDIA. NVIDIA GRID PC Streaming Service, 2015. [Online]. Available: http://shield.nvidia.com/grid-game-streaming, (last accessed 2018/08/31). 36

OpenStack. OpenStack Open Source Cloud Computing Software, 2010. [Online]. Available: https://www.openstack.org/, (last accessed 2018/08/31). 34

OTOY. The Future of Cloud Gaming, 2013. [Online]. Available: http://www.otoy.com/cgc/cloudgaming_2013.pdf, (last accessed 2015/07/09). 36, 56

G. Parulkar, J. Bowie, H.-W. Braun, R. Guerin, and D. Stevenson. Remote Visualization: Challenges and Opportunities. In *Proceeding Visualization '91*, pages 340–344, Oct 1991. 26

C. Pautasso. RESTful Web Services: Principles, Patterns, Emerging Technologies. In A. Bouguettaya, Q. Z. Sheng, and F. Daniel, editors, *Web Services Foundations*, pages 31–51. Springer New York, 2014. 28, 56

C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM. 28

J.-B. Pettit and J. C. Marioni. bioWeb3D: An Online WebGL 3D Data Visualisation Tool. *BMC Bioinformatics*, 14(1):185, 2013. 3, 60, 74

M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2010. 64

A. Plesch and M. McCann. The X3D geospatial component: X3DOM implementation of GeoOrigin, GeoLocation, GeoViewpoint, and GeoPositionInterpolator Nodes. In *Proceedings of the 20th International Conference on 3D Web Technology*, pages 31–37. ACM, 2015. 51

F. Ponchio and M. Dellepiane. Fast Decompression for Web-based View-dependent 3D Rendering. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 199–207, New York, NY, USA, 2015. ACM. 26, 41, 42, 53, 90

N. Rego and D. Koes. 3Dmol.js: Molecular Visualization with WebGL. *Bioinformatics*, 31(8):1322–1324, 2015. 3, 46

G. Reina and T. Ertl. Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *Eurographics/IEEE VGTC Symposium on Visualization*, pages 177–182, 2005. 61

B. Resch, R. Wohlfahrt, and C. Wosniok. Web-based 4D Visualization of Marine Geo-data using WebGL. *Cartography and Geographic Information Science*, 41(3): 235–247, 2014. 51, 52

L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013. 28

A. S. Rose and P. W. Hildebrand. NGL Viewer: A Web Application for Molecular Visualization. *Nucleic Acids Research*, 43(W1):W576–W579, 2015. 9, 46, 47, 60

T. Saito and T. Takahashi. Comprehensible Rendering of 3-D Shapes. *Computer Graphics (Proc. SIGGRAPH 1990)*, 24(4):197–206, 1990. 81, 82

A. R. Sanderson, B. Whitlock, O. Rübel, H. Childs, G. Weber, M. Prabhat, and K. Wu. A System for Query Based Analysis and Visualization. In K. Matkovic and G. Santucci, editors, *EuroVA 2012: International Workshop on Visual Analytics*. The Eurographics Association, 2012. 55

A. Sarikaya and M. Gleicher. Using WebGL as an Interactive Visualization Medium: Our Experience Developing SplatterJs. In R. Chang, C. Scheidegger, D. Fisher, and J. Heer, editors, *Proceedings of the Data Systems for Interactive Analysis Workshop*. IEEE, Oct 2015, DSIA '15. 50

W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit (4th ed.): An Object-oriented Approach to 3D Graphics*. Kitware, 2006. 31

S. Seely. *SOAP: Cross Platform Web Service Development Using XML*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. 28

R. Shea, J. Liu, E.-H. Ngai, and Y. Cui. Cloud Gaming: Architecture and Performance. *Network, IEEE*, 27(4):16–21, July 2013. 35, 56

S. Shi. Reduce Latency: The Key to Successful Interactive Remote Rendering Systems. In *2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 391–392, March 2011. 26

P. Shirley. Objects per Grid Cell. *Ray Tracing News*, 15(1), 2002. 68

R. Sicat, J. Krüger, T. Möller, and M. Hadwiger. Sparse PDF Volumes for Consistent Multi-Resolution Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2417–2426, 2014. 97

C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based Ray-casting of Quadratic Surfaces. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'06, pages 59–65, 2006. 61

J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. 30

K. Sons, F. Klein, D. Rubinstein, S. Byelozyorov, and P. Slusallek. XML3D: Interactive 3D Graphics for the Web. In *Proceedings of the 15th International Conference on Web 3D Technology*, Web3D '10, pages 175–184, New York, NY, USA, 2010. ACM. 19

K. Sons, C. Schlinkmann, F. Klein, D. Rubinstein, and P. Slusallek. xml3d.js: Architecture of a Polyfill implementation of XML3D. In *2013 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 17–24, March 2013. 20

R. Stevens, M. E. Papka, C. Johnson, P. Baker, J. Leigh, and S. Uselton. Challenges for Remote Visualization. In *Proceedings of the Conference on Visualization '01*, VIS '01, pages 519–522, Washington, DC, USA, 2001. IEEE Computer Society. 26

K. Stockinger, J. Shalf, K. Wu, and E. Bethel. Query-driven Visualization of Large Data Sets. In *Visualization, 2005. VIS 05. IEEE*, pages 167–174, Oct 2005. 55

M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013. 38, 56, 108

E. Sundén, P. Steneteg, S. Kottravel, D. Jönsson, R. Englund, M. Falk, and T. Ropinski. Inviwo - An Extensible, Multi-Purpose Visualization Framework. Poster at IEEE Vis, 2015. 97

R. Suselbeck, G. Schiele, and C. Becker. Peer-to-Peer Support for Low-Latency Massively Multiplayer Online Games in the Cloud. In *2009 8th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–2, Nov 2009. 36

J. Sutter, K. Sons, and P. Slusallek. Blast: A Binary Large Structured Transmission Format for the Web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, Web3D '14, pages 45–52, New York, NY, USA, 2014. ACM. 26, 90

Tableau. Tableau Online, 2013. [Online]. Available: `https://www.tableau.com/products/cloud-bi`, (last accessed 2018/08/31). 27, 34, 53

M. Tarini, P. Cignoni, and C. Montani. Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–1244, 2006. 81

A. C. Telea. *Data Visualization: Principles and Practice, Second Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2014. 7

Threejs. Three.js - javascript 3D Library, 2010. [Online]. Available: https://threejs.org/, (last accessed 2018/07/30). 19, 32, 44, 46, 49, 52

K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 121–132, Berkeley, CA, USA, 2014. USENIX Association. 35

TIBCO. TIBCO Spotfire Cloud, 2014. [Online]. Available: http://spotfire.tibco.com/products/spotfire-cloud, (last accessed 2018/08/31). 27, 34, 53

J. Trapp and H.-G. Pagendarm. A Prototype for a WWW-based Visualization Service. In W. Lefer and M. Grave, editors, *Visualization in Scientific Computing*, Eurographics, pages 21–30. Springer Vienna, 1997. 25

C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 4–16, Dec 2009. 38

F. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. ManyEyes: a Site for Visualization at Internet Scale. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1121–1128, Nov 2007. 11, 53

I. Virag, L. Stoicu-Tivadar, and E. Amăricăi. Browser-based Medical Visualization System. In *2014 IEEE 9th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 355–359, May 2014. 49

L. Vu, H. Sivaraman, and R. Bidarkar. GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor. In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 2:1–2:8, San Diego, CA, USA, 2014. Society for Computer Simulation International. 35

W3C. Web Services Architecture, 2004. [Online]. Available: https://www.w3.org/TR/ws-arch/, (last accessed 2018/09/01). 26

W3C. File API, 2013. [Online]. Available: https://www.w3.org/TR/FileAPI/, (last accessed 2018/09/01). 97

W3C. HTML Canvas 2D Context, 2015. [Online]. Available: https://www.w3.org/TR/2dcontext/, (last accessed 2018/09/01). 52

W3C. Web Workers: W3C Working Draft, 2015. [Online]. Available: https://www.w3.org/TR/workers/, (last accessed 2018/09/01]. 17, 95

W3C. HTML 5.2 W3C Recommendation, 2017. [Online]. Available: `https://www.w3.org/TR/html52/`, (last accessed 2018/09/01). 2, 16, 24, 56

W3C. WebRTC 1.0: Real-time Communication Between Browsers, 2018. [Online]. Available: `https://www.w3.org/TR/webrtc/`, (last accessed 2018/09/01). 18

I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Trans. Graph.*, 25(3):485–493, July 2006. 68

I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In D. Schmalstieg and J. Bittner, editors, *Eurographics 2007 - State of the Art Reports*. The Eurographics Association, 2007. 61

H. Wang, K. W. Brodlie, J. W. Handley, and J. D. Wood. Service-oriented Approach to Collaborative Visualization. *Concurrency and Computation: Practice and Experience*, 20(11):1289–1301, 2008. 27

J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 1st edition, 2010. 28

D. Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 8

L. Wen, J. Jia, and S. Liang. LPM: Lightweight Progressive Meshes Towards Smooth Transmission of Web3D Media over Internet. In *Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, VRCAI '14, pages 95–103, New York, NY, USA, 2014. ACM. 26, 41

WHATWG. OffscreenCanvas, 2018. [Online]. Available: `https://html.spec.whatwg.org/multipage/canvas.html#the-offscreencanvas-interface`, (last accessed 2018/09/01). 106

E. Wilde and C. Pautasso. *REST: From Research to Practice*. Springer Publishing Company, Incorporated, 1st edition, 2011. 28

J. Wood, K. Brodlie, and H. Wright. Visualization over the World Wide Web and Its Application to Environmental Data. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, pages 81–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. 25

J. Wood, K. Brodlie, J. Seo, D. Duke, and J. Walton. A Web Services Architecture for Visualization. In *IEEE Fourth International Conference on eScience '08*, pages 1–7, Dec 2008. 27, 30, 31

X3DOM. Terrain of Puget Sound, 2017a. [Online]. Available: https://examples.x3dom.org/BVHRefiner/BVHRefiner.html, (last accessed 2018/09/01). 22

X3DOM. Volume Visualization Medical Dataset, 2017b. [Online]. Available: https://examples.x3dom.org/volren/volrenOpacityTestTF_aorta.xhtml, (last accessed 2018/09/01). 22

C.-T. Yang, H.-Y. Wang, W.-S. Ou, Y.-T. Liu, and C.-H. Hsu. On Implementation of GPU Virtualization using PCI Pass-through. In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 711–716, Dec 2012. 35

Y. Yang, A. Sharma, and A. Girier. Volumetric Texture Data Compression Scheme for Transmission. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 65–68, New York, NY, USA, 2015. ACM. 43, 91

W. Yoo, S. Shi, W. Jeon, K. Nahrstedt, and R. Campbell. Real-time Parallel Remote Rendering for Mobile Devices using Graphics Processing Units. In *2010 IEEE International Conference on Multimedia and Expo (ICME)*, pages 902–907, July 2010. 25

E. Zudilova-Seinstra, N. Yang, L. Axner, A. Wibisono, and D. Vasunin. Service-oriented Visualization Applied to Medical Data Analysis. *Service Oriented Computing and Applications*, 2(4):187–201, 2008. 27, 31