FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Decoupling Object Manipulation from Rendering in a Thin Client Visualization System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Anna Frühstück BSc
Matrikelnummer 0626930

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr.techn. Peter Rautek

Wien, 29. September 2015

Anna Frühstück | Eduard Gröller

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Decoupling Object Manipulation from Rendering in a Thin Client Visualization System

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Anna Frühstück BSc
Registration Number 0626930

to the Faculty of Informatics
at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Eduard Gröller
Assistance: Dipl.-Ing. Dr.techn. Peter Rautek

Vienna, 29th September, 2015

Anna Frühstück

Eduard Gröller

Erklärung zur Verfassung der Arbeit

Anna Frühstück BSc
Argentinierstr. 28/11
1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. September 2015

Anna Frühstück

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisors. I would like to thank Prof. Eduard Gröller for his support of my study and related research, for inspiring me to go abroad to work on my project and for his willingness to supervise me remotely.
It was a privilege for me to work with Dr. Peter Rautek on my thesis. His guidance helped me during the time of research and writing of this thesis. I am happy to have had him as my advisor and mentor for my thesis project.

My loving parents Martha and Anton Frühstück have made it possible for me to pursue the studies of my choice, supported me without reservation and have always encouraged me to find my way, even if it was not always the shortest path chosen. For their never-ending, considerate and most of all unconditional support I would like to thank them with all my heart.

To my wonderful siblings, who are friends to me as much as family, thank you for being always there, readily providing help, inspiration and laughter in equal parts.

Furthermore I want to express my thanks to my very good friends and colleagues in Vienna that continuously helped and motivated me throughout the course of my studies.

I am forever grateful for the amazing opportunity I got to be part of a group of smart, inspiring colleagues at the King Abdullah University of Science and Technology. It was truly a joy and eye-opening to get an insight into how passionate my colleagues feel about their research, be part of their stimulating discussions and to be invited into their amicable, welcoming and creative environment.

Lastly, I would like to extend my gratitude to the International Office of the Technical University of Vienna which supported my stay at the King Abdullah University of Science and Technology with a stipend for short-term scientific work abroad.

Danksagung

Zunächst möchte ich mich aufrichtig bei meinen beiden Betreuern bedanken. Prof. Eduard Gröller danke ich für die Unterstützung meiner Diplomarbeit und der damit verbundenen Forschungstätigkeit, für das Angebot, im Ausland an meinem Projekt arbeiten zu können und seine Bereitschaft, mich dabei zu betreuen.
Ich bin dankbar für die Möglichkeit, unter der Anleitung von Dr. Peter Rautek an meiner Diplomarbeit zu arbeiten. Seine Beratung während der Forschungsarbeit und der Erstellung der Diplomarbeit war von großem Wert für mich, und ich bin froh, ihn als Betreuer und Mentor meines Projektes zu haben.

Meine liebevollen Eltern Martha und Anton Frühstück haben es mir ermöglicht, das Studium meiner Wahl zu verfolgen. Dabei haben sie mich vorbehaltslos unterstützt und mich stets dazu ermutigt, meinen eigenen Weg zu verfolgen, selbst wenn es nicht der kürzeste war. Für ihren nie endenwollenden, fürsorglichen, unaufdringlichen und bedingungslosen Beistand möchte ich mich bei ihnen von ganzem Herzen bedanken.

Meine wunderbaren Geschwister, die für mich gleichzeitig Familie und Freunde sind: Danke dafür, dass ihr immer für mich da seid, um mir je nach Anlass bereitwillig mit Hilfe, Inspiration und Humor beiseite zu stehen.

Zudem möchte ich meinen guten Freunden und Kollegen in Wien, auf deren Unterstützung und Motivation ich im Laufe des Studiums immer zählen konnte, meinen Dank aussprechen.

Immer dankbar werde für die großartige Chance sein, für die Dauer dieser Arbeit Teil einer Gruppe von kompetenten und inspirierenden Kollegen an der King Abdullah University of Science and Technology sein zu können. Es war mir ein Privileg, in dieser Umgebung zu spüren, welche Begeisterung meine Kollegen für ihre Forschungsprojekte empfinden und in ihrem freundschaftlichen und kreativen Umfeld leben und arbeiten zu dürfen.

Zuletzt möchte ich mich beim International Office der Technischen Universität Wien bedanken, das meinen Aufenthalt an der King Abdullah University of Science and Technology mit einem Stipendium für kurzfristige wissenschaftliche Arbeiten im Ausland unterstützt hat.

Kurzfassung

Nutzern von Visualisierungsanwendungen stehen häufig keine leistungsfähigen Systeme für rechenintensive Visualisierungsaufgaben zur Verfügung. Ein *Remote Rendering* der Visualisierung und der Einsatz eines Thin Clients (z.B. ein Web Browser) zur Darstellung der Resultate ermöglichen den Zugriff auf die Visualisierungen sogar von Geräten, die nicht für Grafikanwendungen geschaffen sind. In derartigen thin-client Konfigurationen erleidet jedoch die Flexibilität, die Daten interaktiv zu manipulieren, Einbußen. Dadurch wird eine sinnvolle Interaktion mit Datensätzen, die viele Objekte enthalten, erschwert. Besonders für In-Situ Visualisierungssysteme stellt die Möglichkeit zur direkten Interaktion mit den Daten eine Herausforderung dar.

Wir lösen dieses Problem durch unseren Ansatz, die Berechnung der Visualisierung anhand einer *Deferred Visualization Pipeline* zwischen Client und Server aufzuteilen. Unser Client baut auf Webtechnologien auf (HTML5, JavaScript) und ermöglicht mittels der D3 Library interaktive datengesteuerte Visualisierungen. Wir führen eine Zwischenrepräsentation der Objekte ein, die die Daten, welche vom Server zum Client gesendet werden, beschreibt. Dabei führt der Server die rechenintensiven Teile der Pipeline durch, während der Client durch die Möglichkeit, ohne erneutes Rendering Objektmodifikationen vorzunehmen, Flexibilität bewahrt.

Wir bezeichnen die Zwischenrepräsentation der Deferred Visualization als Volume Object Model. Dieses Modell besteht aus Metadaten und gerenderten Visualisierungen für jedes Objekt in einem Datensatz.

Um selbst für große Datensätze die Interaktivität des Clients sicherzustellen, werden dem Client in einer Pre-Visualization Stufe zunächst nur die Metadaten übertragen. Dadurch kann der Anwender anhand der Metadaten eine Filterung der Daten vornehmen, wodurch die Komplexität der Visualisierung bereits vor dem Streaming der Bilddaten reduziert werden kann. Ist der Anwender mit der Filterung zufrieden, werden die Objektbilder vom Server angefordert. In Verbindung mit den Metadaten kann so die finale Visualisierung aus den Bildern zusammengesetzt werden. Zudem können alle Objekte der Visualisierung untersucht und mittels einer integrierten Konsole programmiert werden.

Zusammenfassend gestattet unser System dem Nutzer, voll interaktive objektbezogene Visualisierungsschritte in einem Web Browser vorzunehmen, ohne ein aufwendiges erneutes Rendern am Server zu erfordern.

Abstract

Often, users of visualization applications do not have access to high performance systems for the computationally demanding visualization tasks. Rendering the visualization remotely and using a thin client (e.g. a web browser) to display the result enable the users to access the visualization even on devices that do not target graphics processing. However, the flexibility to manipulate the data set interactively suffers in thin-client configurations. This makes a meaningful interaction with data sets that contain many different objects difficult. This is especially true in in-situ visualization scenarios, where direct interaction with the data can be challenging.

We solve this problem by proposing an approach that employs a deferred visualization pipeline to divide the visualization computation between a server and a client. Our thin client is built on web technologies (HTML5, JavaScript) and is integrated with the D3 library to enable interactive data-driven visualizations. An intermediate representation of objects is introduced which describes the data that is transferred from the server to the client on request. The server side carries out the computationally expensive parts of the pipeline while the client retains extensive flexibility by performing object modification tasks without requiring a re-rendering of the data.

We introduce a novel Volume Object Model as an intermediate representation for deferred visualization. This model consists of metadata and pre-rendered visualizations of each object in a data set.

In order to guarantee client-side interactivity even on large data sets, the client only receives the metadata of all objects for a pre-visualization step. By allowing the user to perform filtering using the metadata alone, the complexity of the requested visualization data can be reduced from the client side before streaming any image data. Only when the user is satisfied, the object images are requested from the server. In combination with the metadata, the final visualization can then be reconstructed from the individual images. Moreover, all objects in the visualization can be investigated and changed programmatically by the user via an integrated console.

In summary, our system allows for fully interactive object-related visualization tasks in a web browser without triggering an expensive re-rendering on the server.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

List of Figures **xvi**

1 Introduction **1**
1.1 Motivation 2
1.2 The Pipeline Concept 3
1.3 Deferred Visualization Pipeline 4
1.4 Thesis Overview 5

2 Fundamentals and State of the Art **7**
2.1 Deferred Rendering Pipelines 8
2.2 Remote Visualization 12
2.3 In-Situ Visualization 14
2.4 Visualization in the Browser 16
2.5 Rendering of Segmented Volume Data 17

3 Remote Visualization with Deferred Object Interaction **21**
3.1 Remote Visualization Architecture 22
3.2 Volume Object Model 25
3.3 Summary 37

4 Implementation **39**
4.1 Framework 39
4.2 Data and Segmentation 43
4.3 Server Implementation 43
4.4 Client/Server Communication 51
4.5 Client Implementation 52
4.6 Limitations 54

5 Results **57**
5.1 Server-side Object Rendering 57
5.2 Web Client 61
5.3 Client-Side Interaction 63
5.4 Evaluation 78

6 Conclusion and Future Work **81**
6.1 Outlook 82

A Appendix **83**
A.1 Rectangle Packing Algorithm 83

Bibliography **85**

List of Figures

1.1 Conceptual Steps of a Graphics Pipeline 3
1.2 The Deferred Visualization Pipeline 4

2.1 Venn Diagram of Concepts 7
2.2 Illustration of Deferred Shading 8
2.3 Visualization by Proxy 12
2.4 Classification of Remote Rendering 13

3.1 System Architecture 21
3.2 Pipeline on the Server 23
3.3 Pipeline on the Client 26
3.4 Scene Recompositing Depth Problem 32
3.5 Solving the Depth Problem 33
3.6 Object Access through DOM Handles in Web Client 35

4.1 VolumeShop Session View 40
4.2 VolumeShop Web Viewer 41
4.3 Block Subdivision in Rectangle Packing 46
4.4 Intermediate Rectangle Packing Result 47
4.5 Final Rectangle Packing Result 48
4.6 Object Layer Representations 49
4.7 Rendering Objects to their Target in the Render Pass 51

4.8 Screenshot of the Web Client 52

5.1 Human hand data set 58
5.2 Christmas tree data set 58
5.3 Object Rendering According to the Label Color 59
5.4 Object Rendering with Colors based on Transfer Function 59
5.5 Object Rendering with Blending 60
5.6 Object Rendering with Depth Buffer 60
5.7 Hand Data Set Displayed on the Client 61
5.8 Client-Side Scene Reassembly 62
5.9 Object Positioning on the Client 62
5.10 Object Reordering on the Client 63
5.11 Object Grouping through DOM Handles 64
5.12 Object Operations on the Client 66
5.13 Filtering Workflow on the Client 70
5.14 Object Modification Workflow on the Client 71
5.15 Scatterplot Visualization: Axes Generation 75
5.16 Scatterplot Visualization: Plotting the Values 76
5.17 Scatterplot Transition 77

CHAPTER 1

Introduction

Contemporary scientists are confronted with the task of investigating real-life or simulated data in order to gain novel findings and insights into their field of research. The concept of observation and exploration of data for the acquisition of scientific knowledge is not new. However, modern technology has changed the character of the available data significantly.

When, in the pre-digital era, data was primarily procured through human observation, methodologies for the analysis of the data were usually close at hand. The data at the disposal of present-day scientists is obtained through machine-supported processes, which renders the results virtually always either very abstract, extensive or both.

Data is frequently acquired through the sampling of real-world signals in order to evaluate physical conditions occurring in actual settings. This includes, among others, applications in industrial, medical, biological and geophysical fields of research. Depending on the nature of the investigated data, the process of gathering data differs. The core component of data acquisition consists of sensors transforming physical phenomena to electrical signals. For instance, CT (x-ray computed tomography) and MRI (magnetic resonance imaging) are two commonly used techniques for the recording of three-dimensional volumetric data sets from real-world objects. The output generated by any type of acquisition technique needs to be converted into a numeric format which makes it applicable for the manipulation through computerized systems.

Simulation, on the other hand, imitates processes of real-world systems by numerical models that represent the key characteristics of the respective process as closely as possible. While it is generally not possible to take into account every physical property of the modeled process, scientists develop complex models. The behavior of these models corresponds to the outcome that the underlying system would be expected to have. This procedure allows researchers to make predictions about events and to evaluate the impact of different conditions and courses of action. The simulation of a system facilitates an experimental insight even if the system does not exist, is inaccessible or the experiment would be impractical.

1.1 Motivation

Visualization has gained a significant role in the investigation of many different fields of science. The nature of the acquired data often makes it difficult to understand without the aid of further computational methods. Suitable tools like scientific visualization that help in the investigation process are therefore often indispensable. In most cases of research, the continuously growing amount of data which scientists wish to analyze is not only far too extensive for an individual examination of the data values. Also, the human brain is not particularly well adapted to find patterns in data in numerical form.

On the other hand, an important quality of the brain is its inherent ability to recognize visual patterns, which enables humans to quickly and instinctively evaluate complex situations. Therefore, devising meaningful visual presentations of data does not only accelerate the recognition of particular data properties, frequently it is a crucial instrument in the investigation process. Along these lines, volume visualization has become an important tool for gaining insight into three-dimensional, possibly also multivariate data, in many scientific fields.

Volume visualization systems are typically used by domain scientists who wish to investigate their data through a high-level interface, rather than interfering directly with source code and data at a low level. Therefore, visualization environments usually provide graphical user interfaces for the configuration of the system as well as for the direct interaction with the visualization parameters. However, this interaction metaphor is limiting in complex environments. To extend the adaptability and flexibility of visualization interfaces at a higher level, Rautek et al. [RBGH14] have proposed the use of domain specific languages (DSLs). Their system *ViSlang* incorporates multiple DSLs targeting individual volume processing, querying and visualization tasks of the visualization pipeline. The individual components of a visualization system are separated and can be programmed using different DSLs.

By offering the scientist a programmable interface that executes commands in DSLs targeted to describe common tasks for the manipulation of visualizations, a compromise is made between flexibility and ease of use.

The architecture of modern visualization systems is modeled according to the established visualization pipeline. Pipeline models structure the tasks that are performed in a process into several big steps. The visualization pipeline model formulates the necessary steps between the acquisition of raw data and the visualization image that is presented to the user. When traversing the visualization pipeline, these individual steps can be performed either in one software solution or using multiple tools. The visualization pipeline has to be customized for each data type and application.

Our approach aims to develop a pipeline for visualization tasks on segmented objects. Given large amounts of data, it makes sense to move most of the workload to a remote server. This is particularly relevant in in-situ visualization scenarios, in which visualization is performed without intermediate I/O-operations directly where the data is generated. An example would be a simulation performed on a supercomputer that generates data faster

Figure 1.1: The basic rendering pipeline. This pipeline is composed of three conceptual steps: application, geometry and rasterization. These stages can be parallelized and can be pipelines themselves. The pipeline model is adopted by modern graphics hardware. This allows the GPU to parallelize operations in several pipeline stages.

than the I/O-subsystem can store it to disk.

We therefore attempt to find the best possible stage to split the pipeline between the client and the server. Our goal is to create a visualization pipeline that provides the client with a high degree of interactivity for object-related investigative tasks. To that end, we give the user access to program the client-side visualization. However, we want to perform the computationally intensive volume rendering entirely on the server side. As a result, the objective of our approach is to output an intermediate representation from the server. This representation enables the client to recreate the visualization without requiring further access to the original data set.

In the following, we describe and compare common pipeline models and discuss how we structure the pipeline for our own approach.

1.2 The Pipeline Concept

The transformation of three-dimensional data (like point clouds, polygon meshes or volumetric data) into two-dimensional images is a common task in computer graphics applications. The rendering process generates images from the three-dimensional data in a series of consecutive computational steps. Rendering pipelines are the conceptual definitions of the steps of a graphics system. Since these steps depend on soft- and hardware constraints as well as the specifics of the data and required software features, there is not one universal graphics pipeline.

Frequently used pipelines are exposed to developers in graphics APIs like *Direct3D* [Bly06] or *OpenGL* [WNDS99], which abstract the underlying hardware. These Real-Time Rendering APIs use the rendering pipeline model to expose the independent steps of their pipeline to developers. Individual steps of the pipeline can therefore be optimized separately. This separation allows the vendor to specialize the operations of one step and to implement specific functions in massively parallel hardware.

Stages

The rendering pipeline is conceptually divided into the three stages *application*, *geometry* and *rasterization* [AMHH02], as illustrated in Figure 1.1. The term pipeline implies that the

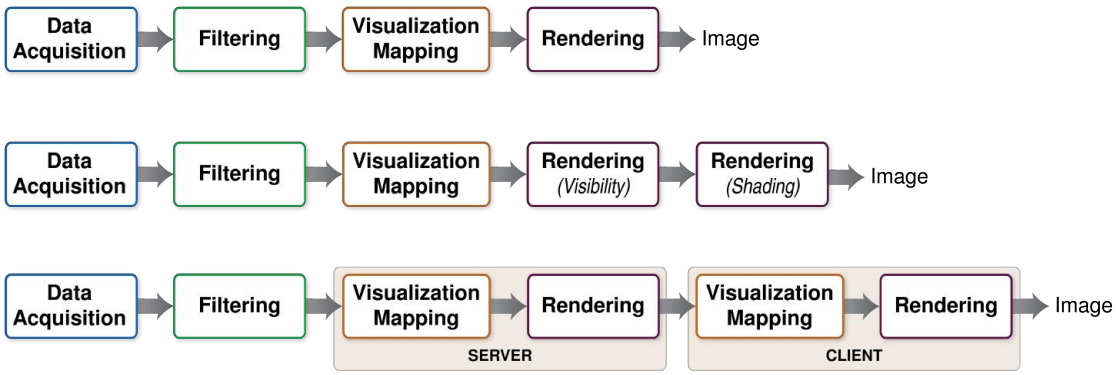


Figure 1.2: Different versions of the visualization pipeline: The top row shows the traditional visualization pipeline with Data Acquisition, Filtering, Visualization Mapping and Rendering stages. The middle row depicts a modified pipeline that applies the concept of deferred rendering to the last step. One example for such a pipeline was used by Hadwiger et al. [HSS+05]. In the bottom row, we show a split pipeline which performs different visualization mapping and rendering stages on the server and on the client, respectively. Tikhonova et al. [TCM10c] use such a pipeline for deferred visualization in volume rendering where the transfer function can be modified on the client. Our approach also employs this pipeline structure. However, we focus on separating the rendering of objects from their manipulation. This allows us to interact with objects on the client without re-rendering on the server.

calculation of one stage needs to be completed and the results made available before the next stage can start its processing. Each of the stages of the rendering pipeline may consist of several substages and hence be a pipeline itself.

The application stage, usually executed on the CPU, is responsible for gathering commands from input devices and execute corresponding operations on the data. The objective of the application stage is to prepare the geometrical primitives for being passed to the geometry stage. In the geometry stage, the majority of per-polygon and per-vertex operations are executed. The rasterizer transforms the geometrical primitives into discrete fragments, where each fragment corresponds to a pixel in the frame buffer.

While the internal structure of conventional graphics cards that offered hardware accelerated rendering relied heavily on implementing the graphics pipeline, these conditions became restrictive with rising demands for flexibility. This is why modern graphics cards offer a programmable shader-controlled pipeline which allows the developer to program individual steps of the pipeline.

1.3 Deferred Visualization Pipeline

In a visualization context, common pipeline models integrate the iterative user exploration. Two typical visualization pipeline models are shown in comparison to our pipeline in Figure 1.2. The user typically repeats the filtering and mapping tasks until a desirable visualiza-

tion is achieved. Frequently, scientists are interested in particular parts of the visualization while other parts are considered less important and only displayed to provide context. When the scientist wishes to interact with the object of interest, the conventional visualization pipeline suggests that the entire view needs to be re-rendered. The re-rendering, however, is largely redundant and rendering the entire volume interactively on user input produces considerable computational overhead. In a remote visualization system, this introduces additional strain on the network performance, since the requested rendering has to be streamed to the client over the network. These tasks are prohibitive in large-scale environments.

We propose a modified pipeline for visualization which renders to an intermediate representation. This setup permits the user to interact with the objects based on this representation rather than on the full dataset, greatly reducing the complexity of the operations.

In order to adapt our approach to in-situ visualization scenarios, we furthermore subdivide the object-level interaction on the data into two passes, as illustrated in the bottom row of Figure 1.2. On the server, we first generate an intermediate representation of the data. The client then performs separate visualization mapping and rendering stages which allow deferred interaction with objects. Instead of interacting with all of the data, the user interacts with a more lightweight representation of the data. In practice, we implement the deferred visualization pipeline as a two-pass process where the first pass is executed on a server and the second pass on a thin client.

To accommodate operations on large-scale data, we allow the user to define *what* constitutes an object in the first pass and *how* objects are visualized in the second pass. The metadata that is gathered in the first pass can be visualized in an initial pre-visualization step. This allows for a manipulation and filtering of the objects based on the object attributes in the metadata. When the actual visualization is requested from the server, the visualization data is filtered according to the result of the pre-visualization step. Only the objects that are actually relevant to the user are transmitted. This workflow therefore makes our approach applicable for large-scale in-situ visualization systems.

1.4 Thesis Overview

In the scope of this thesis a deferred visualization system was designed and implemented. The concepts and architecture of this system are the principal subjects in the following chapters.

We first review related work and the fundamental concepts applied in our system in Chapter 2. The main concepts of our approach are presented in Chapter 3. We discuss the structural stages of our pipeline, the intermediate representation and a conceptual overview of the client/server-architecture. We present details of our implementation in Chapter 4, including the framework and optimizations we implemented in our client/server system. Results and use cases of our system are shown in Chapter 5. Finally, we conclude with Chapter 6, which summarizes our approach and discusses our contributions and future work.

CHAPTER 2

Fundamentals and State of the Art

The contribution of this work is based on several core concepts that will be described in this chapter of the thesis in order to provide an overview of the context within which we place our work. The following subsections aim to give an overview of the fundamentals of the specific topics relevant to our approach. Moreover, related work will be discussed for each of the areas.

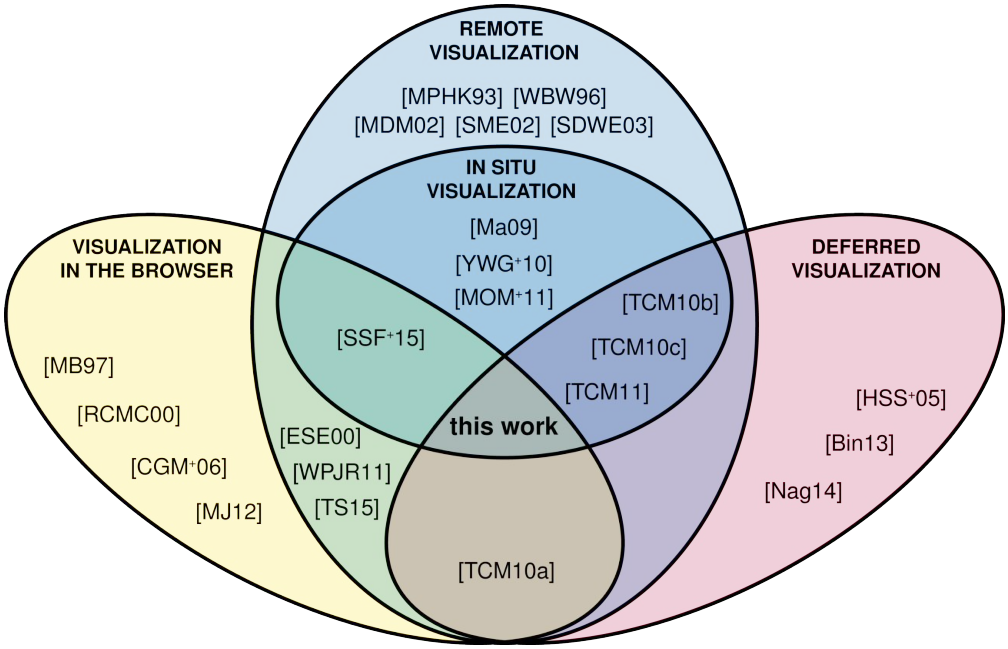


Figure 2.1: Diagram of the different concepts used in this thesis. We outline these concepts as overlapping areas and present how the related work described in this chapter can be classified among these concepts. Our work is placed at the intersection of all four main concepts.

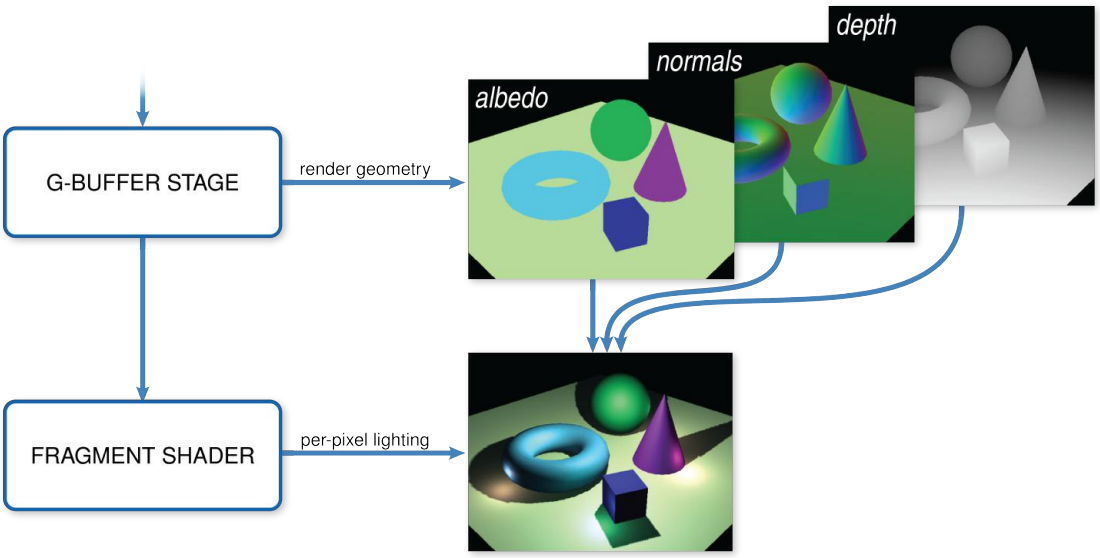


Figure 2.2: The first pass of deferred shading stores the albedo, the depth values and the normals into the respective off-screen buffers. The result of the combination of the intermediate deferred shading results is output at a later stage. The deferred shading approach thus allows for the surfaces to be lit in the final fragment shader.

Images from *Wikimedia Commons* [Ast15]

Figure 2.1 illustrates how the related work we describe in this chapter is placed at the intersections of the central methodologies we integrate in our work. We believe that our work is the first to incorporate all of these concepts into one approach.

2.1 Deferred Rendering Pipelines

The standard pipeline for image generation in computer graphics employs *forward rendering* [AMHH02], where visibility computation and shading are performed in the same rendering pass. However, for some applications, it has proven useful to defer specific parts of the rendering process to a later point in the graphics pipeline for increasing the performance or flexibility of the approach. These *deferred rendering* pipelines work by writing intermediate results of earlier pipeline stages to corresponding image buffers. In a later stage or an additional render pass, the values of the intermediate buffers are taken into account for the generation of the final image. By retaining these buffers describing certain scene properties for every output pixel, the combination of all influencing factors on the value of a pixel can be delayed to a later stage, which allows for a recombination of the buffers according to certain properties or the gathering of additional information before the ouput is evaluated.

2.1.1 Deferred Shading of Geometry

The first approaches that chose to defer certain calculations of the rendering process to a later stage of the computation pipeline were devised in the field of 3D computer graphics when the concept of deferred shading, sometimes also referred to as deferred rendering [HH04], postponed the actual shading of the geometry to a second, independent pass.

When the concept of deferred shading was first introduced by proposing the methodology of shading every pixel only once after depth resolution [DWS+88], the term 'deferred' was not mentioned in association with this technique yet. Even the introduction of deferred shading as we know it today, which operates through the use of G-buffers [ST90], still does not use the term 'deferred'. Since deferred shading became feasible on consumer graphics cards around 2005, the technique became popular in video games [Shi05, Koo08].

In the first pass of this technique, the vertex and pixel shader are executed to gather attributes of the scene's geometry. This information includes the position or the depth value, normals and diffuse color of every visible surface, as shown in Figure 2.2. Other approaches enhance the technique and opt to additionally store material properties, emissive maps or per-pixel specular values for later passes. However, no actual shading is performed in the first pass and the gathered information is stored in off-screen render targets. The intermediate computation results of the first render pass are stored into the geometry buffer (G-buffer), which then contains a projection of certain attributes of the geometry as seen from the viewpoint.

For any frame buffer, the computation of the fully lit result image can be done in the fragment shader by solving the BRDF for each pixel once per light source. This gives the ability to render a large number of light sources in a scene without a significant performance-hit.

Variants of Deferred Shading

In recent years, techniques that optimize the algorithm and address limitations of the original approach have emerged [Pla06]. The original deferred shading approach is not able to handle multiple materials, a limitation which can be overcome by storing additional properties in the G-buffer at the cost of the increased buffer size [Shi05].

A serious disadvantage is also the algorithm's inability to deal with transparent objects, which some approaches try to address by either using a separate pipeline to render transparent materials [Koo08] or interlacing transparent and opaque data and de-interlacing them in a later composition step [Pan09].

The separation of the lighting from the geometric stage also entails that anti-aliasing does not produce correct results, since a single pixel in the G-buffer maps to a single output pixel, which is a drawback of the original algorithm. Recent approaches use supersampling techniques through performing multi-sampling anti-aliasing [Thi09], subpixel reconstruction antialiasing [CML11] or adaptive supersampling [HBE13] to deal with this issue.

Light indexed rendering [Tre09] stores an index to the light affecting each pixel instead of storing the normals and colors per pixel.

Decoupled deferred shading for hardware rasterization [LD12] stores shading samples independently from the visibility in order to cache and reuse shading computation.

Deferred Lighting

Deferred Lighting [GPB04], also referred to as Light Pre-Pass Rendering [Eng09] can loosely be described as a modification of deferred shading. The concept differs in that only the lighting – and not the shading – computations are deferred. This approach is frequently used in video games.

This technique uses three passes in contrast to deferred shading, which is performed in two render passes. The first traversal of the scene geometry serves to evaluate the attributes necessary to compute per-pixel lighting and stores them to the G-buffer. The second step is a pass over the screen-space which computes per-pixel diffuse and specular shading equations, writing to specular and diffuse accumulation buffers. In the actual render pass, the scene geometry is traversed once more, accessing the lighting data from the accumulated textures and modulating them with the diffuse and specular colors before writing the final per-pixel shading into the color buffer.

The benefit of the deferred lighting approach is a reduction in the size of the G-buffer, which can get very large in deferred shading settings. Disadvantages include the obvious cost of an additional scene traversal as well as the fact that the diffuse and specular irradiance values are stored separately during the deferred pass. This either requires multiple render targets or two passes, whereas the deferred pass in deferred shading outputs a single combined radiance value. The separation of diffuse and specular irradiance overcomes the disadvantage of the original deferred shading algorithm of being only able to render one type of material.

Deferred Ambient Occlusion

Ambient Occlusion is a rendering concept where the influence of ambient lighting on a specific point in the scene is calculated. While not physically correct, this approach can be used to achieve a realistic-looking shadowing of a scene at relatively low computational cost.

Screen Space Ambient Occlusion [Mit07] uses the positional and normal data stored in the G-buffer to calculate the screen-space ambient occlusion values. Since the G-buffer can be reused, this strategy makes this approach very well suitable for a combination with deferred shading. The result of the ambient-occlusion calculation is stored to a gray-value representation which is multiplied with the result of the regular rendering as a post-processing effect.

In order to achieve more physically realistic occlusion effects, Bunnell [Bun05] converts the polygons of the meshes represented in a scene to disk-shaped elements in a preprocessing pass to facilitate the calculation of the degree of shadowing influence one surface has on another one. This method also demonstrates how Dynamic Ambient Occlusion can be used for ambient lighting. Hoberock and Jia [HJ07] extend this approach by building a hierarchical tree from the shading-disk representations and applying them in a deferred shading method.

The approaches Deferred Occlusion from Analytic Surfaces [SBSO09] and Ambient Occlusion Volumes [McG10] extend Hoberock and Jia's method into a single-pass screen-space technique.

2.1.2 Deferred Shading of Isosurfaces

In the process of rendering isosurfaces of dense volumetric grids, the shading of the isosurfaces can be deferred to the end of the pipeline [HSS+05]. During volume ray-casting, the location of the intersection of the ray with the isosurface is stored to an off-screen buffer in volumetric coordinates or in depth values that can be transformed to volume coordinates. This allows for all further computations and shading operations to be performed in image space.

An arbitrary number of screen-space passes for further operations on the data can be performed. For every pixel, the recorded volumetric location can be evaluated from the image. The results of calculations on the corresponding voxel can be stored to additional buffers. Through this process, buffers can be generated for the gradient and normal vectors as well as second derivatives for the visualization of curvature. Using the curvature, illustrative visualization techniques like ridge and valley lines can be implemented.

Binyahib [Bin13] allows for the selection of multiple isovalues. The isosurfaces corresponding to the specified values are then rendered to deferred buffers from a fixed view point. A selection of the resulting isosurfaces can be superimposed with correct visibility while facilitating a modification of the colormap and relighting.

2.1.3 Deferred Shading of Line Fields

Nagoor [Nag14] uses a deferred rendering approach for line fields consisting of path lines corresponding to particle trajectories in large-scale combustion simulations. This deferred line-field representation stores per-pixel linked lists for every image pixel. Then, a variety of attributes for the rendering of the pathlines, such as particle creation time and scalar properties such as temperature or soot density, can be modified interactively.

2.1.4 Deferred Volume Visualization

Explorable Images [TCM10a] are multi-layered volume representations that permit an interactive deferred exploration of volume data. These representations are extracted from the data set by combining similar values along a ray into one layer. By recombining the images opacity changes of individual features can be achieved. The layer extraction and modification of the properties of each layer can be influenced by the user.

By generating proxy images as a deferred representation for volume data [TCM10c], different types of proxy images allow for the deferral of exploratory operations on the volume, as is shown in Figure 2.3. Deferred view changes can be performed through the use of multi-perspective proxy images. The storage of a depth proxy facilitates a relighting of the data set. A deferred transfer function exploration is implemented by accessing accumulated

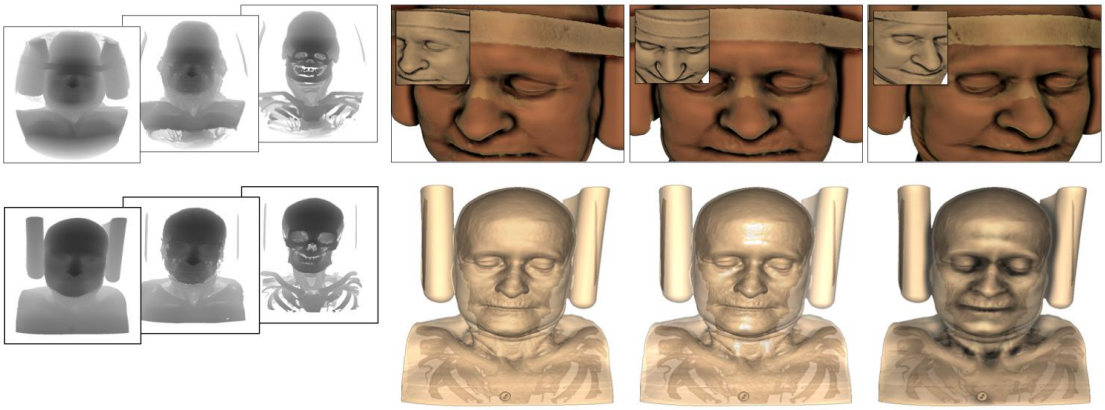


Figure 2.3: Through the generation of proxy images (depicted on the left) in the approach by [TCM10c], common exploratory tasks in volume visualization can be performed without a re-rendering of the scene. On the top row, multi-perspective proxy images incorporate multiple viewpoints of the data set. View changes and ambient occlusion can be estimated through this technique. The bottom row shows the use of a depth proxy that enables interactive relighting of the scene.

Images from Tikhonova et al. [Tik12]

attenuation proxies. These techniques provide an interactive exploration of large data sets without performing potentially expensive volume rendering operations.

Attenuation functions summarize the distribution of attenuation along a ray, which can be used to generate a data representation for dynamic volume exploration [TCM10b] in large-scale visualization environments. The ray attenuation function is a compact representation that nevertheless enables the user to dynamically filter the information on the screen and modify color and opacity values of the visualization. Constructing the ray attenuation distribution recursively from a hierarchy of blocks permits the computation to be performed in parallel [TYCM11], resulting in significantly better performance.

2.2 Remote Visualization

In many visualization systems, the massive sizes of complex three-dimensional data sets combined with the processing power that is needed to calculate meaningful representations from this data place a high demand on the workstations available to the analyzing researchers [SME02, MDM02]. With the emergence of the internet, the partitioning of computational tasks to different machines and the externalization of computationally demanding operations combined with a retrieval of the results became feasible.

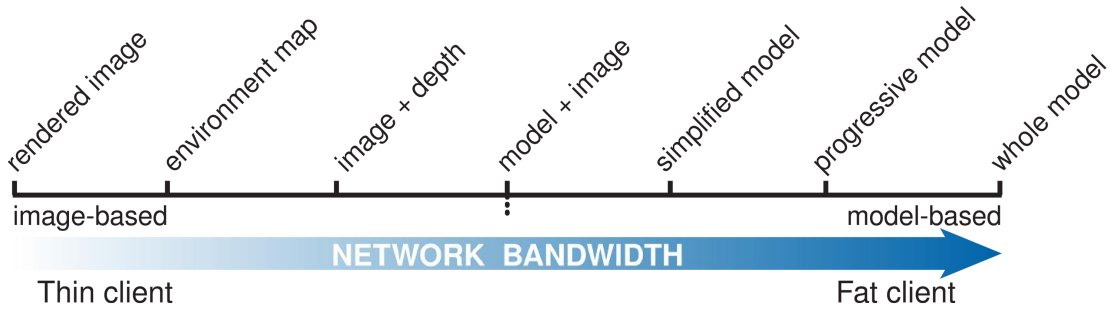


Figure 2.4: A classification of remote rendering systems according to the type of data transferred from the server to the client, as described by Shi et al. [SH15]. Whereas thin clients receive an image stream without additional information on the data, fat clients receive the data set and perform the rendering process largely or entirely on the client side.

Motivation

The motivation behind remote visualization is to keep the memory and processing requirements low on the client side. Rather than trying to equip researchers with a high processing power and large amounts of memory, the ubiquity of web technology suggests that outsourcing the computationally intensive tasks may be a better solution to this problem[EE99, ESE00].

Processing and rendering the data remotely on machines specifically designed for these purposes [BD13] while providing the end user with clients that only display the generated results not only makes it possible for multiple researchers to share the high-performance equipment. Also, whenever these resources need to be updated, this can be done in a centralized way, which greatly reduces the overhead of keeping the visualization system up to date.

Through this setup, it is sufficient for each researcher to have access to a regular workstation instead of having a dedicated high-end machine with specialized hardware designed for computationally intensive tasks. Remote visualization frameworks enable the access of results on mobile platforms that were previously unsuitable for scientific investigation. Especially when working with data sets or simulations at sizes that call for in-situ approaches, the accessibility of the visualization results from regular workstations is of paramount importance.

Furthermore, if the data set lives in a centralized host location, the effort of transferring the data when either the data set is updated or the simulation parameters change can be omitted. Directly related to this, the replication of the data onto the machine of every researcher that is working with the data multiplies the strain on storage space, an issue that becomes particularly severe for large-scale data. The process of moving the bulk of the computational effort to a remote location falls into the category of remote rendering problems [HS98].

Evidently, the network bandwidth and latency aspects need to be taken into account whenever data has to be transferred via a network connection at an interactive frame rate. De-

pending on the amount of data that is streamed – which, based on the underlying architecture, might be anything from a simple image stream to complex models – this can be a bottleneck for remote visualization. For this purpose, the use of multi-threaded asynchronous compression was proposed early on in order to reduce the impact of the bandwidth [SDWE03].

Architecture of Remote Visualization Systems

The design of remote rendering systems can, in general, be classified according to the stage in the rendering pipeline (cf. Figure 1.1) at which the computation is split between the server and the client [WBW96], separating fat clients that perform a large part of the rendering pipeline themselves from thin clients that merely stream the results of the rendering performed on the server. Depending on the type of data that is transferred from the server to the client, Shi et al. [SH15] propose a classification into model-based and image-based remote rendering systems. The model-based systems transfer large amounts of the model data to the client where the rendering occurs. The image-based systems perform server-side rendering and the data being transferred is mostly limited to result images and augmenting data. Since the variety of approaches is very big, the classification from model-based to image-based systems is rather a gradual transition than just a two-class division, as Figure 2.4 illustrates.

When removing all stages of the visualization pipeline from the client, the final rendered result for a user's request is simply sent in an image stream [MPHK93]. While the client is relieved from the computational workload, the flexibility on the client side suffers. Every change in the visualization settings induces a new streaming request from the server. Increasing bandwidth and improved compression schemes make this a viable solution for many settings [MHUnC12].

2.3 In-Situ Visualization

Today we see a continuous and unprecedented growth of data gathered from high-precision measurements as well as generated through large-scale simulation. More and more data are being continuously generated through high-performance computing, with every new generation of hardware boosting the capacities of super computers, thereby enabling the modeling of more complex scenarios. On the other hand, the task of performing a meaningful analysis of the collected information becomes increasingly demanding. The aggregation of data sets measuring Tera- and even Petabytes is already common in contemporary research projects [YWG+10].

Traditionally, the acquired data is stored to disk and, at a later point, retrieved on a visualization platform for analysis and investigation. While the throughput discrepancy between the hardware simulating the data and the data storage was acceptably small, this approach was suitable and it was reasonable to take on the cost of writing data to disk. The process of storing the entirety of the generated amounts of data is, however, prohibitive, at the scale that simulations are outputting nowadays [Chi07]. Additionally, visualization tools are not able to process visualization tasks of petascale data size without initially performing

some simplification of the data in order to expressively visualize the data either at lower resolution than originally generated or in smaller chunks.

Challenges

As a matter of fact, the biggest bottleneck in a traditional simulation-visualization workflow is the storage and retrieval of the data. The application of post-processing operations to the generated data frequently still needs huge amounts of I/O operations. In certain scenarios this approach is not practicable. Data reduction techniques may introduce additional errors that were not existent in the acquired data and a probabilistic selection of data values may not yield satisfactory results, making parallel visualization at massive scale a highly complex task.

Since the architecture of modern GPUs is nowadays flexible enough for them to be used for general purpose computations, the widespread use of GPUs as computational nodes in high-performance computing allows for the utilization of the same hardware for both the computational as well as the visualization tasks. To that end, the concept of running a visualization in parallel to the simulation is referred to as *in-situ* visualization and is currently one of the most important research topics in large-scale visualization [Ma09]. Many large visualization frameworks have embraced this development and offer in-situ visualization integration solutions, like *ParaView* [CGM+06] and *VisIt* [WFM11].

Recently, even browser-based in-situ approaches have been proposed [SSF+15].

In Transit

The limited speed of the disk transfer rate is also a serious constraint for modern supercomputers, where the I/O speed is usually greatly exceeded by its computation rate. The problem of the occurrence of latency while writing the data to disk can be addressed by setting up a separate I/O layer that performs a buffering of the data and subsequently writes it to disk while the computational nodes of the supercomputer continue running the simulation without obstruction. The term *in transit* visualization describes the strategy of performing a specialized form of in-situ visualization in exploiting this layout by intercepting the transport infrastructure that is performing the I/O and using it to additionally perform visualization and data analysis [MOM+11]. For this purpose, an additional set of staging processors can be allocated to communicate the data via MPI (message passing interface) to perform an analysis or processing of the data without interfering with the main simulation. Rather than performing the in transit visualization on specifically assigned nodes on the supercomputer, the architecture of the staging area can also be set up to operate as a separate service while communicating with the client through a low-level network transport.

For in-situ environments, which are engineered to outsource tasks that would greatly transcend the scope of a regular workstation, the workload is split between server and client. The approaches vary between very thin clients that only display pre-rendered images and clients that partially render the resulting images using an intermediate representation. This intermediate representation usually allows for deferred interactions.

For the purposes of this project, a high performance computing system might output complex object representations that would be too costly to compute on a regular desktop workstation, which could then be viewed, manipulated and interacted with on a client. If the object representations could be generated in situ at simulation time, this might be additionally helpful for investigation purposes.

2.4 Visualization in the Browser

Modern web browsers are powerful tools that integrate many well-engineered technologies for accessing content from the Internet. The web browser as an environment for visualization was explored early on [WBW96, MB97, RCMC00]. Recent tendencies to outsource most computation to the cloud while accessing any content from a thin-client interface has shifted the execution of many computerized tasks from standalone applications into the browser. This general shift towards the preference of browser applications for many purposes can be explained due to the attractiveness of browser interfaces for multiple reasons.

The technologies employed in modern web browsers can be used as a framework for visualization tasks. HTML5, CSS3 and JavaScript are standard technologies incorporated by default in web browsers. This makes visualizations easily accessible since the user does not have to concern himself with obtaining, installing and updating the tools for the visualization application. Furthermore, using the browser as the visualization framework guarantees that the application is platform-independent.

Visualization approaches that rely on remote content based on web servers have fewer issues with keeping the data consistent over many workplaces and most servers guarantee protection from data loss.

Tools

Additional tools for specific tasks can easily be incorporated within a web application. Due to the popularity of browser-based systems, many extensive frameworks are available for free.

WebGL [Par12] allows for hardware-accelerated rendering of 3D graphics within the context of the browser without need for further plugins. WebGL is based on OpenGL ES in combination with JavaScript. *Three.js* [thr15] is a free environment for three-dimensional rendering implemented in JavaScript and WebGL.

Especially for information visualization, many toolkits and libraries exist that facilitate the use of browser technology standards for visualization tasks. *Prefuse* [HCL05] is a visualization toolkit that renders to a Java Plugin within the browser. *ProtoVis* [BH09] is a JavaScript library that generates SVG graphics from data. *D3* [BOH11] (short for data-driven documents) is a direct successor of Prefuse and ProtoVis. It is a JavaScript library that is embedded into a web page and capable of generating data visualizations through manipulation of the DOM objects that describe the content of the web page. It supports the scripting of SVG, HTML5 and CSS properties.

Collaborative Visualization

Recent Web-based visualization solutions also aim to incorporate tools for cooperative exploration and investigation of data. Outsourcing the calculation to a remote server aims at reducing the bandwidth limitations when visualizing large data sets[WPJR11]. Marion and Jomier [MJ12] use WebGL to interactively render their data set within the browser and pass the communication between the collaborators through WebSockets.

Visualization on Mobile Devices

The presentation of visualization content in the web browser makes it possible to use devices that were previously unsuitable for this purpose. Mobile devices are omnipresent nowadays and are more and more used as a substitution for workstations. Many application providers have picked up this development and offer solutions that perform processing in the cloud [MHUnC12, SH15] or offer applications that run entirely in the browser.

Accessing visualization from thin clients like mobile devices requires for computationally intensive calculations to be executed on a dedicated server component [TS15].

For thin-client settings, *VirtualGL* [Com07] offers the streaming of 3D rendering commands to a server while streaming the output to the client.

2.5 Rendering of Segmented Volume Data

Object-based visualization approaches require the dataset to be segmented. This process identifies individual regions or objects within the volume, e.g. according to tissue type. After segmented objects have been defined, specific properties such as a separate transfer function, can then be assigned individually to each object. The segmentation step is often the first task after the data acquisition.

2.5.1 Direct Volume Rendering

In scientific visualization, volume rendering is the process of generating a two-dimensional rendering of a volumetric data set [EHK+06]. Such data sets are typically represented as a three-dimensional scalar field. They are obtained through 3D scanning devices, registration of slices of 2D imaging techniques, or through simulation.

Images of volumetric data can be generated with multiple approaches. These can be classified in image-order and object-order techniques. Image-order algorithms start from the result image and gather information per output pixel whereas object-order algorithms iterate over all voxels and map them to their respective output location.

Direct volume rendering is an image-order approach. For each pixel a ray of sight is shot through the volume which is traversed until it exits the volume. Samples along the ray are reconstructed, colored, shaded and accumulated. The opacity and color value is usually specified by means of a transfer function that maps scalar values to RGBA values.

Volume Ray Casting [RPSC99] is a technique that shoots a ray of sight into the volume which is traversed until it hits the bounding box of the volume. All voxels that are hit along the ray are sampled, shaded and accumulated. The composition of the values along the ray depends on the chosen ordering and projection (front-to-back, back-to-front, maximum intensity projection).

Isosurfaces are surfaces connecting voxels within a volume that feature a constant value (e.g. density, velocity, temperature). This constant value is called isovalue. Isosurfaces can be extracted from the volume through dedicated algorithms, most notably Marching Cubes [LC87], which constructs a polygonal model from all voxels within the isosurface. The generated polygon mesh can be rendered through standard rendering techniques.

Splatting [Wes91] is an object-order approach. Every voxel is projected onto the viewing surface in back-to-front or front-to-back order. The splats are represented by disks with varying color and transparency values.

In our system, we use a modified ray-casting algorithm to render segmented volumetric datasets to the object image proxies incorporated in our Volume Object Model.

2.5.2 Segmentation Techniques

The task of classifying data values to a certain object, also referred to as segmentation, can be undertaken in many different ways [Wir07]. The approaches range from fully automated segmentation where all steps in the segmentation process are performed algorithmically via user assisted approaches to manual segmentation by hand in instances where algorithms are not able to identify the correct separation of objects. Evidently, the utilized segmentation algorithm depends not only on obtaining the envisaged segmentation results but also on the nature of the data. A segmentation of meshes calls for different approaches than volumetric or multidimensional data sets. Also, the quality of the data set and the pursued accuracy can influence the choice of the approach.

Available segmentation techniques can roughly be classified into structural techniques which rely on forming a segmentation of the volume by accessing structural information and stochastic algorithms, which, in contrast, perform the segmentation through statistical analysis only. In-between these categories, hybrid strategies try to combine the advantages of multiple methods.

Three-dimensional *edge detection* [Liu77], which is a structural approach, defines an edge at the intersection of two regions with different intensities of specific features. Local edges are thus detected by using differentiation and are then grouped together to form boundary contours separating the distinct regions. Morphological approaches use combinations of the most fundamental transforms of mathematical morphology, erosion and dilation, in order to approximate the shapes of the objects within the volume.

In *graph based algorithms* [MB98], edges and surfaces are depicted as graphs. In order to generate objects from the graph, the algorithm searches for the lowest-cost path between two nodes of the graph. Different search algorithms are used in various fields of science.

Deformable models [KWT88] are described by curves within the volume that deform under the influence of external and internal forces. The external forces in this physically based concept are influenced by the data which cause the model to adjust to the force, whereas the internal forces counteract in trying to keep the model smooth.

Isosurfaces in three-dimensional volumes are defined by faces that connect voxels sharing an isovalue. In combination with the usage of *level sets* [OS88], which are numerical techniques to track the evolution of interfaces, this can be applied to track the contours of objects within the scene.

Thresholding [SSW88] segmentation algorithms can classify objects according to whether the value of a specific data attribute falls below or above a certain threshold. If a single threshold is used, the result is a binary segmentation, but multiple thresholds can also be defined. In volumes with a good contrast between regions, the use of thresholding methods is a very effective approach. However, the technique is very reliant on the finding of a good threshold, which is mostly done manually using visual feedback, although strategies that try to automate the finding of correct thresholds [JMP88] exist.

Classification techniques [PXP00] are derived from pattern recognition approaches. N-dimensional feature vectors computed for each voxel span an N-dimensional feature space. Training data with known pre-segmented labels are fed to a classifier to correctly evaluate the remainder of the data set.

Clustering algorithms can be used to partition a data set into a certain number of clusters where each voxel is assigned to a specific cluster according to the minimization of some distance function. Thereby, the algorithm groups all voxels of similar attributes into an object. The most commonly used clustering algorithm for segmentation is K-means clustering [CA79], where the objects are iteratively grouped into K desired clusters while minimizing the dissimilarity of the elements within one cluster.

Markov random field (MRF) modeling [Li94] is a statistical model that describes the spatial interaction between nearby voxels. This method takes into account that statistically, most pixels belong to the same class as their neighboring pixels. While by itself, it is not a segmentation method, it can be applied within segmentation algorithms.

CHAPTER 3

Remote Visualization with Deferred Object Interaction

When investigating real-world data in a visualization environment, often many objects occur in a scene. Our usage of this term follows the definition of an object, or physical body, as an "identifiable collection of matter in three-dimensional space which lives within a defined

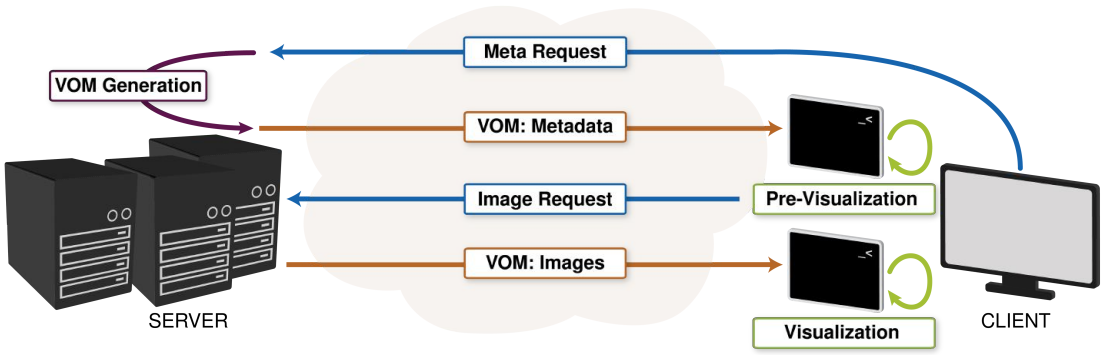


Figure 3.1: This diagram illustrates the architecture of our system on a high level. When the client sends a request for the visualization to the server, the server initiates the generation of the Volume Object Model. The server returns the metadata generated during the rendering to the client. This data allows the client to investigate the objects in a pre-visualization step, without having access to the images yet. Tasks that can be performed in this step include filtering and information visualization on the metadata. In a next step, the client requests the full visualization from the server, at which point the server streams the images. Based on the images, the client can reconstruct the visualization, which can be programmed on the object level.

contiguous boundary and may be more or less constrained to move together"[1].

The architecture of our object-based remote visualization solution will be presented in this chapter. We propose the introduction of a *Volume Object Model* (VOM) as an intermediate stage in the volume visualization pipeline. Furthermore, we discuss the design of a remote visualization architecture that employs this model in a similar way as a *visualization by proxy* [TCM10c] approach. The Volume Object Model allows for a deferral of the user's interaction with individual objects in a segmented volumetric data set in order to manipulate the data set on the object level in a thin client visualization system.

This solution is particularly advantageous if applied in an in-situ visualization scenario, where the large quantities of data amassed in an HPC environment need to be rendered into a suitable representation in order to be transmitted to a regular workstation and to be made accessible to the researcher. The following sections describe the main conceptual notions of our approach as well as the methods utilized in our implementation. A comprehensive description of the technical implementation details of our system will be presented in Chapter 4.

3.1 Remote Visualization Architecture

Remote Visualization aims at transferring the computationally demanding tasks from the client to a remote server. Many remote visualization systems are designed to perform all calculations on the server which compromises the client-side flexibility. We describe our architecture which proposes a deferred visualization pipeline to allow for client-side object manipulation.

Remote Visualization with Deferred Interaction

Our concept is based on the notion that many use cases in visualization demand for interactivity in the context of object-wise interaction and manipulation. We therefore reduce the visualization of our data set to a two-dimensional representation while still preserving interaction capabilities on the object level at a later point in the pipeline. We call this intermediate object representation that is transferred from the server to the client Volume Object Model. The client provides interaction methods with the individual objects within the scene. By providing the user with a scripting console we maximize the object manipulation possibilities. The user types in commands that are interpreted interactively and manipulate both the appearance and behavior of the object representations. Since many tasks carried out on sets of objects can be broken down into a few core operations, the user can program a multitude of object-related queries and modifications and apply them to the visualization.

In contrast to pure remote visualization the client can render the final image from an intermediate representation. This is especially desired when the rendering step on the client is computationally cheap and the computation of the intermediate result is expensive. For instance, the computation of object properties as well as spatial relations between objects is relatively expensive. These object-level properties can be stored into an intermediate

[1] Definition from "Physical body", *Wikipedia, The Free Encyclopedia*

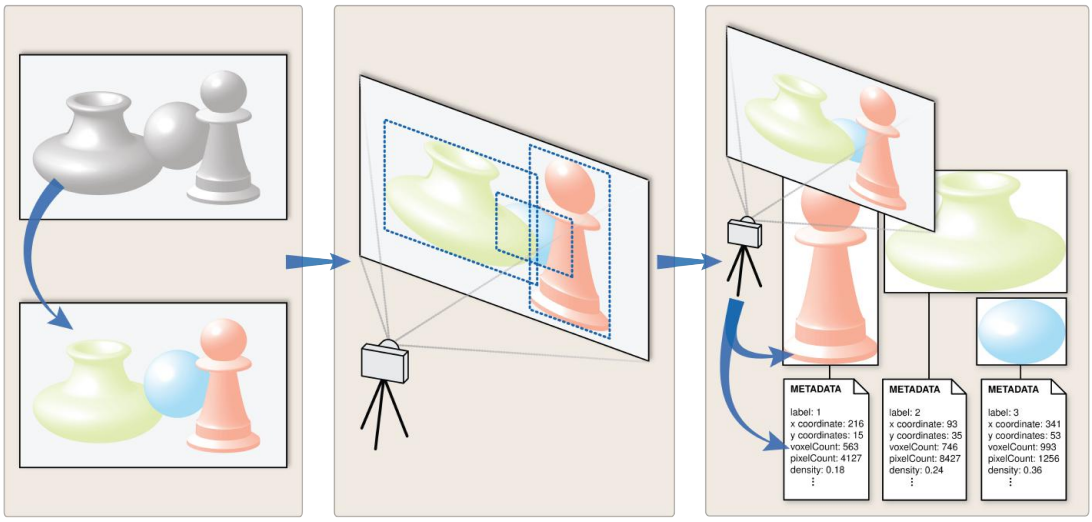


Figure 3.2: The server side of our system consists of several steps: First, the volume is segmented into a number of distinct objects (left). When the objects are determined, their boundaries in screen-space are evaluated in a first traversal of the volume (middle). The boundaries define the size of the render targets for each object. In a second traversal of the volume, each object is rendered to its target (right). Furthermore, metadata is accumulated about the objects. The visualization and the metadata together constitute the Volume Object Model. The execution of each of these steps is dependent on the result of the previous one.

representation. Deferred interaction on the client side is then possible based on these object attributes.

3.1.1 Overview

The design of our system as shown in Figure 3.1 was created for an in-situ visualization scenario, where a powerful HPC environment serves the computationally intensive requests of the client. In order to comply with the entailing prerequisite of a thin-client visualization setup, we designed a system which shifts the bulk load of the computational effort to the server side while still preserving extensive adaptability for manifold object-manipulation tasks on the client side. We believe this configuration results in a good trade-off between client-side flexibility and an outsourcing of the processing performance to a suitable remote device.

Our approach proposes the integration of deferred object manipulation capabilities within a client/server-framework communicating via a socket-based streaming solution. For the demonstration of our concept, we implemented a complete remote rendering system that introduces the Volume Object Model in order to allow for client-side object manipulation capabilities. To that end, a volumetric renderer for the server-side pipeline was implemented within the context of the VolumeShop [BG05] visualization framework. A client that can

be accessed through any web browser capable of displaying the transferred data as HTML5 content was developed as a user interface for the system.

3.1.2 Server

The rendering process executed on the server side of the application prepares the data and generates the Volume Object Model from the data set. A high-level illustration of the server's pipeline is shown in Figure 3.2.

As a first step, the server initiates the segmentation of the volume, unless the input data is pre-segmented. For this purpose, any appropriate segmentation algorithm can be implemented.

The server outputs the Volume Object Model, which consists of the visualizations of the objects and the corresponding depth buffers. Additionally, metadata is extracted for each of the objects during the traversal of the volume.

The server instance that performs the rendering of a data set on request can be executed on any device. On execution, it opens a socket and listens to incoming connections. Whenever the client requests data from the server, it processes the request and sends the resulting data back through the socket.

3.1.3 Intermediate Representation

The Volume Object Model facilitates the generation of proxy representations for every object within our scene. These proxies have reduced dimensionality, but allow for a scene recomposition on the client side.

The Volume Object Model representation of the objects consists of a representation of each of the objects rendered to a region defined as their minimal bounding rectangle. Every object is rendered from the current viewpoint, but independent of occlusion. A depth layer is also included into the representation for each object. In addition to the two-dimensional proxies, metadata is generated in order to allow for a recomposition and for meaningful query and interaction methods on the client side.

This metadata includes both view-dependent as well as volume-dependent object properties, respectively. The data can be object-wise information or data about inter-object relationships.

View-dependent properties include the screen-space bounding boxes, the average depths and an occlusion matrix. The occlusion matrix contains information for every object which other objects it (partially) occludes. These view-dependent properties need to be updated whenever the viewport changes.

View-independent metadata consists of properties like voxel counts, view-independent relationship matrices and arbitrary object-wise features such as density or temperature. An example for a view-independent relationship matrix is a distance matrix, which stores

the minimal distance between two objects in the volume. It is sufficient to evaluate these properties once.

The construction of the visualization images contained in the Volume Object Model requires two passes over the volume. This is necessary because we need to evaluate the size of the objects in screen-space before we can create the appropriate render targets. These areas are filled when a second pass over the volume renders each object to the dedicated target. The view-dependent metadata is also calculated in these passes.

3.1.4 Client

Figure 3.3 gives a general overview of the steps the client performs before the visualization is provided in the viewer. The client is implemented in HTML5 and JavaScript, providing a console for user scripts which are executed interactively.

This strategy makes the environment highly customizable and gives the user complete freedom to modify the visualization, thereby allowing for a variety of object-related manipulations that can be scripted interactively through the D3 JavaScript library.

If the browser-based client requests a visualization from the server, the object properties are transmitted over the network to the client. Without transferring any actual image data yet, the client constructs a pre-visualization solely from the metadata. This pre-visualization step allows for the filtering and investigation of object properties. The request for the object visualizations stored in the Volume Object Model can limit the bandwidth usage to the targeted subset of objects.

The scriptable environment allows for queries, filtering and information visualization tasks to be performed on this metadata. The user can program this pre-visualization, e.g. show points as placeholders for all objects, visualize the metadata using techniques such as scatterplots by mapping each object's position to a scatterplot using selected object properties, and perform filtering operations on the objects.

When the user is satisfied with the operations he made on the data, the query is sent to the server. The object proxy images that the server transmits in response correspond to the filtering of the objects specified in the pre-visualization. Through this strategy, the user can limit the amount of data before it even gets transferred.

Only after the requested object images are received, the client-side rendering recomposes them into the visualization. Like in the pre-visualization, the objects can be programmed interactively in the visualization step.

3.2 Volume Object Model

We propose the Volume Object Model as an intermediate volume representation in a visualization pipeline for segmented volumetric data. This intermediate representation can be transferred from the server to the client and enables the client to fully reconstruct the scene from the two-dimensional object visualizations provided within our model.

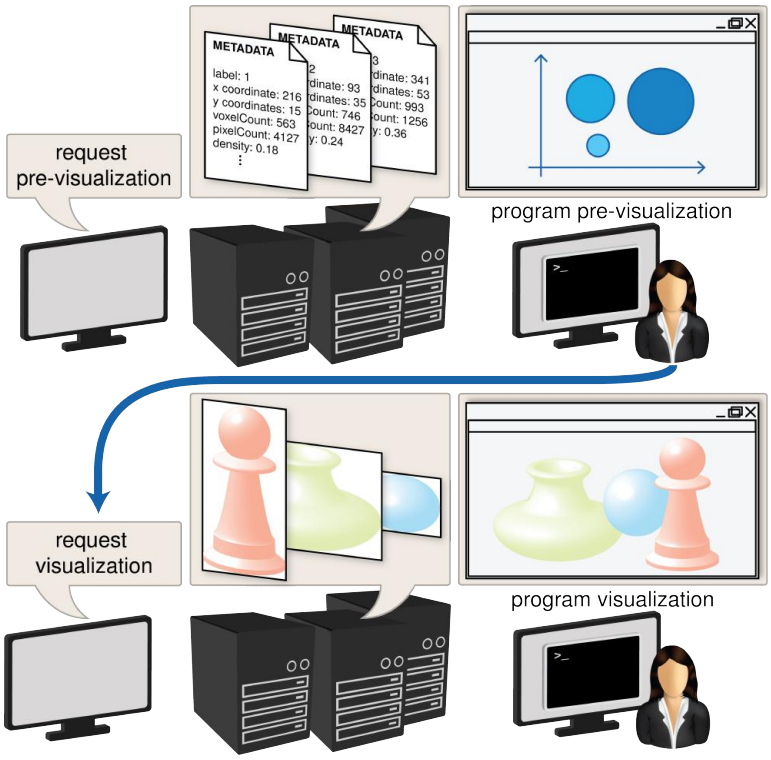


Figure 3.3: The steps in the client-side visualization pipeline: First, the client requests the pre-visualization from the server. This request triggers the server-side rendering, after which the server returns the metadata to the client. As soon as the client receives the metadata, the user can perform pre-visualization tasks on this data. This includes investigation using information visualization techniques as well as filtering of the objects. When the user is satisfied with the results, the images are requested from the server. As soon as the images are transmitted, the client can then reconstruct the original visualization from the images and program object-level manipulations via a scripting interface.

3.2.1 Overview

This section will outline the motivation for the concept we devised as well as the details of the steps to output an object-proxy representation for a data set. We describe how the visualization and the metadata contained in our Volume Object Model are constructed. Lastly, we will provide a description of how the volumetric representation of the object can be reassembled and what modes of object interaction our model allows.

3.2.2 Objects in Volume Data Sets

When investigating large-scale volume data sets, often the nature of the data is such that it can inherently be divided into a number of distinct objects. Subject to the level of detail of the data set, multiple elements of the smallest data unit will frequently be related to each

other in that they share common properties, belong to the same structure or are contained within a joint surface. If one is manipulated, the rest of the components within an object will be constrained to react as well. These objects might, for instance, be different types of tissue in a medical data set, distinct components of a technical device in an industrial data set or delimited strata in a data set for geoscience. Depending on the research field, data sets may consist of few (e.g. $< 10^2$ objects in medical data sets) up to many (e.g. $\sim 10^6$ objects in material science data) distinct objects.

Segmentation

In order to access the objects contained in a volume, rules for the distinction between individual objects within the data set need to be applied first. As described in Section 2.5.2, a variety of approaches exist for the allocation of data values to a certain object.

The choice of segmentation algorithm for a specific data set depends on many factors. The nature and quality of the data set needs to be taken into account and the targeted result has to be analyzed in order to select an appropriate algorithm. Since every algorithm has strengths and weaknesses, often hybrid approaches are applied in order to obtain the best possible result. Also, the quality of the segmentation depends heavily on whether the data set contains a lot of noise, missing or corrupted values, which may impair an accurate segmentation. Since the segmentation widely varies also with the scientific domain, our system is not focused on one particular segmentation algorithm. The segmentation is an interchangeable module in the design of our system. However, each algorithm comes with a set of parameters and possibly with user interactions.

Our system exposes the segmentation step to the user with a programming interface. The user can specify a predicate that either evaluates to true or false for each voxel using a domain specific language. We determine the objects within our data set by performing a connected-component labeling of the voxels contained in our volume. This process assigns an integer to every voxel which designates the label of the object it belongs to. The background (i.e. non-segmented parts of the volume) are assigned the label zero.

Object Specification and Object Attribute Computation

Our system uses a domain specific language (DSL) for the object specification and the object attribute computation. The user specifies what constitutes an object and how to compute attributes of objects. These programs are entered in the console of the client. The client then transmits the program to the server. The used DSL is ViSlang [RBGH14] which translates the object specification and attribute computation code to OpenCL code. The server efficiently executes the OpenCL code on heterogeneous architectures using multiple processors or GPUs scaling well to large data sets.

```
predicate inRange[voxel v] (float fMin, float fMax) {
  return ((v.value >= fMin) & (v.value <= fMax));
}
server.inRange(0.5,0.75);
```

Listing 3.1: *An example for ViSlang object specification code*

A predicate *inRange* (cf. Listing 3.1) is defined and evaluated over all voxels in parallel. It evaluates to a Boolean value for each voxel resulting in a binary volume. Using a connected component algorithm the binary volume is transformed into a label volume where each binary region gets assigned a unique ID. These regions constitute the volume objects.

The user can assign attributes to the volume objects by using the DSL.

```
/* declare mapping function for parallel reduction that allows counting
    of voxels in a region */
integer isInRegion[voxel v](voxel vox, integer id) {
  if (vox.value == id) {
    return 1; // return 1 if voxel value matches id
  }
  return 0; // return 0 otherwise
}

object attributes:
integer id = label; // make the id part of the objects attributes
/* compute voxelCount of each object using a parallel sum reduction */
integer voxelCount = sum[voxel v in volume]isInRegion(v, object.id);
```

Listing 3.2: *An example for ViSlang object attribute computation example*

First a mapping function *isInRegion* is defined as shown in Listing 3.2. This function returns 1 if the voxel value matches the object id and 0 otherwise. To define the object attribute *voxelCount*, the mapping function is evaluated in parallel for each object ID. The sum of all voxels that are part of the region equals the voxelCount.

Object-Level Manipulation

While each of these objects may be of arbitrary size and consist of many different voxels, frequently scientists wish to treat the entirety of such an object as one entity and manipulate the data on the object-level rather than on the lowest granularity of the data set. Both real-world and simulated data sets will usually consist in large parts of regions that are of low interest to the researcher, thereby constituting the background. Through object-based approaches, scientists are able to investigate individual objects of their data set in detail while either completely masking out the rest or keeping an unobtrusive representation of the other objects in order to provide some context within the data set.

In multivariate data sets, object-centered approaches can provide the ability to combine the attributes of individual objects into combined features for easier comparison with other objects. Operations on the object-level allow the researcher to query the data set according to object features, highlight objects with similar properties or display the objects with different opacity values according to their importance.

If the spatial context of the individual objects is not relevant, the objects can also be rearranged into views that facilitate an expedient investigation. Examples include exploded views, sorting by a certain attribute, resizing, or other layouting criteria.

3.2.3 Object Model Computation

The central contribution of this work is a novel object model representation that enables a segmented data set to be rendered on the server side into an intermediate object representation. This representation consists of object attributes and 2.5D images.

To that end, each distinct object within the volume is rendered to its individual representation, which happens independent of occlusion and culling. The object representation allows for user interaction and object manipulation as long as the viewpoint of the scene remains unaltered. In our implementation, this representation is generated through the application of a process that can conceptually be divided into three steps, during which two ray-traversal passes over the volume are necessary. Refer to Figure 3.2 for an overview of our server-side computation pipeline.

View-Independent Object Attribute Computation

Our architecture is designed to handle an arbitrary number of objects. The actual number of objects is determined by an initial pass over the scene geometry. This pass counts the number of heterogeneous objects within the scene, which is constituted by the previously executed segmentation step.

In the pass over the scene, when evaluating the label value of every voxel, we not only count the number of distinct objects but also count the quantity of voxels associated to each of the objects.

To select a subset of objects from the total number of objects in case the network bandwidth or the computational power do not support a presentation of the full number of objects, the user can query and filter these object attributes before requesting the object proxy images from the server.

View-Dependent Object Attribute Computation

Since we want to store every object's two-dimensional rendered output image to a dedicated rectangular area, we need to determine the size of every object's screen-space bounding box. This view-dependent object attribute is evaluated in the first render pass of the implementation. For this purpose, an object tracing kernel traverses the entire volume. Other view-dependent properties can be added to the metadata in this step of our process.

We perform direct volume rendering on the GPU via a modified ray-casting technique. For every viewport pixel, a view ray is cast into the scene, passing through all the voxels in the scene that contribute to that specific pixel when looked at from the current viewing position. This ray starts at the eye point and is defined through a spatial vector denoting the direction of the ray. Then, the ray is traversed step-wise from the volume entry position to the volume exit position.

In standard ray-casting, the color value of the pixel is calculated by traversing the ray until the composited color reaches an opacity value of 100%, at which point the ray is terminated. In contrast, our modified approach always traces the ray until it exits the volume, determining

all objects that are intersected. At each step, the current position along the ray is computed and rounded to the nearest voxel in the volume, the value of which is then considered as lying along the ray. Whenever a voxel is hit, its label and therefore its affiliation to an object is determined. If the value differs from that of the previous voxel (which designates it as belonging to the front face of an object), the current position is appended to a data structure storing all the voxels that were hit along this ray.

After the ray has terminated, all encountered objects are evaluated. For the screen-space bounding box evaluation of every object that was detected along the ray as seen from the viewport, the minima and maxima values of each of the objects are compared to the current pixel's position. If the voxel's projection to the viewport lies outside of the current bounding box, the values are updated to the current location. An analogous technique is applied for finding the minimal and maximal depth values of the object during ray traversal.

Object Depth Computation We calculate an average depth value for each object, which can be used for a simplistic depth sorting on the client. For complex scenarios, however, the object-wise depth value is not sufficient. Therefore, we also store the depth data to a screen-space buffer. This measure increases the size of the buffers but allows us a correct reconstruction of the visibility of the objects even for convex or intersecting objects.

Metadata Assembly

Since we want to not only output our image buffer, but also make the necessary metadata available to be utilized for object-related calculations on the client side, we can integrate the corresponding calculations into the render pass. Aside from the indispensable metadata, which is needed to reassemble the objects into an accurate scene representation, depending on the use case and the nature of the data, we can append additional characteristics. The way in which we define the metadata requires object-wise properties which can be used to define our objects in more detail and can be calculated either in the course of the object tracing pass, render pass or in a separate calculation step. This additional metadata can consist of object-wise attributes like statistical measures (e.g. averaged features from multivariate data sets like density, temperature or pressure values per object), geometric properties (e.g. elongation, eccentricity) or topological properties (e.g. holes and tunnels).

Object Relationship Matrices In the course of the render pass, we can also calculate inter-object relations. Such relationships are stored in matrix form, each entry specifying the value of a specific property for a single object-object-pairing. Typical examples of object relations are: occlusions, partial occlusions, distances, accessibility (i.e. is there a path between two objects), similarity, correlation, etc.). Our approach aims to give our user full access to program these relationship attributes from the client-side using DSLs.

Metadata Storage Our approach allows for an arbitrary number of metadata entries to be stored for each object. Some of these properties may be stored object-wise while other describe object-object-relations. However, we define some metadata properties that are

indispensable in order for our Volume Object Model to be reassembled. Therefore, first and foremost, we need to store the information for client-side reassembly. We will show that these attributes are sufficient for a straightforward object recomposition performed by the client for simple scenarios.

The first of these properties is a list of all the object labels included in the scene, which is attached for reference. The attributes for the composition of the object layers can be divided into scene-related data and image buffer-related data. As for the scene, we need to store the minimal bounds for each object within the scene in x- and y- direction as well as the width and height of the objects in the scene. The data needed related to the image buffer consists of the location of the object's render target and the width and height in the output buffer. We choose to normalize all of these properties to the $[0, 1]$ range to facilitate a rescaling on the client side. Therefore, we also transmit the dimensions of the scene as well as the dimensions of the generated output. Furthermore, the average object-wise depth values computed in the render pass are appended to the metadata.

3.2.4 Object Model Transmission

A server instance keeps a socket open listening to requests. The client, which runs in a web browser, can then access the host's address. The user is first offered a list of previously stored sessions. If a session is selected, the client opens the viewer interface. After the server loads the volume, it generates the output and sends the rendered image as a response through the socket. The client also requests the metadata, which the host also transmits. Our modified client does not instantly display the streamed image in the viewer but waits for both of the responses to arrive and then initiates the recomposition of the scene.

3.2.5 Client-Side Scene Reassembly

On the client's request, we first transmit the object attributes from the Volume Object Model without the image data. By giving the client access to the metadata, we allow for a pre-visualization mode. The user solely interacts with object attributes and is able to filter and investigate them. Since the environment is fully programmable, the user can assign each object a placeholder (e.g. a dot) and create visualizations with these placeholders.

Only after the user is satisfied with the outcome of his pre-visualization, the actual visualization data is requested for the filtered selection of objects. On the web client, the two-dimensional object representations are received and need to be evaluated and recomposed in order to arrange the objects into their original positions within the input scene. The client triggers a repaint of the viewer window as soon as the image stream has finished transferring from the server.

The goal of our approach for the client-side object reassembly is to reconstruct the entire scene from HTML5 elements. Besides the fact that by rendering the scene into an HTML representation, any modern browser can display it correctly and no further software needs to be installed on the client, we also consider it a very user-friendly solution. This design is motivated by the idea that the appearance and behavior of HTML5 DOM elements can easily

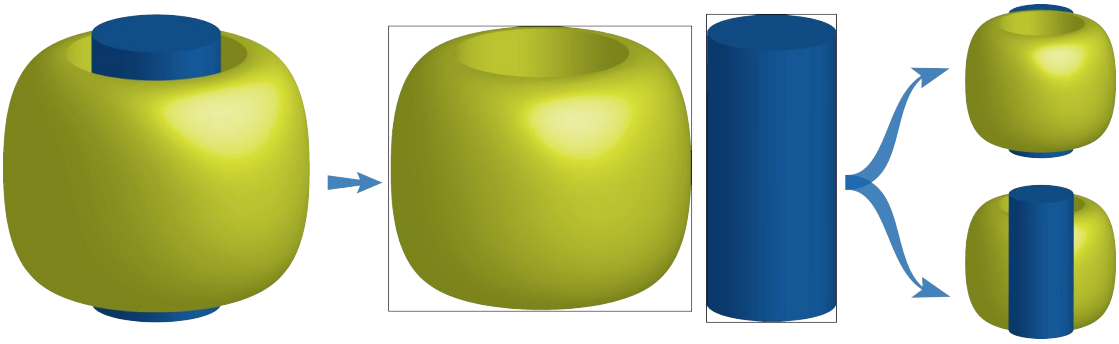


Figure 3.4: An illustration for a depth issue that can occur whenever non-convex objects interpenetrate. In such instances, each of the objects partially overlaps the other object. If both objects are rendered to a flat representation, a simple layering approach is not sufficient to guarantee correct visibility.

be accessed and edited even by non-computer scientists using standard web technologies - the JavaScript, CSS and HTML languages. We therefore generate our object representations as HTML5 canvas elements and provide a console in our viewer which allows for an editing of these DOM elements.

Object Reassembly

The resulting two-dimensional object representations need to be arranged in the viewer according to their original position within the scene, which is looked up from the supplemental metadata and mapped to the client's viewer size.

We write every object to a separate HTML layer in order to make these layers programmable through JavaScript. The metadata can be attached to the layer as attributes, which can be queried through the respective string descriptor.

The most crucial task is to ensure a correct depth sorting of all of the objects since we reduced their representations to two-dimensional layers. In order to arrange the objects in an approximate z-ordering, the canvas is assigned the averaged depth value. This strategy results in a correct reproduction of the original object arrangement for scenes with non-intersecting convex objects.

Per-Pixel Compositing

For many purposes in the process of assembling the scene from the individual objects, the object recomposition can be done by arranging the layers containing the individual objects in the correct position in x- and y-dimension and adjusting the z-index of the layers in order to compose the objects in the right depth ordering. However, for scenes with non-convex objects or intersecting objects it is impossible to recompose the scene by two-dimensional layering. A more sophisticated approach to combine the objects is required in this case. Figure 3.4 illustrates the depth-sorting problem. Another way of looking at the issue of complex depth

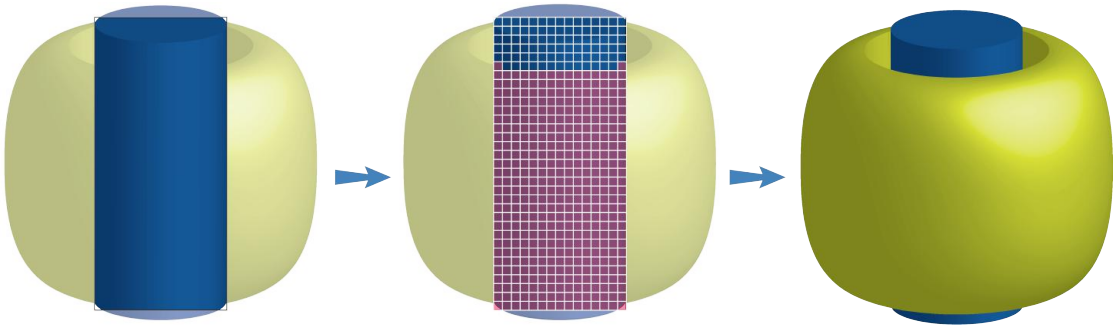


Figure 3.5: For scenarios where two objects overlap each other a pixel-wise depth test is performed. If one pixel of the front object has a depth value that actually indicates that this pixel lies behind the back object, it is hidden from the representation. This method increases the calculation overhead on the client, however, it ensures that correct depth is rendered even for complex objects, which could not be obtained with the z-layering approach.

is to figure out for two objects whether those occlude each other. If both objects overlap each other, neither of them can be completely arranged in front of the other through simple layering without causing incorrect occlusion values.

Whenever two objects intersect, their representations would need to be either cut into smaller parts that can be set to visible and hidden according to the intersection, or the objects need to be traversed to determine which pixel is visible. The first option has great disadvantages – for every object and camera modification, the subsections of the objects would have to be updated since different parts would become visible or get occluded. We therefore decided to implement the second approach.

Checking every object pairing in order to determine whether they partly occlude each other would work, but is too costly for the client side. We therefore compute and transmit an object occlusion matrix as part of the metadata. The viewpoint-dependent object occlusion matrix stores information about whether one object is fully or partly occluded by any of the other objects. A partial occlusion does not necessarily imply that reciprocal occlusion occurs in the object pairing. Therefore, the occlusion matrix has to be checked in both directions: If Object x occludes Object y and Object y occludes Object x, we perform pixel-wise z-comparison.

Since we can determine through the use of the occlusion matrix, whether two objects overlap, we can reduce the overhead of checking all object pairings to the few that are actually relevant. In a case where both objects overlap each other partially, it is necessary to check for overlaps on a per-pixel basis in order to filter out wrong occlusion values. To this end, we need to determine the area of overlap of the two canvases and compare the depth values from the object-wise depth buffer within the overlap, as shown in Figure 3.5.

The pixel-wise comparison increases the client-side overhead, which is why for highly complex scenes with many interleaving objects, the calculation effort may rise in the course of the recomposition.

3.2.6 Deferred Object Interaction

Both in the pre-visualization step and in the actual visualization, the programmability of our client framework gives the user the freedom to modify and query the objects. We describe the techniques of interaction that our user has access to in the visualization composed from our Volume Object Model.

Modes of Interaction

Scientists often perform a series of exploratory operations in order to optimize the presentation of the data for their use case. Frequently users will try to detect the objects and regions that they are interested in, will then zoom to the areas of interest and try to distinguish the interesting data from the context.

Additionally, if the exact placement of the objects within their context is of low relevance, the users may want to alter the data set in order to improve the visibility of specific objects, to arrange them by different attributes to better understand relations between them. Many of the tasks that scientists typically perform can be reduced to fundamental object-level operations, most notably selection, filtering, sorting and rearrangement.

Selection We assign our objects their label as a unique ID (`#objectN`) in order to be able to address them directly in a query. Furthermore, all objects are assigned a common class tag (`.object`). Figure 3.6 illustrates how the DOM objects displayed on the client can be interacted with by addressing them by their handle. Users can add additional class tags to a subset of objects which makes it possible to address them as a group.

To that end, we can either access objects individually or write functions to perform operations on a group of objects. Scripting a command that is executed on one object can be performed by accessing it by its identifier `#objectN`. If we want to perform an operation on multiple objects simultaneously, we need to group these elements by assigning a custom class or through a selection and filtering operation. Then, we can apply our function through either the JQuery method `$('.myObjectGroup').each(function(){})` or D3's equivalent. D3 selectors `select` and `selectAll` inherently loop over all elements that the selector applies to and performs the specified operations. For applying custom functions to selections in D3, the function `d3.selectAll('.myObjectGroup').each(function(){})` can be used.

For instance, in a medical data set, for every type of tissue, the corresponding objects can be grouped into a specific class. This allows for the bones, skin, muscle, etc. to be addressed via their class. The selection of all objects that were previously assigned the class tag "bone" is scripted as follows:

```
d3.selectAll('.object') // select by class 'object'
  .filter(function(d){ // filter by specified function
    var elem = d3.select(this);
    return (elem.density > min_density && elem.density < max_density);
  })
  .classed('bone', true); // assign class 'bone' to filtered objects
```

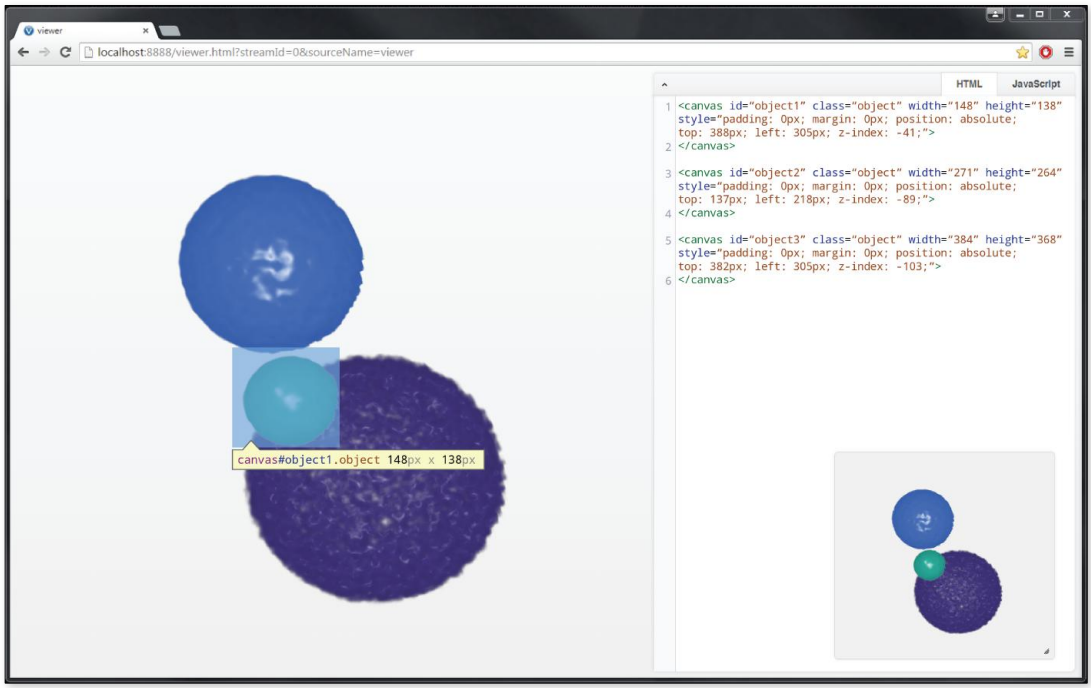


Figure 3.6: Each object is assigned a unique ID according to its label. This allows every object to be addressed by its handle from JavaScript and D3. Most modern web browsers offer tools that are able to highlight the elements within the web page and show a detailed description about the element. In this image, one of the objects is highlighted. We can see that it is a `canvas` *object with ID* `#object2` *and class* `.object` *of size* 216×205 *px.*

```
var bones = d3.selectAll('.bone'); // select by class 'bone'
```

Filtering One of the most important instruments in the exploration of any kind of data set is filtering. Segmented data sets are particularly suitable for the application of filtering operations since a classification of the data into objects makes the object-wise querying of properties much more convenient. As most real-world data sets contain large quantities of less relevant information that can be either masked out or set visibly distinct as background in order to provide context in an unobtrusive way, filtering is a crucial tool to figure out areas of interest. Furthermore, filtering operations can be used to visually encode objects with attributes according to some property, which makes it easy to distinguish interesting characteristics at first glance.

A filtering operation can usually be described by the function that specifies which subset of objects is relevant for the operation and by the task that should be executed on the objects that fall within this subset.

A simple example for a filtering operation might be
"*Select all objects whose temperature lies between 10 and 30 degrees and set opacity to 50%.*"

Such an operation is implemented within our framework with a function:

```
d3.selectAll('.object') // select by class 'object'
  .filter(function(d){ // filter by specified function
    var elem = d3.select(this);
    return (elem.temp > 10 && elem.temp < 30); //filter function
  })
  .style('opacity', .5); //adjust style of resulting set of objects
```

Analogously, more sophisticated filtering functions that query data from the parameters attached to the objects can easily be implemented in few lines of code. There are many possibilities as to what kind of visual distinction can be applied to filtered objects in order to either highlight them or make their appearance unobtrusive. Parameters that can be edited include HTML attributes like opacity (and visibility) and borders, but D3 also allows for the coding of more complex appearance modifications.

Another example for more complex object filtering is shown here:
"*Select objects with a voxel count greater than the average voxel count and display in blue color.*"

This operation requires us to first evaluate the average voxel count, then modify the objects that correspond to the query accordingly. For more complex appearance modifications like this one, we delve a little deeper into CSS3. JavaScript also allows us to append new elements to the DOM in the course of our operations.

```
/* select all objects and store the voxel count to a list */
var numVoxels = d3.selectAll('.object').attr('data-voxels');
/* calculate the average voxel count */
var avgVoxels = d3.sum(numVoxels) / numVoxels[0].length;
d3.selectAll('.object') //select by class 'object'
  .filter(function(d){ //filter by specified function
    var elem = d3.select(this);
    return (elem.num_voxels > avgVoxels); //filter function
  })
  .append('div') //append a new DOM element to the object
    .classed('layover', true) //assign a common class
    .style('background-color', 'rgba(0, 100, 170, 1.0)') //assign color
    .style('mix-blend-mode', 'color'); //set the blend mode to color
```

The HTML5 canvas furthermore offers a straightforward text rendering functionality through which arbitrary information can be shown as text on top of each of the objects.

Additionally, filtering may be done not only attribute-related but also on an object-relation basis. That is, objects may, for instance, be filtered according to their distance to another object using a distance matrix incorporated in the metadata. To this end, once again, D3's extensive capabilities allow for the generation of custom filter functions in very few lines of code which makes the use of JavaScript for our purposes very powerful.

Rearrangement Aside from modifying the visual appearance of our objects, if the preservation of the spatial location of the individual objects of interest is not paramount, we can just as easily rearrange the objects as we altered their visual attributes. This approach can be useful in many visualization scenarios. A simple example for such a use case would be if one object is occluded by another object of interest, in which case the user may benefit from rearranging them in order to see both objects at the same time.

Furthermore, well-known visualization techniques have already proven that oftentimes, it can be beneficial to reorganize the objects within a scene in order to get a more meaningful representation. These applications range from the generation of exploded views to a clustering of the objects by chosen attributes rather than arranging them by their original spatial location. Moreover, the objects can also easily be resized by accessing their height and width properties.

3.3 Summary

In this chapter, we have shown on the basis of several exemplary application tasks that our approach provides powerful tools for the interaction with objects within a visualization and exploration environment. Our technique runs fully interactive in the client's web-browser without requesting new data from the host unless the viewpoint of the scene changes, at which point a re-rendering of the data set is triggered on the server.

CHAPTER 4

Implementation

This chapter aims to provide a detailed discussion of the development and technical details of the remote visualization system that was implemented during a research project at the King Abdullah University of Science and Technology.

4.1 Framework

For the implementation of the system we used the VolumeShop [BG05] research framework. The VolumeShop framework is designed to be easily extensible through the implementation of plugins. We created a rendering plugin within the context of the application that accesses a segmented data volume and outputs the Volume Object Model representation.

VolumeShop also provides a simple client/server-interface for remote visualization. This environment executes as a command-line server application, which is run on the host computer. The server instance then communicates with the client via sockets. We used this setup as a basis for our own client/server implementation and built our web client on top of the existing framework by modifying and extending it.

For the extension of these frameworks and the implementation of a new renderer within the context of VolumeShop, the C++11 programming language was used and built with the Visual Studio 2013 compiler for 64-bit platforms. For rendering on the GPU, we implemented OpenCL kernels in the OpenCL 1.2 standard. The web client makes use of the HTML5, CSS3 and JavaScript (with the JQuery [JQu15] and D3 [BOH11] libraries) technologies in order to be executed within modern web browsers.

4.1.1 VolumeShop

The VolumeShop framework by Bruckner et al. [BG05] provides a complete interface for illustrative volume visualization. It was used as the main environment for the implementation of this thesis project.

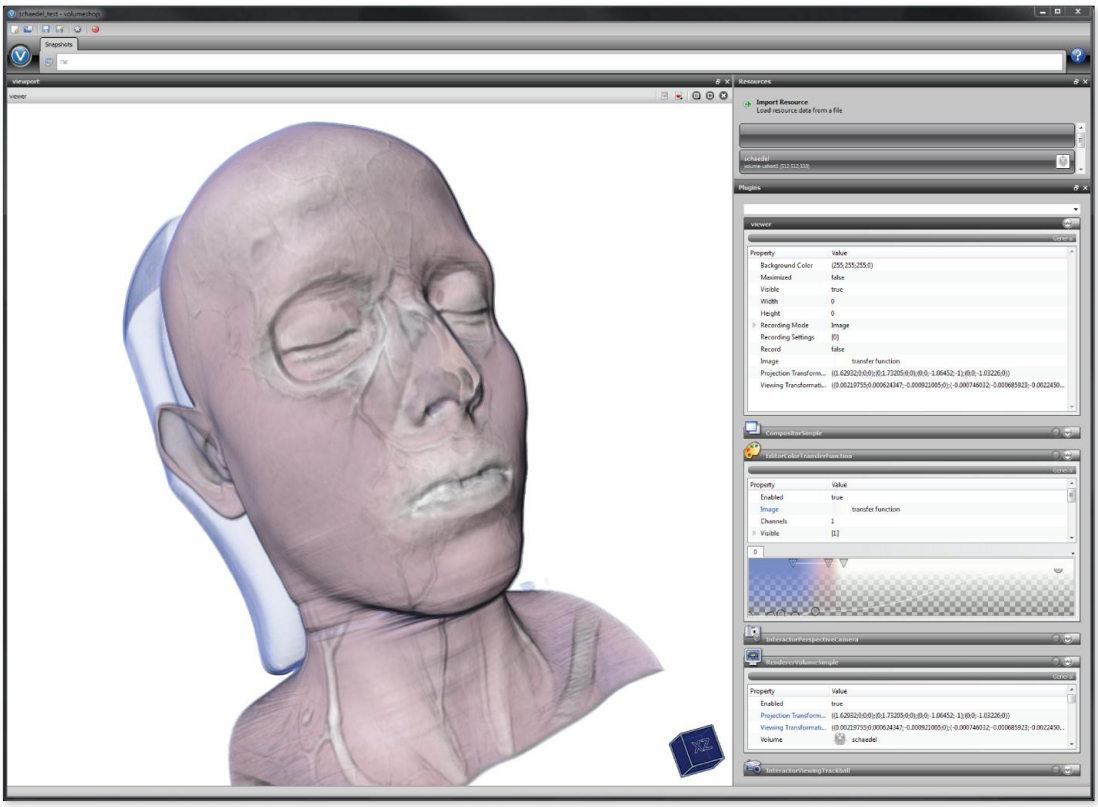


Figure 4.1: This figure shows a screenshot of a VolumeShop session. A 3D view of the loaded dataset is depicted in the prominent viewer window on the left side. If desired, this view can be interactively rotated and zoomed via an interactor plugin, which is signified by the blue cube at the bottom of the viewer window. To the right of the viewer, property windows allow for a detailed specification of the settings of the different plugins used in the current session.

VolumeShop was designed as an interactive standalone solution for illustrative visualization tasks. It gives the user the flexibility of choosing between different renderers and visualization techniques in order to devise meaningful representations of their data set. As such, it also offers approaches for interacting with both segmented and unsegmented volumes. The interaction techniques VolumeShop offers for visual exploration and analysis include, among others, cut-aways, ghosting and exploded views.

Within the VolumeShop application, scientists can navigate and explore their data dynamically and in real time via a three-dimensional viewer interface directly operating on the volumetric data set. A screenshot of the application is shown in Figure 4.1. Users of the VolumeShop environment are able to combine multiple interactive renderers for meaningful and customizable non-photorealistic representations.

The extensibility of the application allows for an arbitrary number of plugins to be added, modified and applied at the same time. It is also possible to add and display more

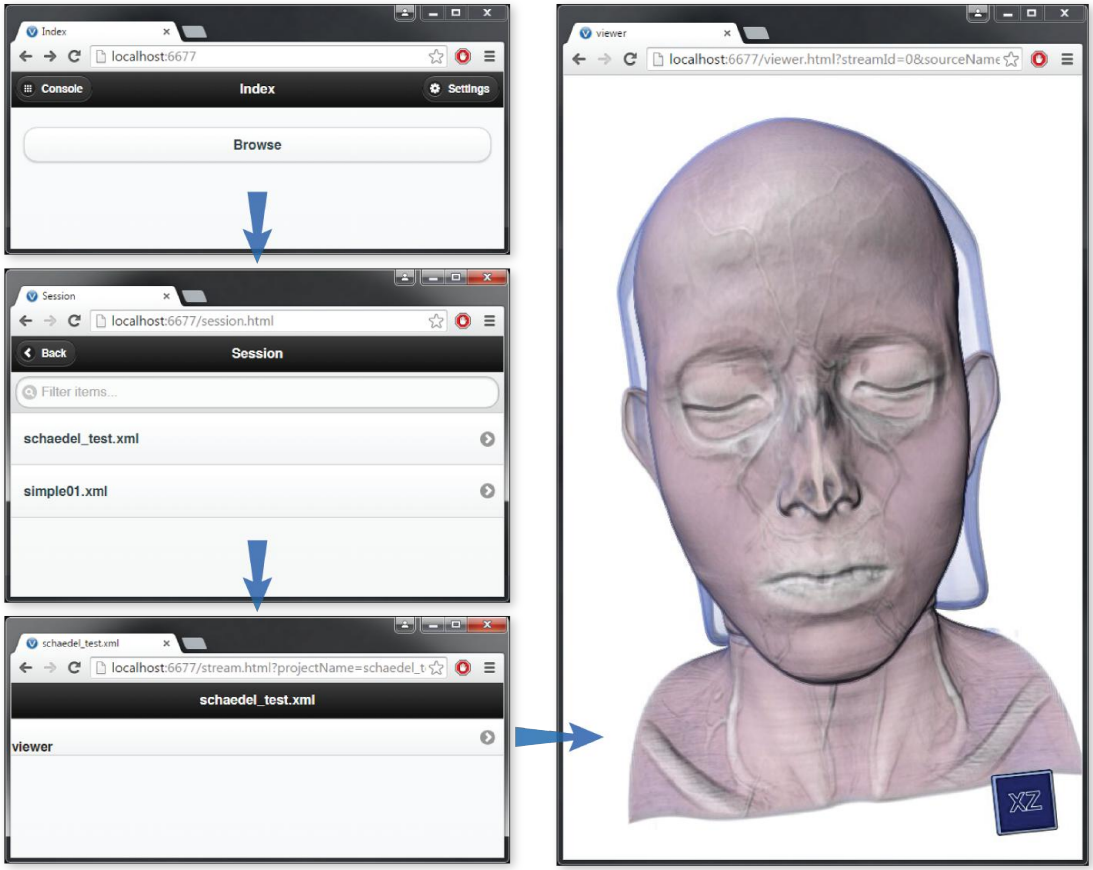


Figure 4.2: The VolumeShop web viewer in its original state as pictured here is a thin client interface that streams the image output from the host. This diagram shows the process of loading the same session which was displayed in the standalone VolumeShop program in Figure 4.1. After the corresponding session and viewer were selected from the interface, the viewer interface is loaded into the browser window, which can be interacted with through 3D rotation and zooming. Note that the property windows provided in the standalone tool are not available in the simplified web client, which is why all the settings need to be specified beforehand and saved to a session file.

than one viewer in order to display multiple visualizations or the same volume in various representations simultaneously. If there are several renderers applied to one viewer instance, a compositor is responsible for the superimposition of the outputs of the renderers in the order that they are sorted.

The architecture of the framework facilitates a straightforward implementation of additional components by offering a plugin-based system with interfaces for different types of plugins.

4.1.2 Remote Visualization with the VolumeShop Framework

VolumeShop is usually executed as a standalone application as described in the previous section. Besides this application, VolumeShop also offers a client/server-framework for presenting rendered content by displaying it remotely in a web browser interface. An instance of this framework and the workflow of accessing a visualization within the web viewer is shown in Figure 4.2. This environment was created to offer an easily accessible visualization interface for end users who do not wish to concern themselves with the deeper settings for modifying the visualization which are available in the regular VolumeShop GUI. This framework was used as the basis for our remote visualization implementation and was adapted and extended into an interactive environment to work with our approach.

When running VolumeShop as a server instance rather than using the GUI, the server needs to be started from the console and opens a socket. The client can open the web interface in any browser and send requests to the server.

Sessions

Prepared VolumeShop sessions can be loaded via the simplified client interface. The session specifies which dataset is loaded and which renderers are responsible for displaying it in the viewer. The server then creates the renderers and writes their output to a framebuffer. For accessing a particular visualization session using the web client, the VolumeShop session needs to previously have been created using the VolumeShop GUI since the simplified client does not offer interfaces to specify the datasets and plugins to be used within this session.

The session is stored in an `.xml` file which describes the location of the displayed data set, the viewers that are made available to display the data and the active renderers and plugins used for the visualization within each viewer as well as all other VolumeShop settings that can be loaded on startup of the application.

Workflow

When loading the session from the web interface, the user is provided with a view which offers a selection of the different viewers the session has stored. Since the interface is meant to be kept simple, there can only be a single viewer displayed at once within the browser window.

As soon as the user selects a specific viewer, the actual content associated with the respective viewer is loaded into the web interface, displaying a window streaming the rendered data from the web server. Here, the composed output of one or multiple renderers is displayed in the way it was defined in the session. For this purpose, the web server creates a VolumeShop instance with the predefined session parameters, streaming a single image containing the composite output of the VolumeShop viewer from the server to the client upon request. The interaction modes of this client/server-framework are limited to rotation and zooming in the 3D view of the data set. The client displays an image stream, while the

rendering process is performed on the remote server. In accordance with the classification of remote rendering systems as presented in Figure 2.4, this can be classified as a thin client.

4.2 Data and Segmentation

We have implemented our novel server-side rendering component as a plugin in the VolumeShop framework. It outputs the Volume Object Model, the intermediate object representation we propose in Section 3.2.

For the purpose of rendering a segmented volume, we load a data set and perform a threshold-based segmentation algorithm. Afterwards, we run a connected component analysis which results in a labeled volume. Alternatively, the segmentation can also be loaded from a volume file. The segmentation can be programmed using a ViSlang DSL [RBGH14].

The current implementation of our renderer receives a voxelized volume where every voxel is labeled by an integer value that can be either zero (background) or a label denoting the voxel's allocation to a specific object.

The input volume is loaded from a `.dat` file which stores the data values in the a simple binary format at `unsigned short` values, which sets the data range to `[0,4095]`.

4.3 Server Implementation

On the server side, a novel renderer was implemented within the context of the regular VolumeShop framework. Since our system runs on a regular workstation, in our implementation, we introduced a deferred visualization pipeline that includes several optimizations.

4.3.1 Overview

The server side is based on a segmentation-data ray-casting algorithm [RBGH14] which is an existing VolumeShop rendering plugin. It is a GPU-accelerated renderer that executes an OpenCL kernel writing the output to an image buffer. The image buffer is displayed using OpenGL or transferred to the client over the network. To efficiently transmit the object visualizations over the network we pack as many images as possible into one output buffer.

In contrast to conventional ray-casting, our renderer outputs the volume not as a composition of all objects, but renders each object into a separate rectangular area in the output image, which is sized according to the object's minimal screen-space bounding rectangle.

After the data segmentation, we perform an object-tracing step, which is responsible for determining view-dependent object properties. For this purpose, the renderer needs to traverse the volume in two passes (each executed in a GPU-accelerated OpenCL kernel) rather than one ray-casting pass. The screen-space bounding boxes of all the objects in the scene are evaluated in the course of this traversal while tracking the boundaries for each object.

A rectangle packing step is executed on the CPU in-between the two render passes. The two-pass approach is necessary because the minimal bounding rectangles per object need to be determined before the actual rendering can take place.

In order to reduce the workload for the second object traversal, we also buffer a per-pixel object count which can be used to look up the expected number of hits in the second render pass. Subsequently, we use the image-space bounding boxes in a rectangle packing step to create a packing of all object boxes within one rectangle. We do this because we would like to transfer all object representations within a single output buffer. This step is necessary to determine the location of every object within the output buffer.

A render pass over the volume by traversing it once again via ray-casting renders the individual objects to the dedicated areas within the output buffer. For this purpose, we pass the render kernel the coordinates of the rectangular regions that define the objects' render targets. From this information, the render kernel evaluates the correct position for every pixel in the output.

4.3.2 Metadata Storage

For the storage of the metadata, we use VolumeShop's integrated property functionality, where plugins have the ability to store and read from different types of attribute data. A property is stored with a string descriptor and a data field. VolumeShop's property fields store data of the VolumeShop API class `Variant`. The data types that can be stored to this field include Integer, Boolean, Float, String, etc. Data structures (Arrays, Matrices, etc.) are composited from the primitive data types.

Additionally, we store and transmit an object occlusion matrix in order to determine on the client side whether we need a more sophisticated depth comparison between two specific objects.

For large datasets, object relationship matrices are typically sparse. For instance, in the case of the occlusion matrix, most objects are not occluding most other objects, leaving most entries of the matrix zero. A sparse matrix representation would save memory consumption and transmission overhead in such cases. However, in our current implementation, we store and transmit dense matrices only.

4.3.3 Object Traversal Pass

If the application calls the renderer's display function, the first render pass is initiated. It calls the OpenCL kernel `d_precompute` for the window, which performs a ray-casting operation for each pixel of the viewport. This kernel determines the viewpoint's position and viewing direction with respect to the data volume, and emits a ray into the data set. In the OpenCL function `evaluateRay_precompute` which is called from the kernel for the ray being cast for the current viewport pixel, the position along the ray is incremented stepwise until it exits the volume. In the course of this traversal, data about the objects is collected.

The information acquired from this ray-casting pass includes the number of objects that are hit along the ray as well as their IDs. In this process, a voxel being hit counts as an object whenever it is nonzero (which is the value assigned to empty voxels) and differs from the voxel previously hit along the ray (that is, it is the front face of the current object as seen from the eye position). This process implies that objects that would be occluded from the viewpoint in a conventional renderer as well as objects that are concave and are therefore hit multiple times along a ray with another object or empty space in-between are not left out.

Most importantly, however, the object-tracing pass determines the bounding rectangle for each object in the scene with respect to the viewport. For this purpose, the minimal and maximal coordinates of all the objects that were detected along the ray are compared to the current position and updated if necessary. The screen-space bounding box values are stored in global buffers for the minima and maxima in both dimensions.

We use atomic operations for the comparison of a potential new value to the old value in order to keep the value's integrity even if multiple threads try to access it simultaneously. Therefore we apply the `atomic_min` and `atomic_max` functions when testing for boundaries.

4.3.4 Object Image Packing

Since our implementation is designed to transfer all object visualizations in one buffer, we try to arrange those render targets optimally within a rectangular area. This approach also has the advantage of letting the use view the visualization output as a single image when instantiating our renderer in the VolumeShop GUI on the server.

The minima and maxima buffers that were output from the object-tracing kernel are then used to determine the minimal possible area of the renderer's final image buffer as well as the location of each individual object within the output. Since the individual rendering of all objects results in an output of rectangular areas of different sizes, these resulting rectangles need to be arranged optimally within the result image in order to minimize the total image area.

Object Bounding-Box Packing

At this point in our approach, we have evaluated the size of the rectangular rendering destination that each individual object requires. This size is determined from the minima and maxima values that we receive as the output of our object-tracing pass, which can be used to calculate the width and height for each object's size. Since we pack images of multiple objects into one common render target, we attempt to arrange the rectangular object areas optimally within the output buffer in order to minimize the memory consumption. To this end, we perform a packing operation on the bounding boxes of our objects.

The Rectangle Packing Problem The task of packing our render targets into one output is as a rectangle packing problem, which has been shown to be NP-hard [Kor03]. It is an optimization problem where two main problem sets can be distinguished: On the one hand, the problem of fitting as many rectangles as possible into an area of fixed size and on the

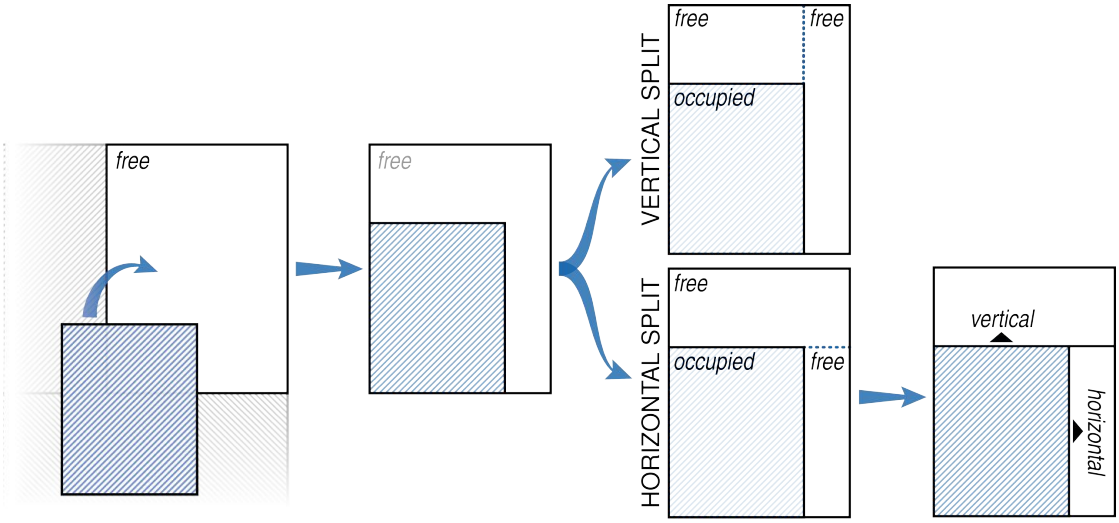


Figure 4.3: If a rectangle is fitted into a free area with remaining space, the block needs to be subdivided. The cut that separates the new horizontal and vertical sub-blocks can be done in two ways as illustrated. We chose the solution that maximizes the area of one of the sub-blocks to increase the chance of fitting another block into the empty space in the subsequent fitting steps.

other hand, finding an enclosing rectangle of minimal size that fits all elements of a set of arbitrary sized rectangles without overlap [Kor04]. Since the first of the problem sets cannot guarantee that all rectangles can be fit into the result and a lot of space may remain empty if either the number or the size of the rectangles is very small, we chose the second approach to find an optimal fitting for our rectangular object areas. Our approach to the rectangle packing task is based on the algorithm described by Gordon [Gor11]. For further reference, Algorithm A.1 in the appendix outlines the implementation of our algorithm in pseudo code.

Rectangle Sorting In order to create a packing that fits all our objects into a minimal containing rectangle, we require a list of all rectangles sorted from biggest to smallest. This is necessary because bigger rectangles need to be packed first, while smaller rectangles can be filled in the free space that remains. For this purpose, we tried sorting the objects by different criteria: by *width*, by *height*, by $max(height, width)$ and by the rectangle's area $height \times width$. We found that a sorting by $max(height, width)$ performs best because for rectangles with uneven proportions (i.e. one side is much larger than the other), the prioritization of the longer side ensures it is inserted as early as possible. The insertion of two or more such rectangles will usually generate a larger free space, which, if fitted early on, is more likely to be filled by the subsequent candidates.

Rectangle Packing For the packing itself, the list of sorted rectangles is iterated from biggest to smallest, ensuring that the big rectangles are first fit into the result with the smaller rectangles filling up the remaining free space. Since we use a growing rectangle approach, the output size is initially set to the size of the first (and biggest) rectangle in our list. Whenever

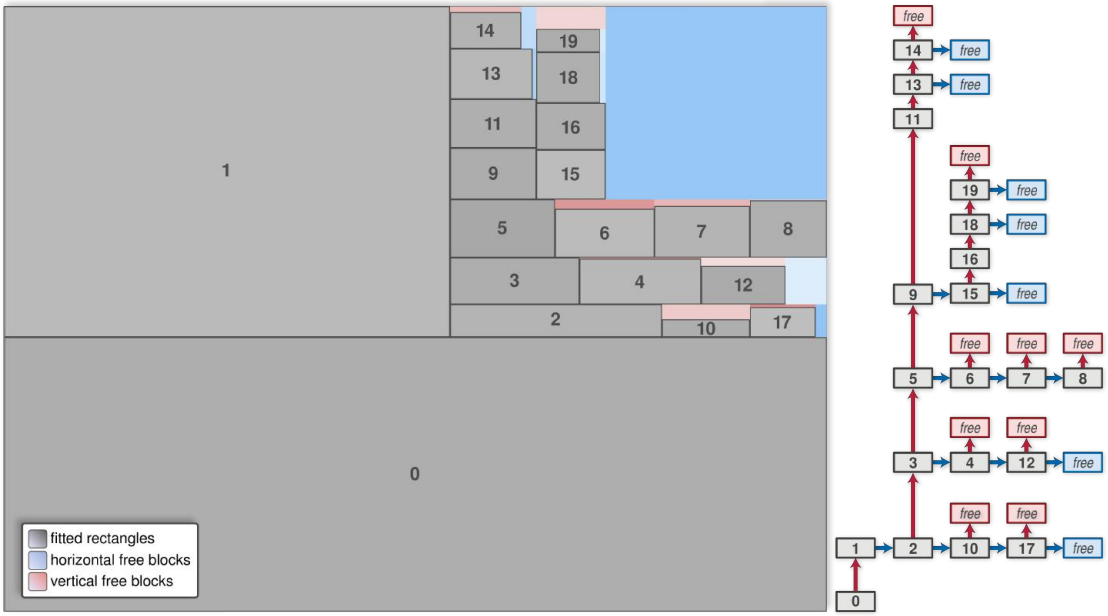


Figure 4.4: This figure depicts an intermediate result during the rectangle packing process of our implementation where 20 rectangles (in gray hues) were already packed into the output area. The free blocks are shown in red and blue hues, depending on whether they are the horizontal or vertical child of their parent block. For better comprehensibility, the placed rectangles are numbered according to the order in which they were inserted. Next to the intermediate result, we illustrate the underlying tree structure of the rectangle subdivision.

a new rectangle is supposed to be fit into the result, the output is recursively searched for any remaining space that is big enough for the width and height at hand. If there is no free space of sufficient size, the output is resized to dimensions that can accommodate the new rectangle. In order to keep track of the allocation of areas and also to search efficiently for free space, the containing rectangle is subdivided into free and occupied blocks, building a binary tree storing horizontal and vertical nodes.

Tree Traversal For the fitting of any rectangle, the first step is to search the existing tree for free areas, and, whenever a free block is found, determining whether its area is big enough. We choose to search the whole tree recursively rather than simply picking the first sufficiently big area in order to find the smallest possible space that can still accommodate our rectangle. To this end, for every free block with large enough dimensions, we calculate a ratio $\frac{rectangleArea}{blockArea}$ between the area of the potential rectangle candidate and the block's area. Finally we select the available block with the maximum ratio, meaning that, from all potential candidates, we choose the one of the smallest dimension.

As soon as the most suitable unoccupied space is found, we set the status of the block

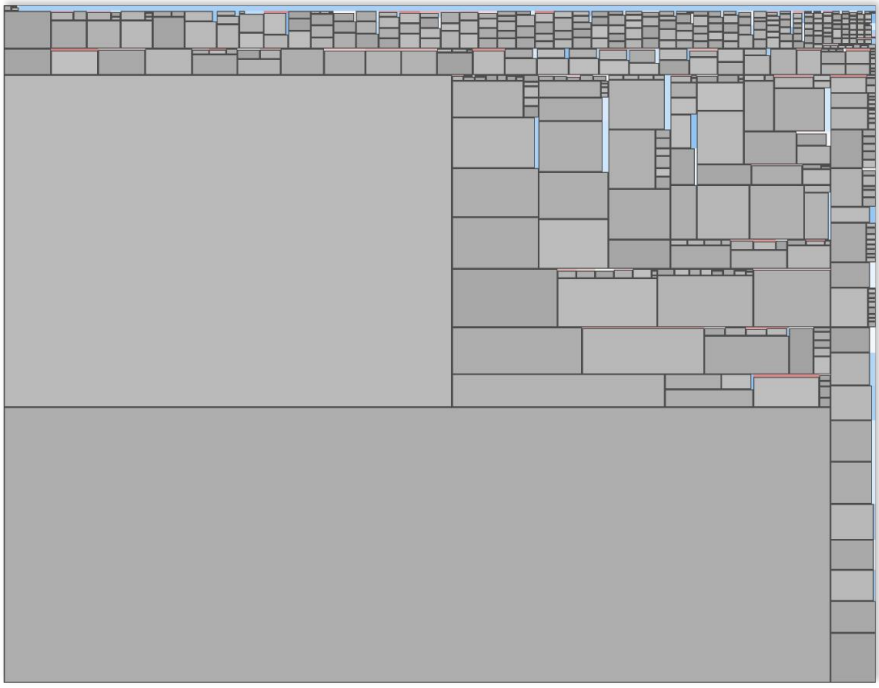


Figure 4.5: An exemplary output of our rectangle packing algorithm, the intermediate result of which was depicted in Figure 4.4, is shown in this Figure. This example shows a packing of 512 rectangles and illustrates that in most conditions, our algorithm is able to pack the rectangles efficiently. This final result serves as input for the consecutive render pass.

to occupied and assign the coordinates of its designated space to the rectangle. It is also determined whether the horizontal and vertical dimensions of the rectangle fit the free area exactly. If this is not the case, the remaining space is divided into one or two blocks that are still unassigned. The unassigned block(s) describe the free area in horizontal and vertical direction and are appended as children to the newly occupied block, thereby extending the tree structure. This process is illustrated in Figure 4.3.

Packing Heuristics Since we do not know which size of rectangle might be fit into the free blocks, there is no efficient way to determine whether a horizontal or vertical split will yield better results in the end. We therefore calculate the sizes of the remaining children and the quotient $\frac{smallerArea}{biggerArea}$ for both possibilities. Small values for this quotient are considered favorable, since it means that one area is much bigger than the other one. Choosing the split that minimizes this quotient therefore results in the subdivision that is more likely to be able to fit other big rectangles.

Generating the Output The searching and placing process is repeated for every object until all objects are packed into the output. Figure 4.4 shows an intermediate result of our rectangle packing algorithm, while also serving as illustration of an interim step of the construction of

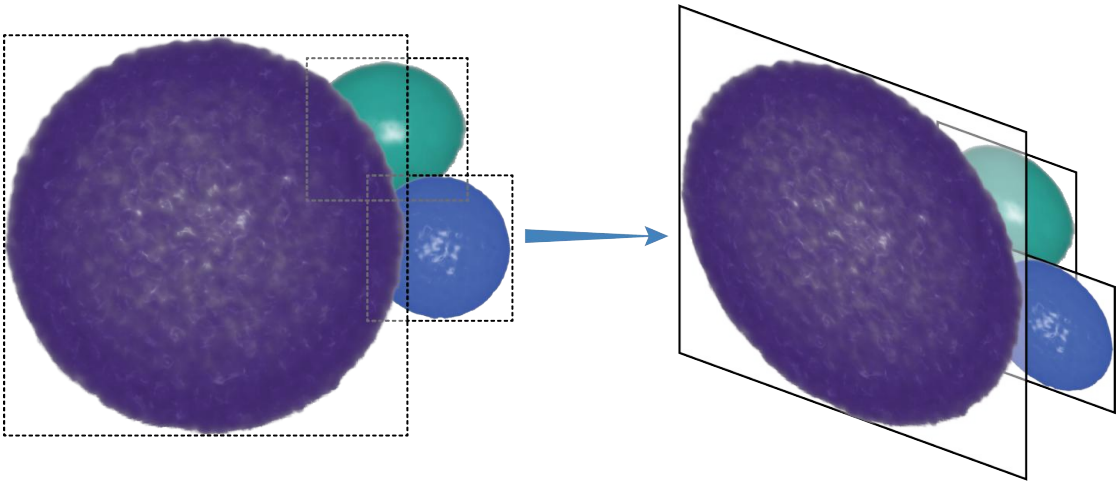


Figure 4.6: Since the objects are rendered to two-dimensional representations, the naïve approach to display them in a correct depth ordering is by layering the objects according to an average depth value calculated for each object similar to the billboard approach common in real-time rendering techniques.

the binary tree built during the process of the packing. We show the underlying tree structure next to the intermediate result, with blue arrows depicting horizontal child relationships and red arrows pointing towards vertical children.

For use cases with a sufficiently large number of objects, Figure 4.5, which shows the final result of the packing displayed in Figure 4.4, demonstrates that our approach generates an output that fills up a rectangular area while leaving little free space.

4.3.5 Volume Rendering

The assignment of the object coordinates within the output buffer serves as a map for the second pass to render the objects into the allocated bounding rectangles which is executed by calling the OpenCL kernel `d_render`. This kernel traverses the volume in a fashion similar to the object-tracing pass by shooting rays into the volume from each pixel using the function `evaluateRay_render`. Each ray through the volume is traversed in step-wise increments, evaluating the voxels the ray passes through.

After the ray traversal has gathered all the information about the objects along the ray, they are rendered into an off-screen buffer. The rendering targets are represented as two-dimensional layers per object, an approach that works in a similar way to the screen-aligned billboards rendering frequently used in real-time rendering scenarios [AMHH02]. An illustration of the objects that are represented by layers is shown in Figure 4.6.

In our implementation, we place all render targets into one common one-dimensional image buffer. The size of the image buffer is determined by $width \times height$ of the dimensions that the rectangle packing determined.

Object-based Early Ray Termination Since we have recorded the number of objects encountered per pixel to a screen-space buffer in the first pass, we can make use of this information in our render pass. The precomputed per-pixel object count is queried to determine whether there are any objects expected along the current ray and whether it therefore needs to be pursued or can be skipped without further investigation.

While in the first ray-casting pass the ray was only terminated when reaching the volume's boundary, we can now end the ray traversal for each ray as soon as we encountered the expected number of objects for this particular pixel. This is particularly helpful for rays emitted from pixels that do not reach any object – through this strategy, these rays can now be aborted right away.

Accumulating the Color Value per Object For every voxel encountered along the viewing ray, we can access both the original volume as well as the segmentation mask, which specifies the voxel's affiliation to an object. The voxel's RGBA values are accumulated to the color value of the object denoted by the voxel's label.

The RGB color of the voxel can be looked up in a global transfer function according to the voxel value in the original volume. Alternatively, a color value can be assigned to each label to allow for a visual distinction between different objects. Through settings in the shader, we can specify a blend factor to take into account both the label color value as well as the transfer function. The alpha value is always looked up in the transfer function.

By storing the object-wise accumulated colors, when the ray is terminated, distinct color values are returned to the kernel for all objects that were encountered. Through this approach, multiple objects can be rendered from one single ray.

Writing to the Image Buffer When the ray is terminated, the kernel writes the accumulated color values for all encountered objects to the image buffer. In order to render to the correct pixel of the image buffer, the renderer is supplied with the coordinate list for the rectangles of all the objects determined in the rectangle packing. Figure 4.7 illustrates how the mapping from the camera coordinates to the image buffer is calculated.
The pixel's index in the one-dimensional buffer sums up to

$$index = width_{imageBuffer}(y_{local} + y_{outputCoords}) + x_{local} + x_{outputCoords}$$

where $x_{local} = x_{viewport} - x_{boundingBox}$ and $y_{local} = y_{viewport} - y_{boundingBox}$.
$x,y_{outputCoords}$ denotes the coordinates calculated in the rectangle packing and $x,y_{boundingBox}$ are the the bounding values of the object in the context of the volume, evaluated in the object-tracing pass.

Not only the first object but all occluded objects are written to their respective render target. This implies that since the color values were accumulated per object in the ray-casting, the pixel buffer can be accessed multiple times per kernel call.

Depth Buffer At this point we note that we do not only store the color values to the image buffer, but also allocate regions of the same size as the objects for storing a depth

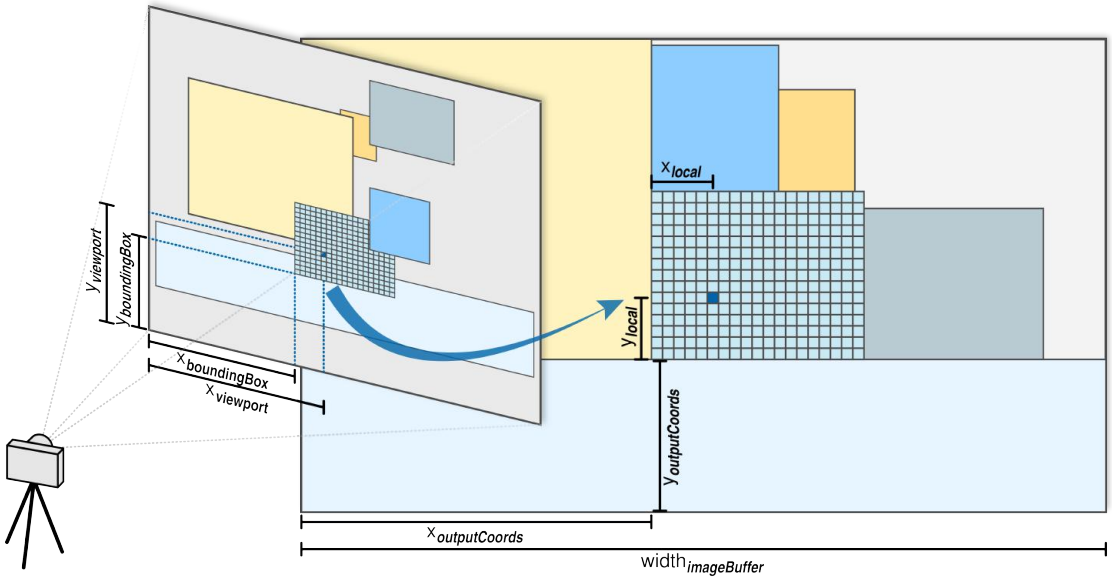


Figure 4.7: Mapping from image space to image buffer coordinates: In the render pass, each object is rendered to its dedicated rectangle in the output image buffer. The position of the rectangle is determined by the rectangle packing algorithm. To render to the correct area, the coordinates need to be translated from the image coordinates to the coordinates of the image buffer.

representation of the object to the output. For this purpose, we double the object's calculated width before rectangle packing and use the second half of the provided area for the storage of the depth values. The storage of the depth values increases the size of the output buffer, but gives us greater accuracy in rebuilding the scene at the client, which is particularly relevant for complex objects. The depth values could also be stored to a dedicated buffer rather than being included in the image buffer, which would lead to both more efficient storage and a more suitable data representation (in per-pixel linked lists) of the objects' depth representations. We were limited in this issue by the current constraints of our framework and consider the improvement of the depth representation as future work.

4.4 Client/Server Communication

The client/server communication operates over sockets. In order to open a socket, the server relies on the socket functionality that is integrated in the Qt framework [Qt15]. It opens a socket on the specified port and listens to incoming connections. The client can then send requests to the server's address.

Our implementation optimizes the deferred visualization pipeline we described in that we request the metadata and the visualization simultaneously. This works for relatively small data sets while it is not applicable to in-situ scenarios. The metadata and visualization

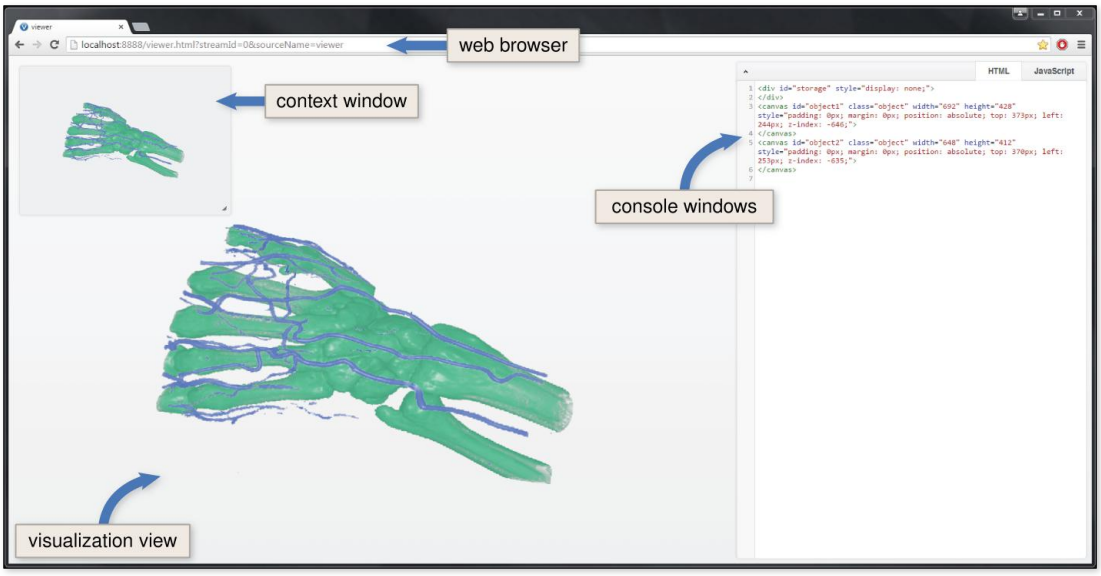


Figure 4.8: The web client can be accessed through any web browser and consists of a main view showing the visualization data as well as a smaller overview, here shown on the top left. If objects in the main visualization are modified, the overview preserves the context of the original object arrangement. On the right, a console shows the HTML content of the visualization, displaying the canvases for each object, and another console is used for scripting JavaScript commands. These consoles can be edited interactively.

are already streamed independently through JavaScript requests, since the visualization is requested via an image stream and the metadata is obtained following a request for the VolumeShop property. In order to wait for both requests to finish their execution, we rely on the JQuery `Deferred` object. This concept allows multiple JavaScript functions to return a promise, which is only resolved when the function has finished. The main function therefore waits for all promises to be resolved before it continues with the execution.

4.5 Client Implementation

The Web Client, which is shown in Figure 4.8, is designed to run in any web browser capable of displaying HTML5 content. This approach has multiple advantages: On the one hand, this ensures broad support for the client on any kind of device. The client is run in an environment that users are familiar with and which does not require additional software or plugins. Additionally, this strategy also gives access to the highly developed technologies that modern web browsers offer, which makes our solution highly customizable.

4.5.1 Pre-Visualization

In order to be able to deal with large data, our approach proposes a pre-visualization step which performs visualization tasks solely on the metadata. This allows the user to pre-filter and modify the objects according to their attributes, thereby optimizing the query for the request of the images. JavaScript can be used to attach glyphs to the abstract metadata to perform information visualization: e.g. scatterplots, histograms and other visualization techniques can be programmed through the console using the D3 library.

Since our implementation was executed on a regular workstation and we were not handling extremely large data sets, we requested both metadata and images at the same time. The pre-visualization tasks can be performed on the metadata nevertheless.

4.5.2 Image Unpacking and Image-based Rendering

In order to convert the streamed data into our object representation, we need to subdivide the image buffer which we received through the host's image stream into the rectangles specified by the image coordinates in the metadata. The image-related metadata is stored in a normalized form with values in the range $[0, 1]$, therefore we need to adjust the coordinate values to the height and width of our image data, which was also transferred in the metadata. Since we output our objects as HTML5 canvas elements, the slicing of the image into object regions is straightforward. The canvas element can be filled from a subregion of a specified input image by indicating the appropriate coordinates and sizes. Furthermore, the canvas element provides functionality that is hardware-accelerated, which is beneficial for the rendering of the objects in the browser.

We iterate through all the objects specified in the metadata and access the canvas elements for each of the objects. On the first loading of the viewer, these canvas elements need to be created and appended to the viewer. In this process, each canvas is assigned a common class as well as a unique ID which corresponds to the label of the appropriate object. Through these descriptors, objects can be addressed individually or as a group. At a later point, i.e. when the view is reloaded due to a rotation of the viewpoint, the canvases are simply updated by accessing their unique IDs. We then fill every canvas with the corresponding rectangular part of the image received through the stream and finally displace the canvas to the location as specified in the metadata.

An advantage of the usage of HTML5 elements is that arbitrary metadata which was transferred in addition to the data for the object reassembly can easily be attached to an object by assigning it to the object canvas as an attribute, which is composed of a string descriptor and a variable value. Through this methodology, metadata can be effortlessly queried from any object by requesting it through specifying the corresponding string property.

We determine from the occlusion matrix we transmitted as part of the metadata, whether a mutual occlusion occurs between two objects. If this is the case, we do a pixel-wise depth test for the overlapping area, looking up the actual depth values of both objects in the depth buffer we transmitted as part of the image buffer. The pixel-wise depth test is a costly operation. It

is, however, necessary to ensure a correct depth representation of the objects at the client.

4.5.3 Object Interaction and Manipulation

We integrate a console into our viewer window which allows the user to script the HTML5, CSS3 and JavaScript behavior of the visualization. For the console environment, we used the CodeMirror [Cod15] library which provides an interactive scripting environment that integrates into web pages. Most web browsers also offer debugging consoles with extensive editing capabilities that essentially provide the same functionality.

Since we are rendering our layers as HTML5 objects, we make use of the capabilities of modern web browsers. The objects' behavior can be programmed by accessing the canvas layers via the built-in browser technologies JavaScript (to influence the behavior), CSS (to modify the appearance) and HTML (to access the document as well as the objects' position and size).

We furthermore included the D3 JavaScript library [BOH11] within our framework in order to facilitate visualization tasks. D3 offers powerful tools for frequently used manipulation tasks of the Document Object Model, especially through providing straightforward selectors for objects from the DOM and then applying operations to each of the selected objects.

4.6 Limitations

Our implementation was integrated within the VolumeShop framework and partly built on existing parts of the framework. This facilitated some tasks, while it limited our implementation in other regards.

First and foremost, we would ideally like to use more than one renderer with our approach. Our `VLabelObjectsRenderer` is a relatively simple volume renderer. However, since our renderer outputs the visualization object-wise and additionally calculates the metadata, any type of renderer would have to be adapted to comply with our approach. Alternatively, we could integrate additional rendering options into the existing renderer to make it more adaptable.

Since we reused parts of VolumeShop's client/server interface, we were not able to completely rebuild the data transmission within the scope of this thesis. This turned out to pose limitations to our implementation, since we had to pack all of our image targets within one image buffer for the transmission. If we output and transmit every image to its own target, rotation (which triggers re-rendering) could be performed on a single object instead of requiring a re-rendering of the entire data set.

Furthermore, the metadata transmission was executed through the VolumeShop properties functionality, which can be accessed remotely through an appropriate request. While this functionality was convenient, it also constrained the way our metadata was stored. Furthermore, the storage and transmission of the depth buffer was integrated into the image

buffer because more efficient storage (e.g. as per-pixel linked lists) would have required a remodeling of the framework.

Additionally, the memory consumption of the OpenCL kernels could be improved. Since we need to store multiple buffers with object-wise data and dynamic memory allocation is not possible in OpenCL, we need to allocate buffer sizes corresponding to the maximum number of labels we allow.

For complicated depth problems, the pixel-wise comparison solution proves to be slow on the client side. This is the case because custom pixel-wise canvas operations are very slow in contrast to the hardware-accelerated functions the canvas provides. Generally, the client side calculations can get slow for larger data sets because JavaScript's performance does not scale very well. The implementation of the pre-visualization step would help alleviate this issue.

CHAPTER 5

Results

In this chapter, we give an overview of the capabilities of our system. We demonstrate the results of our rendering framework as well as the object interaction that is possible on the volume representation which we can access in the web client. Furthermore, we discuss the design of the system and report performance numbers that show the benefits of our approach.

System

We implemented and tested our application on a workstation running Windows 7 x64 on an Intel Xeon X5680 CPU and 48GB of RAM. The graphics card used in our workstation was an Nvidia GeForce GTX TITAN Black. We run both the client and the server on this machine. However, the components are solely connected through web sockets and could reside on different machines.

Data Sets

We show the functionalities of our system by means of two exemplary data sets. Figure 5.1 depicts a volume rendering of a data set of a human hand. This hand data consists of two large objects, measures 244×124×257 voxels in size and is well suited to present the object modification capacities of our system. Figure 5.2 shows a volume rendering of a data set of a Christmas tree [KTM+02]. We use this data set to illustrate the capabilities of the system and demonstrate that it scales to a large number of objects. The Christmas tree data set measures 512×499×512 voxels and consists, depending on the segmentation parameters, of up to 1350 objects.

5.1 Server-side Object Rendering

We first describe the server-side object rendering. The server loads data sets, labels the objects in the volume through a ViSlang script and renders the Volume Object Model representation.

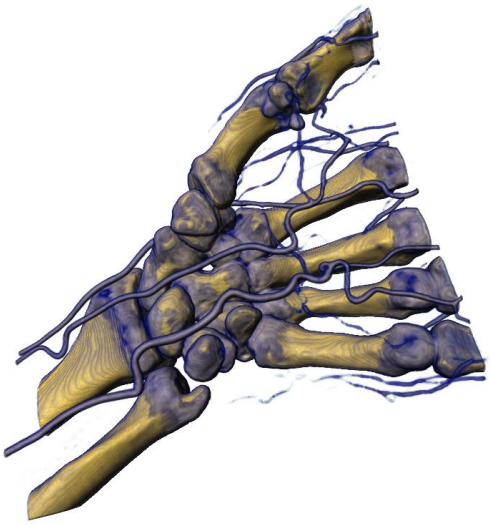


Figure 5.1: Human hand data set

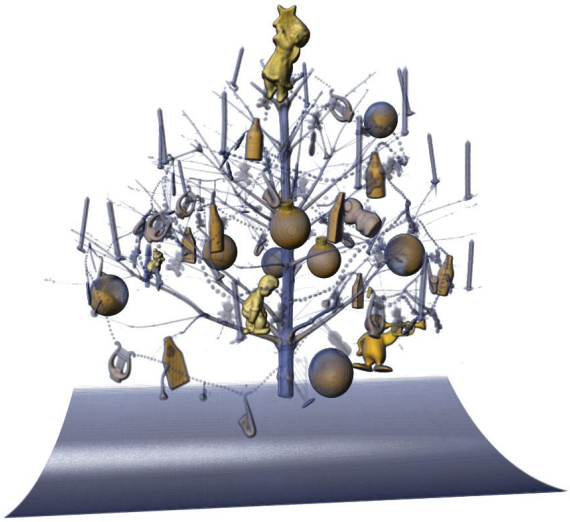


Figure 5.2: Christmas tree data set

In order to generate the Volume Object Model, we perform an object-wise volume rendering of the data set where the appearance of the objects can be configured according to multiple parameters. For each object, we perform a full volume rendering where the RGBA values for the output are accumulated object-wise. Depending on our settings, we can define transfer functions globally or for the individual objects based on their labels.

5.1.1 Object Coloring

In order to be able to visually distinguish different objects, we color code the objects according to the distinct labels as shown in Figure 5.3. The renderer allows us to render the objects based on a global transfer function as depicted in Figure 5.4. The appearance of the colors used in the volume rendering is adjustable through the transfer-function panel. Furthermore, we can smoothly blend between the assigned label colors and the transfer function in order to show the volume rendering with enhanced object distinctiveness (Figure 5.5).

Usually, the rendering is called remotely from the client, but we can also instantiate our renderer in the VolumeShop GUI on the server in order to generate output images of the Volume Object Model for debug purposes. As shown in Figures 5.3, 5.4 and 5.5, the images of the individual objects are packed into a rectangular image on the server side in our implementation. For illustrative purposes, we show a packing of solely the output images for every object, leaving out the metadata and the depth buffer.

The full representation also contains depth buffers in order to render complex occlusion cases correctly. In our current implementation, this means that the image buffer doubles in size. Figure 5.6 is a detail of the result of our server-side rendering where both the object image and the depth image of one objects are rendered side-by-side. The depth is encoded in gray-scale, white signifying near and black signifying far values.

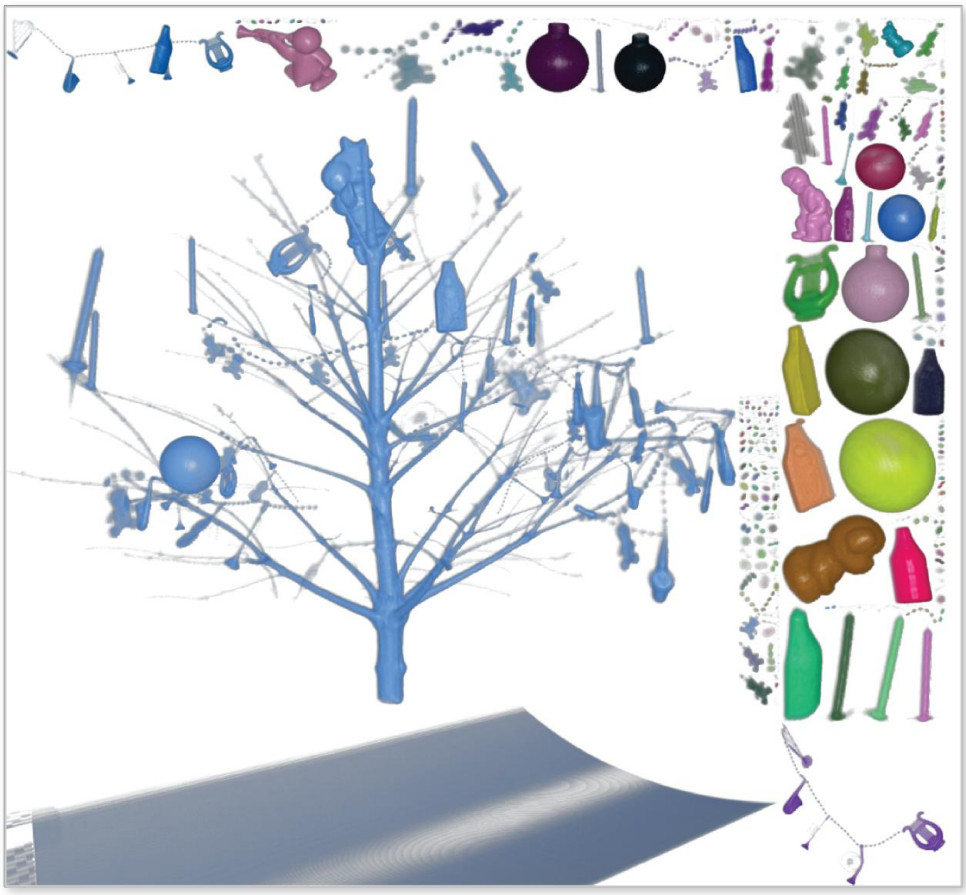


Figure 5.3: Every label can be assigned a color to allow for a distinction between the different objects. In this case, only the alpha value is taken from the transfer function.

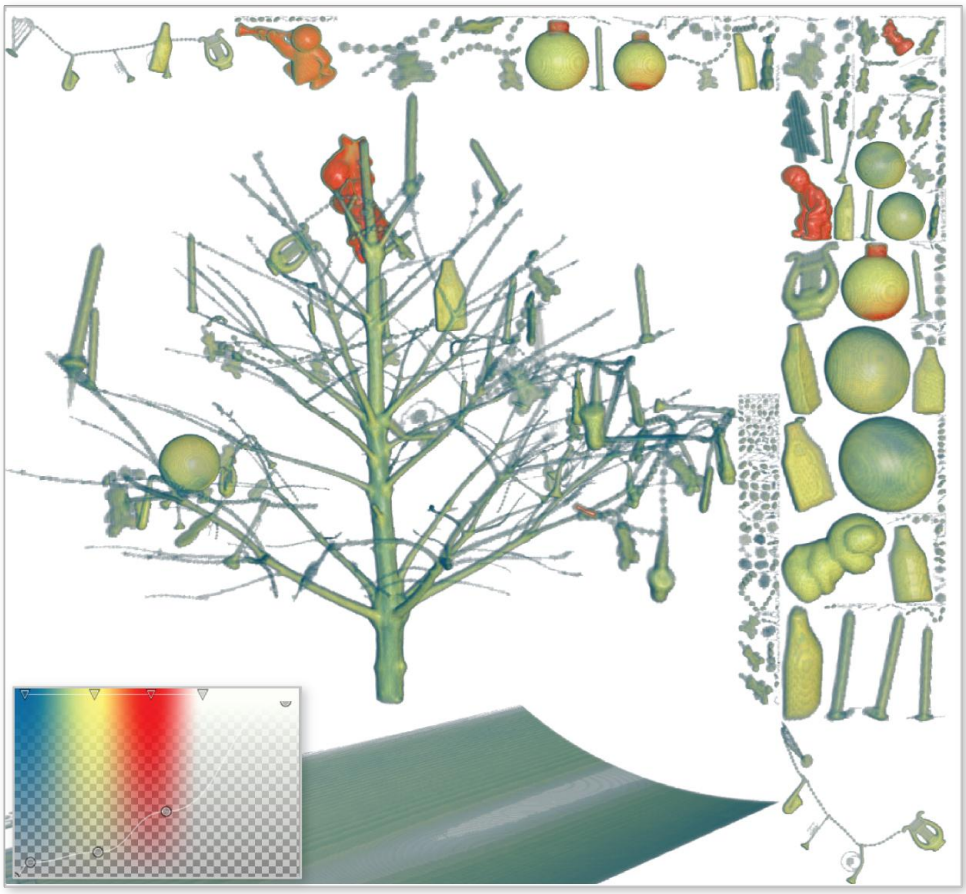


Figure 5.4: The renderer is capable of coloring each object according to a global transfer function. To that end, the color values of the samples along the ray are looked up from the transfer function and accumulated object-wise.

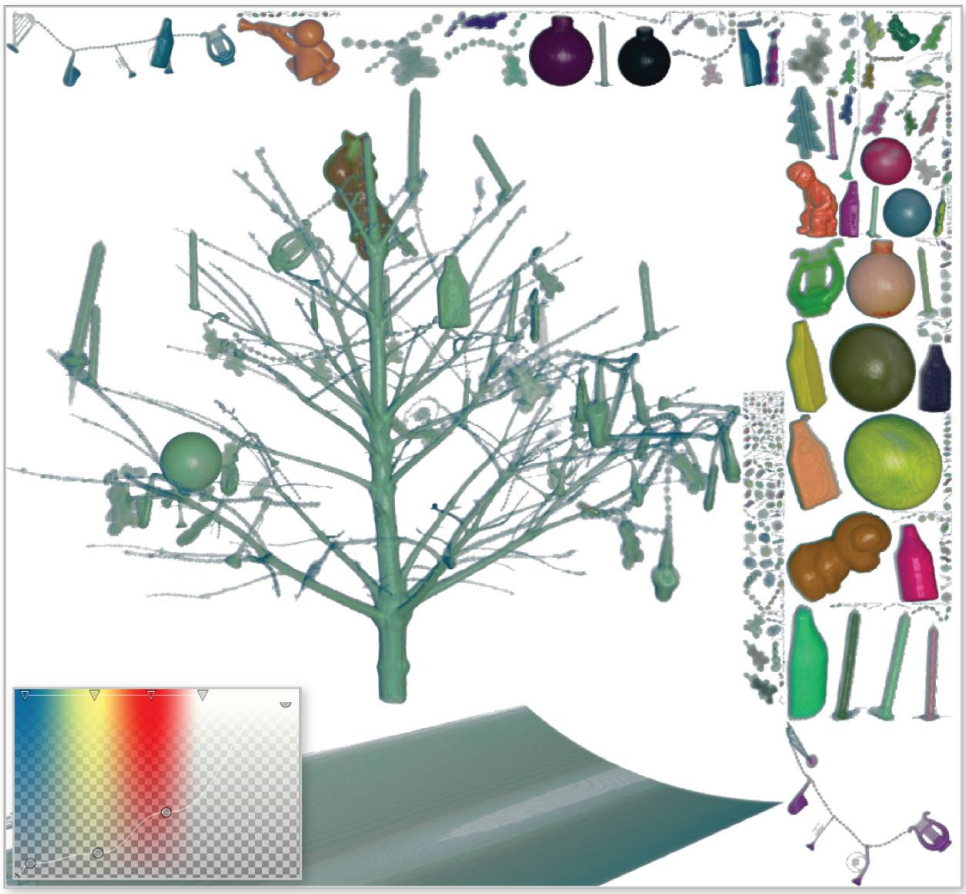


Figure 5.5: The volume rendering can also be blended with the distinctive label colors to enhance the rendering. The factor of this blending is specified in the shader.

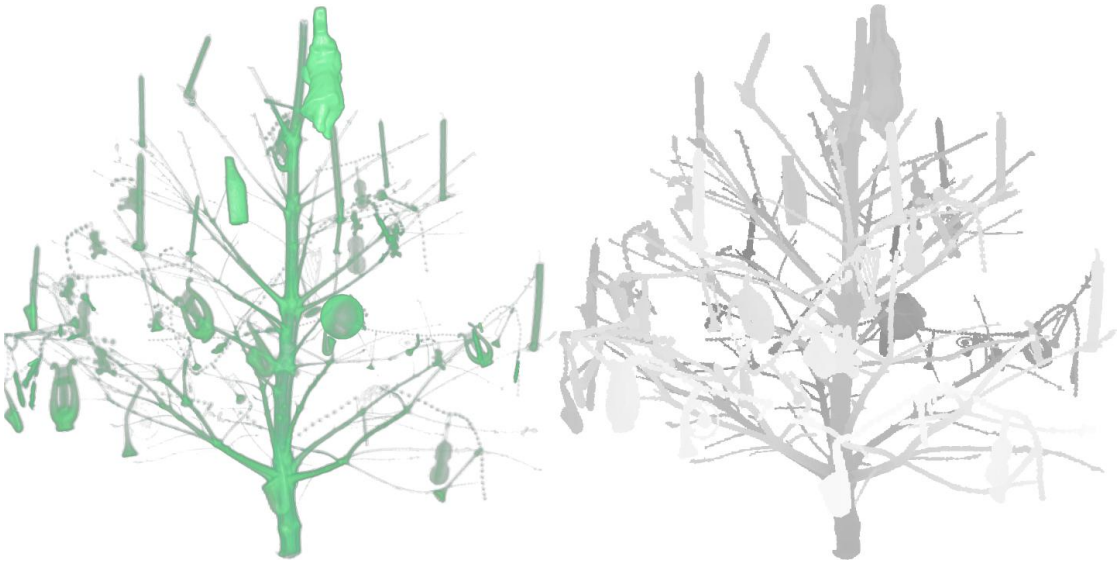


Figure 5.6: We store an object-wise screen-space depth buffer to resolve occlusions on the client side. Depicted is the result of rendering one object from the Christmas tree data set [KTM+02] (left) with the associated depth image (right).

5.2 Web Client

In Figure 5.7 a screenshot of the client can be seen. The interface in the browser window consists of three areas: the main visualization view, the console and a moveable and resizable overview window. The web client requests the Volume Object Model from the server. On receiving the streamed response, it builds the scene from the individual objects, as Figure 5.8 shows. Each object is displayed in its individual rectangle which is placed at the correct x- and y-coordinate and layered using an approximate depth value. Figure 5.9 shows the outline of each of the canvas elements in the scene. It serves to illustrate how these objects are aligned in the visualization according to the metadata from the Volume Object Model. We adjusted the opacity value of all objects and specified a border value in their appearance parameters. An overview of the objects is depicted in Figure 5.10 where the objects are sorted by their voxel count and arranged side-by-side in order to avoid occlusion. These visualizations were generated on the client side. It is noteworthy that the server-side generated Volume Object Models for the visualizations of Figures 5.9 and 5.10 are identical. However, the client-side visualizations greatly differ from each other, conveying different aspects of the data. This means that computationally inexpensive operations can be performed on the client side that result in visualizations that can serve different purposes.

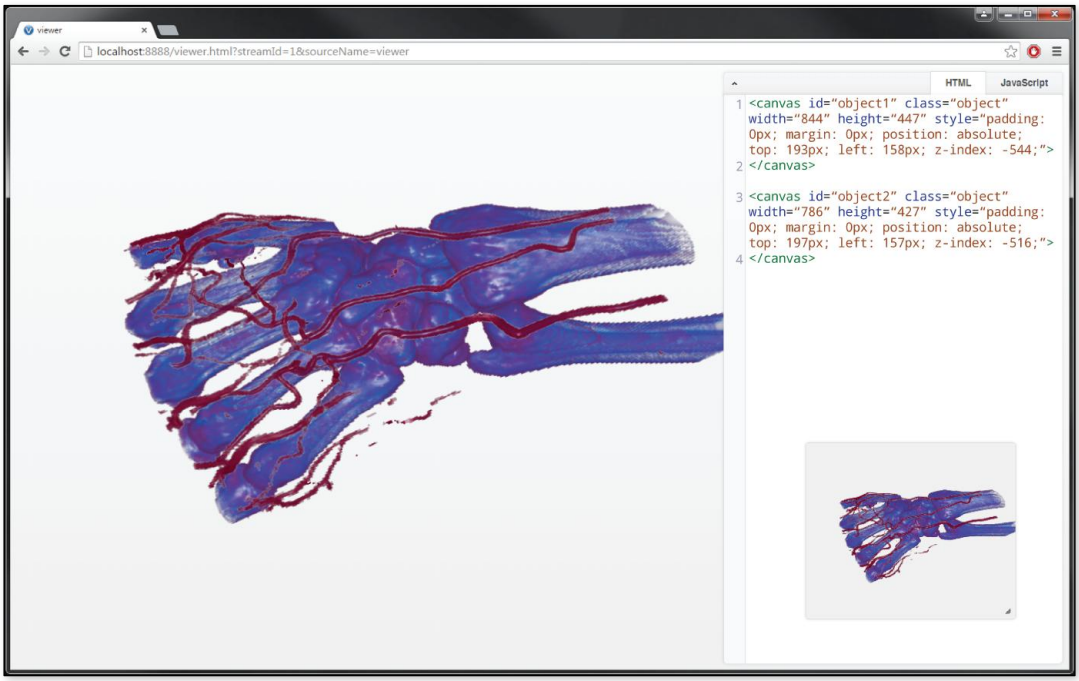


Figure 5.7: Interface of our client, which displays a data set of a human hand containing two objects. The interface consists of three views. The main view shows a visualization of the Volume Object Model. The porgramming interface on the right allows the scripting of HTML and JavaScript commands that are executed interactively. The context visualization retains a view of the original composition of the volume even if object properties are modified in the main view.

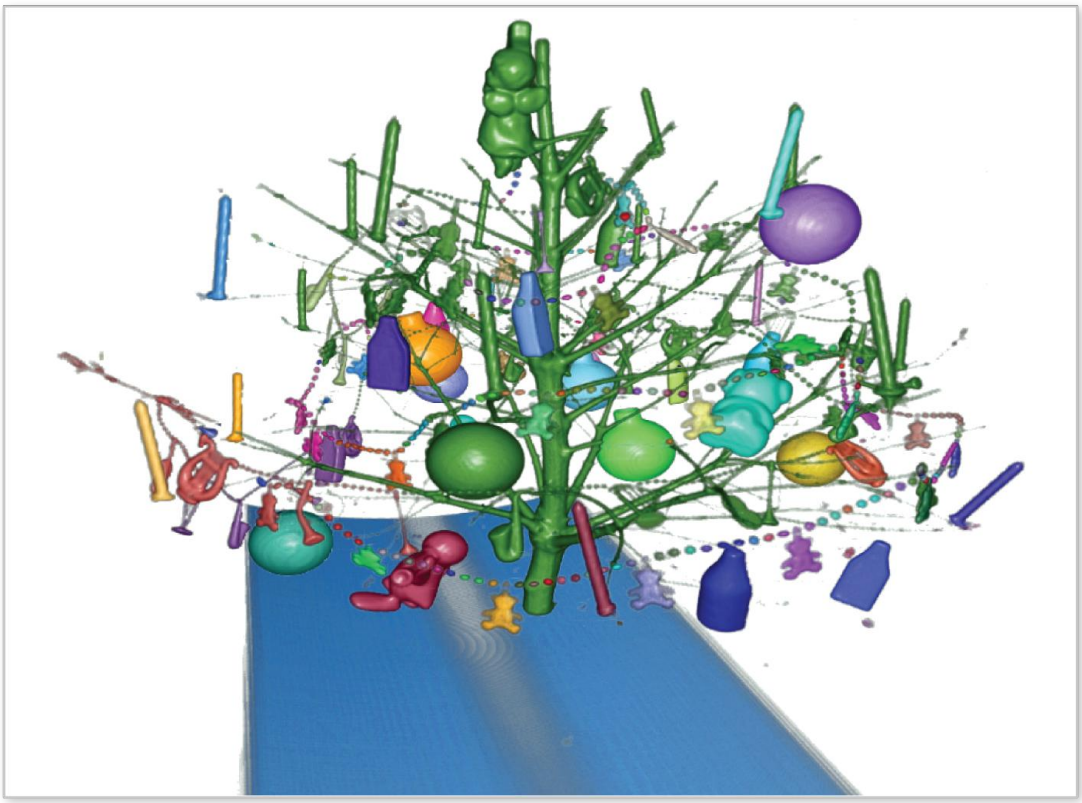


Figure 5.8: The canvas elements containing the objects are positioned in the client window according to the metadata. This allows us to recreate the full scene.

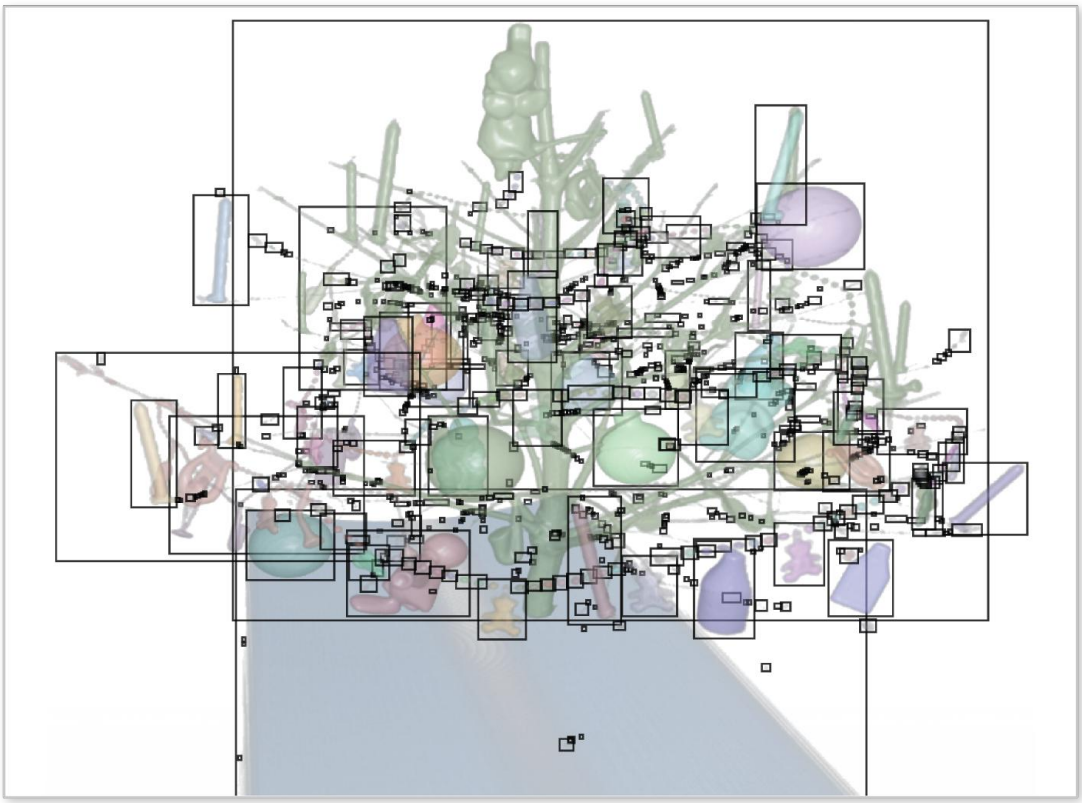


Figure 5.9: The visualization illustrates the positioning of the objects by outlining each object in in the scene.

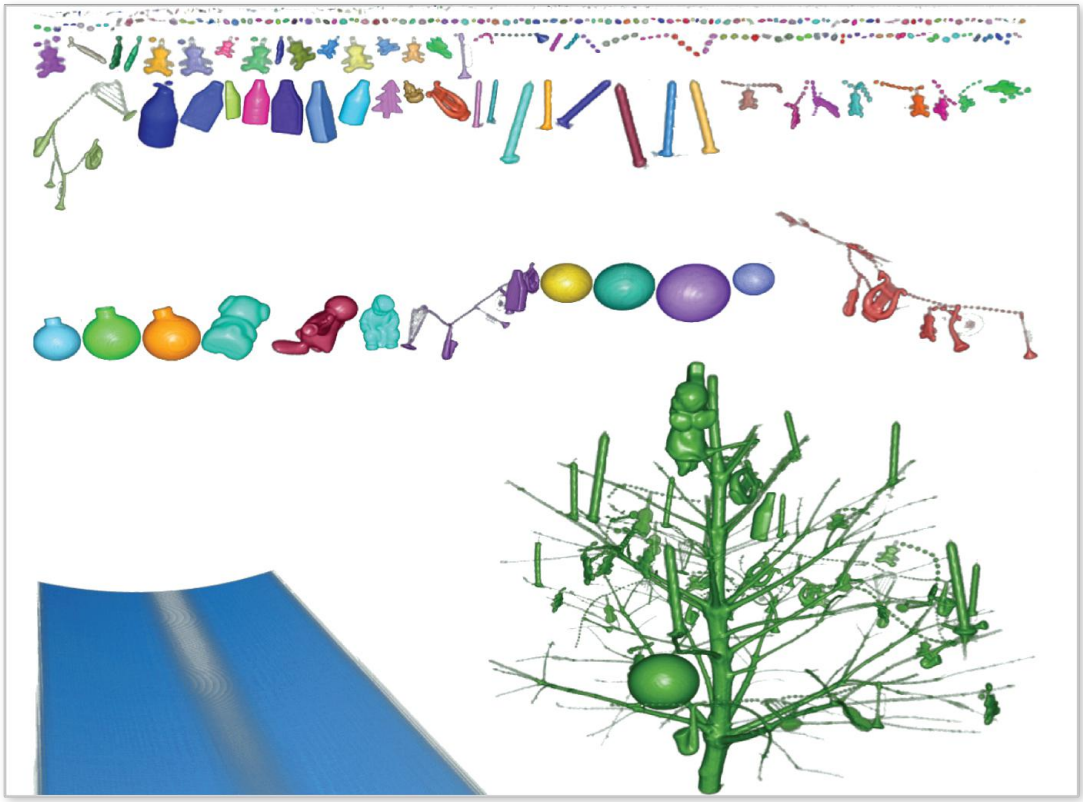


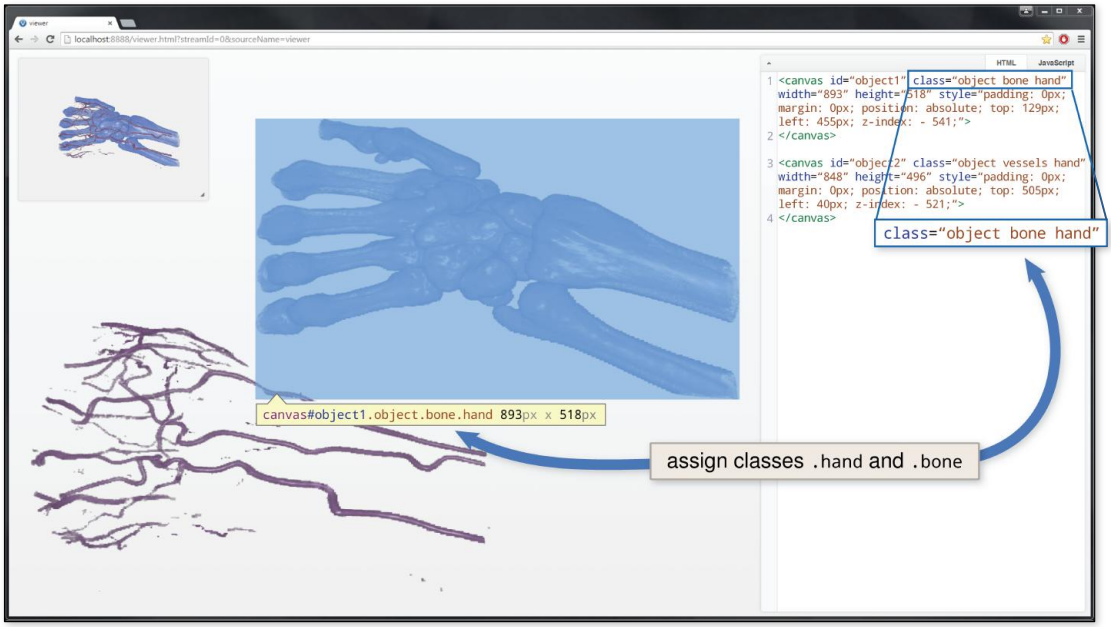
Figure 5.10: For an overview of all the objects, we can sort the objects by their voxel count and arrange them side-by-side.

5.3 Client-Side Interaction

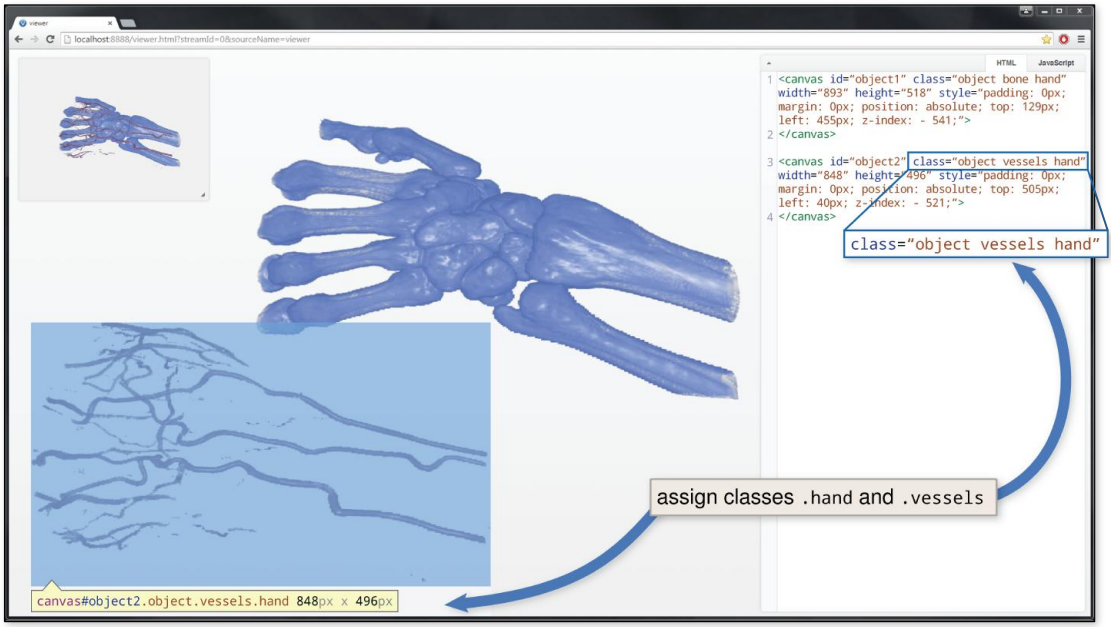
In our web client, we can perform object-wise tasks on the data set without triggering a server-side re-rendering for every operation. Our framework allows us to interact with and modify the objects in the visualization using the Volume Object Model that the client received from the server. We show how we can script commands on the client-side in the browser and apply these to our objects.

5.3.1 Addressing Objects

The client-side interaction with the Volume Object Model was implemented using Document Object Model (DOM) which is integrated with web browsers. We use the JavaScript libraries JQuery and D3 for DOM (hence Volume Object Model) interactions. We can address each object via the unique handle that we attached to DOM element. All objects also are assigned a common class `object`, through which they can be addressed. HTML objects only have one ID, but can have an arbitrary number of classes.



(a) Object `.bone` *highlighted*



(b) Object `.vessels` *highlighted*

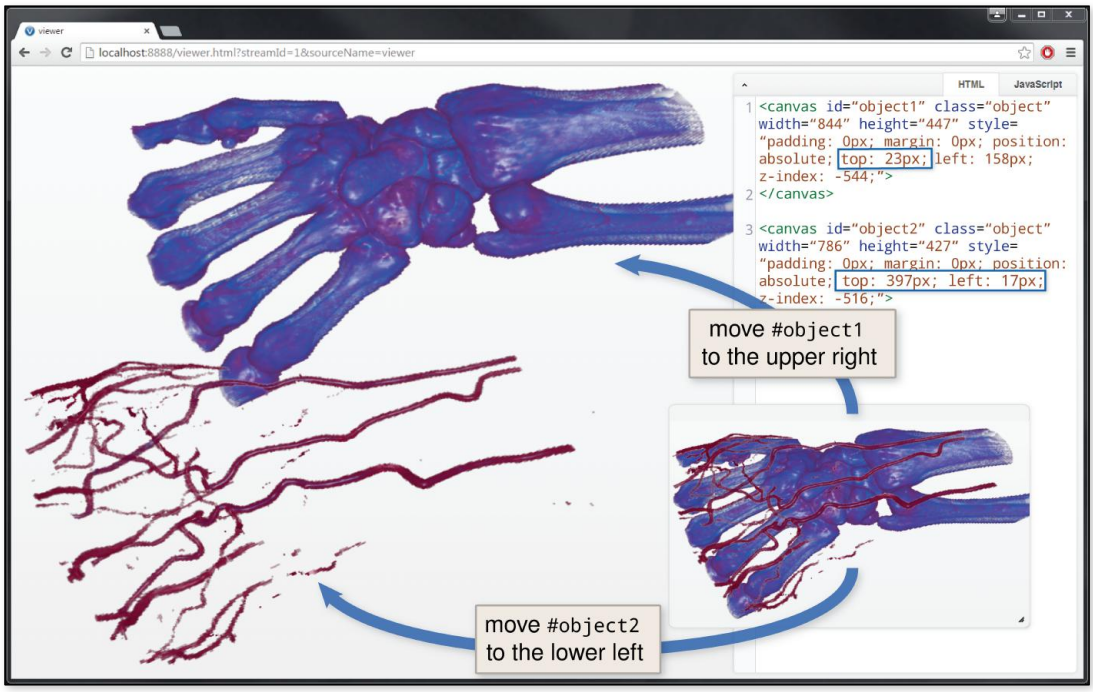
Figure 5.11: This example shows two objects that have been assigned additional DOM handles as classes. `#object1` *was assigned the class* `.bone` *(a) whereas* `#object2` *was assigned the class* `.vessels` *(b). Classes can contain multiple objects – the class* `.hand` *was added to both objects – thereby effectively providing the possibility of combining multiple objects into a group, which can be addressed through the common handle.*

We can append additional classes to objects. Figure 5.11 shows an example where objects were assigned multiple new classes. This can be done by editing the HTML source in the console, or interactively through scripting (e.g. through the JQuery command `$('#object1').addClass('newClass')`), which allows for a grouping of elements into custom categories, as depicted in Figures 5.11a and 5.11b. To retrieve objects of a certain class, we use either the JQuery selector `$` or the D3 selectors `d3.select` and `d3.selectAll`.

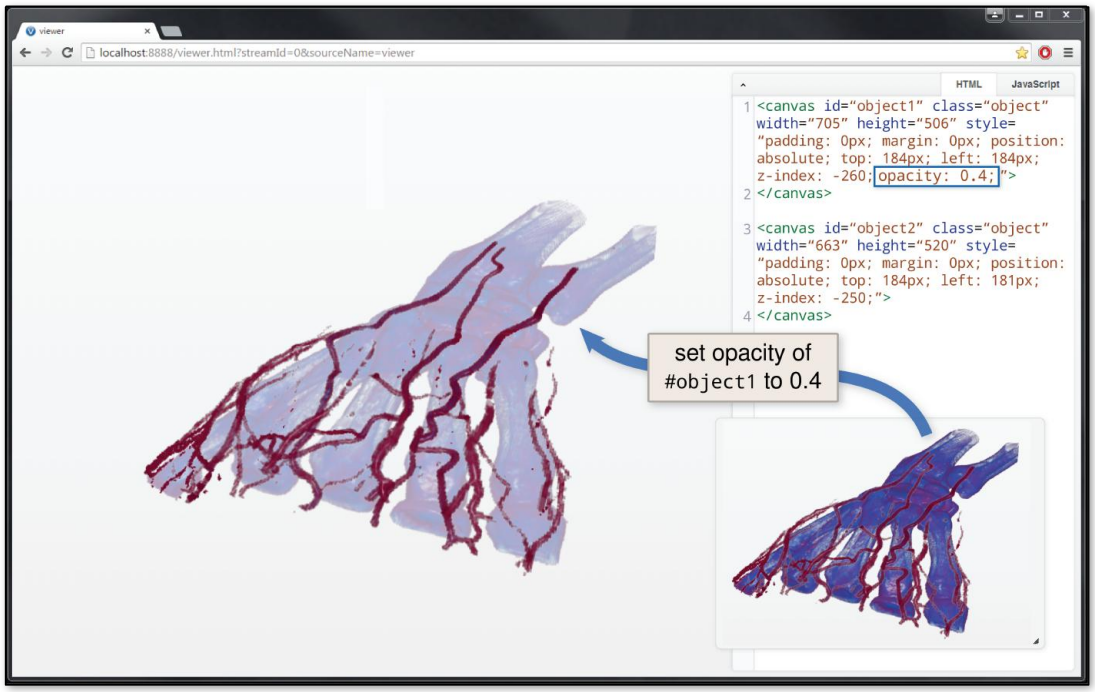
5.3.2 Programming Object Appearance

We can program the appearance of the objects on the client by accessing them via the ID or class handles. In the application case shown in Figure 5.12a, we want to apply a translation operation on the objects in our scene. Translating the objects allows us to move them apart in our visualization in order to reduce occlusion or visual clutter and investigate the objects separately. This can be done by either modifying the CSS properties `top` and `left` for each object directly in the embedded HTML console or by applying a JavaScript function on the objects.

Analogously, we can also change visual properties of the objects, such as adjusting the opacity value. Figure 5.12b depicts how the opacity value of one object can be reduced in order to make other objects visually stand out more. The discernibility of the vessels of the human hand is increased in this case by reducing the opacity of the bone.



(a) Object translation of two objects, the bone and the vessels



(b) Opacity change of one object, the bone

Figure 5.12: We can program operations that change the appearance and behavior of the objects. Here, we show how the objects in the scene are originally composed (in the overview) and can be modified via scripting. We can translate objects (a) as well as change their visual properties (b).

Complex Appearance Modifications

We will now describe a workflow outlining how to program more complex modifications according to object parameters carried out on the objects in a visualization. The procedure we describe is shown in Figures 5.13 and 5.14.

In this example, we look at the Christmas tree [KTM+02] data set, which, in this case, was divided into 1047 objects using thresholding with ViSlang on the server side. In order to examine this data set, we first remove one object (the base of the tree) from the visualization, since it is uninteresting to our current investigation. We locate the ID of the appropriate object by assigning an object class to big objects according to their voxel count.

```
/* we assign objects separate classes according to their voxel count */
d3.selectAll('.object')
  .filter(function(d) { //filter function
    var object = d3.select(this);
    return object.attr('data-voxels') > 50000;
  })
  .classed('big', true); //add class 'big' to objects
```

Only the base object and the tree are classified as big. Therefore, we can easily assign the base object the class `base`. For our purpose, we set this object to `hidden`.

```
d3.select('.base')
  .style('visibility', 'hidden'); //hide object
```

The result of this script is shown in Figure 5.13a. Since it is hard to get an overview of the objects contained in the volume, we want to align them without occlusion. In order to do that, we first sort the objects by height.

```
var heights = [];
/* create array containing indices of all objects and their heights */
d3.selectAll('.object').each(function(d, i) {
  var object = d3.select(this);

  heights.push([i, object.attr('height')]);
})

/* sort the array according to heights */
heights.sort(function cmp(a, b) {
  return b[1] - a[1];
})
```

We implement a function that iterates over the sorted list and translates each object within the viewport to be displayed next to each other as seen in Figure 5.13b.

```
var dimensions = [$('body').width(), $('body').height()];
var xPos = 0,
    yPos = dimensions[1];
```

```
/* iterate over sorted list and translate objects to positions */
$.each(heights, function(d, entry) {
  var object = $(objects[entry[0]]);

  var obj_width  = parseInt(object.attr('width'));
  var obj_height = parseInt(object.attr('height'));

  if(xPos + obj_width > dimensions[0]) {
    xPos = 0;
  }
  if(xPos == 0) {
    yPos -= obj_height; //decrement y position of object
  }

  object.css({
    top:  yPos,
    left: xPos
  })
  xPos += obj_width; //increment x position of object
})
```

In this view, we can see that many of the objects are very small. These objects are displayed on the top of the visualization in Figure 5.13b. Since we choose to investigate mid-sized objects, we remove the small objects from the visualization by applying a size threshold. Only objects bigger than the threshold remain, as depicted in Figure 5.13c.

```
d3.selectAll('.object')
  .filter(function(d) { //filter function
    var object = d3.select(this);
    return object.attr('data-voxels') < 5000;
  })
  .classed('small', true) // add class 'small' to filtered objects
  .style('visibility', 'hidden'); //hide objects
```

We can return to the original arrangement of the volume and view the data set without the filtered out objects, as shown in Figure 5.14d.

```
d3.selectAll('.object').each(function(d, i){
  var object = d3.select(this);

  object.transition()
        .duration(2000)
        .style({ // move object to original stored position
          top:  object.attr('data-top')  + 'px',
          left: object.attr('data-left') + 'px'
        })
})
```

We then decide that the tree is not as relevant to us as the decorative items on the tree, but we want to keep it for context, therefore reducing its opacity.

```
d3.select('.tree')
  .style('opacity', '0.4');
```

When addressing multiple objects in order to perform the same operation on each of them, we need to group them either by using a selection-and-filtering operation each time they are addressed, or by assigning them a class once. In Figure 5.14e, we assign all decorative items a common class, which allows us to easily address them as a group in future operations.

```
d3.selectAll('.object')
  .filter(function(d) { //filter function
    var object = d3.select(this);
    var voxels = object.attr('data-voxels');
    return (voxels > 5000 && voxels < 50000);
  })
  .classed('decoration', true); //add class 'decoration' to objects
```

In a last step, which is shown in Figure 5.14f, we visually emphasize all objects contained in the class `.decoration` by scaling their size to 120%.

```
/* scale each object by 120% around object center */
d3.selectAll('.decoration').each(function(d, i) {
  var object = d3.select(this);

  var object_width  = object.attr('width');
  var object_height = object.attr('height');

  var new_width  = object_width  * 1.2;
  var new_height = object_height * 1.2;

  var diff_width  = (new_width  - object_width)  / 2;
  var diff_height = (new_height - object_height) / 2;
  object.style({ // update object position and size
    width:  new_width,
    height: new_height,
    left:   object.style('left') - diff_width,
    top:    object.style('top')  - diff_height
  })
})
```

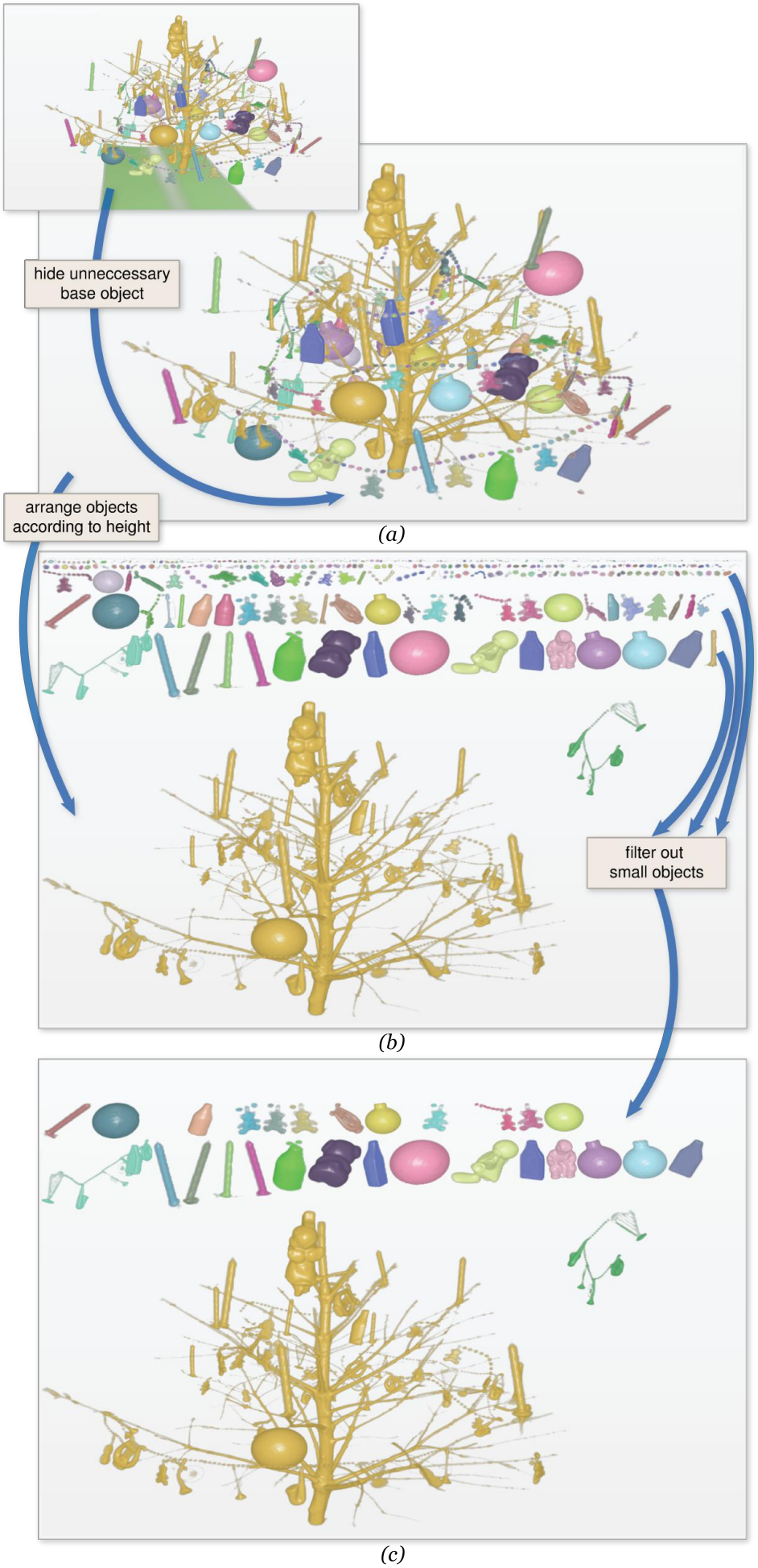


Figure 5.13: We can selectively hide objects from our visualization. In order to get a better overview, we can rearrange the objects according to specific object parameters and filter uninteresting objects in our current investigation. (continued in Figure 5.14)

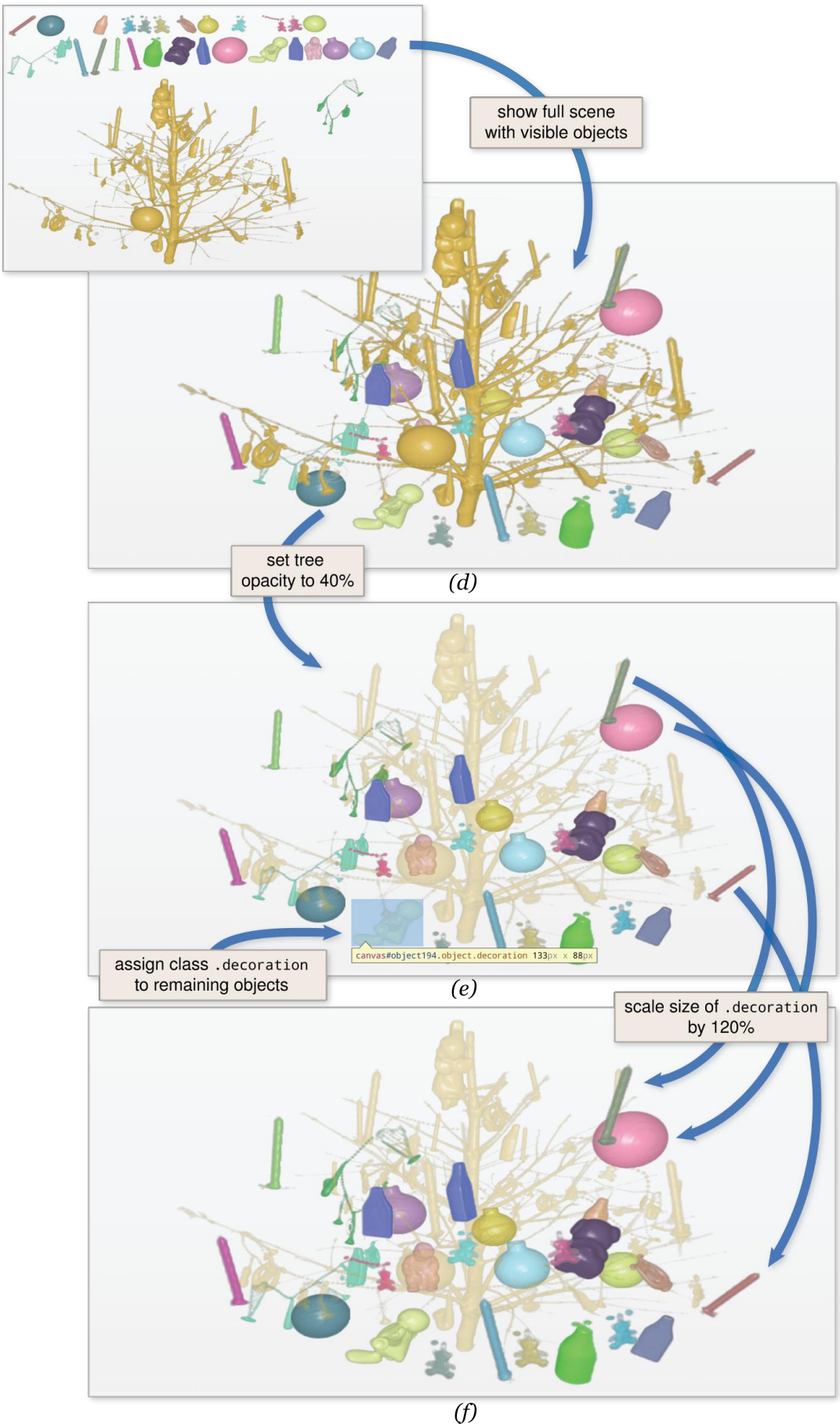


Figure 5.14: We can rebuild our visualization from the filtered objects and perform visual modifications on subsets of the objects. In this instance, we set one object to semi-transparent while enlarging the objects we classified as decoration.

5.3.3 Programming Visualizations

By means of another exemplary workflow on the client, we show how we can utilize the powerful web technologies embedded on the client to create entirely new visualizations from the data we provide. This process is illustrated in Figures 5.15 and 5.16.

Once again, we want to investigate the Christmas tree data set. We want to explore the relationship between the object-wise properties *voxel count* and *density* by means of a scatterplot visualization.

Since our scene contains two very large objects (i.e., the base object and the tree) we filter them out as shown in Figure 5.15a. Their voxel count is significantly higher than the voxel count of the decorative items on the tree and would therefore distort our scatterplot visualization.

```
d3.selectAll('.object')
  .filter(function(d) { //filter function
    var object = d3.select(this);
    return object.attr('data-voxels') > 50000;
  })
  .style('visibility', 'hidden'); //hide big objects
```

Then, we traverse all the objects and gather information about their voxel counts and densities in order to find maxima and minima values for these properties.

```
var range_px = [Number.MAX_VALUE, 0],    //range for pixel count
    range_vx = [Number.MAX_VALUE, 0],    //range for voxel count
    range_dn = [Number.MAX_VALUE, 0.0]; //range for density


/* evaluate ranges for object properties */
d3.selectAll('.object').each(function(d, i){
  var o = d3.select(this);

  var px = o.attr('data-pixels'),
      vx = o.attr('data-voxels'),
      dn = o.attr('data-density');

  range_px = [Math.min(range_px[0], px), Math.max(range_px[1], px)];
  range_vx = [Math.min(range_vx[0], vx), Math.max(range_vx[1], vx)];
  range_dn = [Math.min(range_dn[0], dn), Math.max(range_dn[1], dn)];
});
```

We use these ranges to create mapping functions from the data range to the scatterplot dimensions, which we can use to look up our objects' coordinates in the scatterplot coordinate system.

```
var dimensions = [$('body').width(), $('body').height()]; //window size

var pad = 50; //padding for the plot
var w = dimensions[0] - 2 * pad, //plot dimensions
    h = dimensions[1] - pad;
```

```
var scatterplot = d3.select('body')
    .append('svg') //create svg element for scatterplot
    .attr('width',  w)
    .attr('height', h);

//functions to map values to scale for x-Axis, y-Axis and radius
var x = d3.scale.linear().domain(range_vx).range([2 * pad, w - pad]),
    y = d3.scale.linear().domain(range_dn).range([h - pad * 2, pad]),
    r = d3.scale.log().domain(range_px).range([0,30]);
```

Axes for the scatterplot can be created from this information by using the designated axes functionality provided by the D3 library. Figure 5.15b shows the unchanged data set with axes created for density and voxel count.

```
//set scales for axes
var xAxis = d3.svg.axis().scale(x).orient('bottom'),
    yAxis = d3.svg.axis().scale(y).orient('left');

/* create scatterplot axes */
scatterplot.append('g') //create x-Axis
    .attr('class', 'axis')
    .attr('transform', 'translate(0, ' + (h - pad) + ')')
    .call(xAxis);

scatterplot.append('g') //create y-Axis
    .attr('class', 'axis')
    .attr('transform', 'translate(' + ( 2 * pad - pad ) + ', 0)')
    .call(yAxis);
```

We can modify all object positions in the scene to be arranged according to the objects' voxel count (on the x-axis) and density value (on the y-axis). In order to retain an overview of the objects' trajectories, we can transition in a smooth animation from the spatial arrangement to the scatterplot and back. This animation is computed by D3 and illustrated in Figure 5.17. Using JavaScript and D3, this can be achieved in a few lines of code.

```
/* moves objects to their position in the plot by looking up the value
   in the mapping function */
d3.selectAll('.object').each(function(d, i){
  var o = d3.select(this);

  //translate to new position for object
  o.transition()
   .duration(2000)
   .style('top',  y(o.attr('data-density')) + 'px')
   .style('left', x(o.attr('data-voxels'))  + 'px')
})
```

At any point in the visualization process, we can easily revert to the original arrangement of the objects within the volume by looking up the coordinates from the stored values.

```
/* optional: moves objects back to their original position */
d3.selectAll('.object').each(function(d, i){
```

```
  var o = d3.select(this);

  o.transition()
   .duration(2000)
   .style('top',  o.attr('data-top')  + 'px')
   .style('left', o.attr('data-left') + 'px')
})
```

The scatterplot in Figure 5.16a shows that similar decorative objects from the Christmas tree are nicely grouped together within the scatterplot, even though the three-dimensional perspective of the volume makes them appear in different sizes.

If the representation of each object by its visualization makes the view too cluttered, we can hide each object and substitute it by a glyph. This is illustrated in Figure 5.16b. In this example, we chose circles with a diameter logarithmically corresponding to the number of screen-space pixels of the objects. The logarithmic scale was selected because the majority of objects are very small (clustered in the scatterplot on the bottom left). This lets us achieve a meaningful mapping of the size difference in small objects to the radii of the circles.

```
/* hides objects and replaces them with svg circles */
d3.selectAll('.object').each(function(d, i){
  var o = d3.select(this);
            .style('visibility', 'hidden'); //hide object

  scatterplot.append('circle') //append glyph
      .attr('class', 'circle')
      .attr('cx', function (d) { return x(o.attr('data-voxels')); })
      .attr('cy', function (d) { return y(o.attr('data-density'));})
      .attr('r',  function (d) { return r(o.attr('data-pixels')); });
})
```

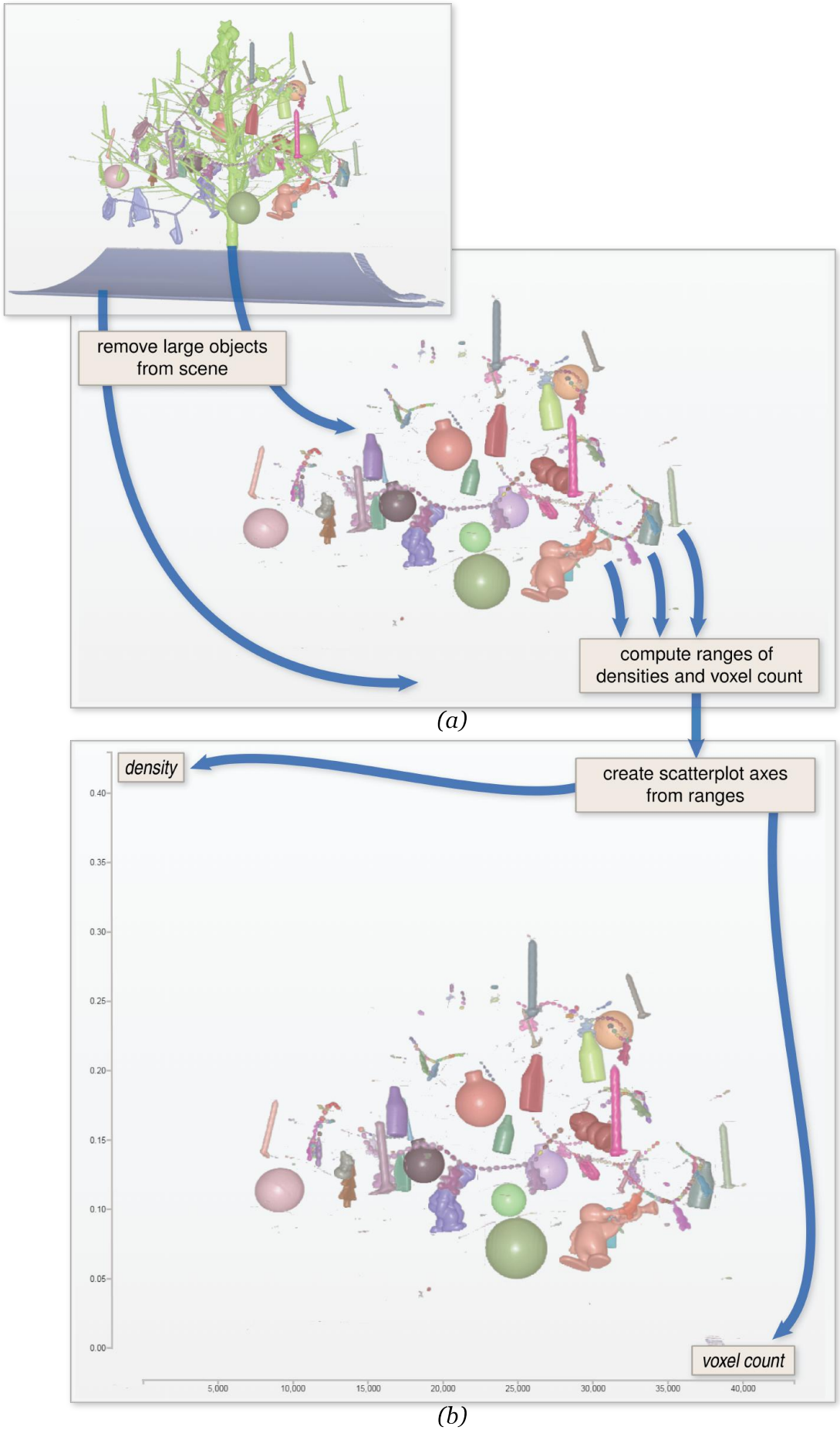


Figure 5.15: Workflow for generating a scatterplot of the objects in the Christmas tree data set. First, we accumulate the ranges of the data values we plot along the axes from the objects. From these ranges, we plot the axes for the scatterplot and generate mapping functions from the data range to the scatterplot range. (continued in Figure 5.16)

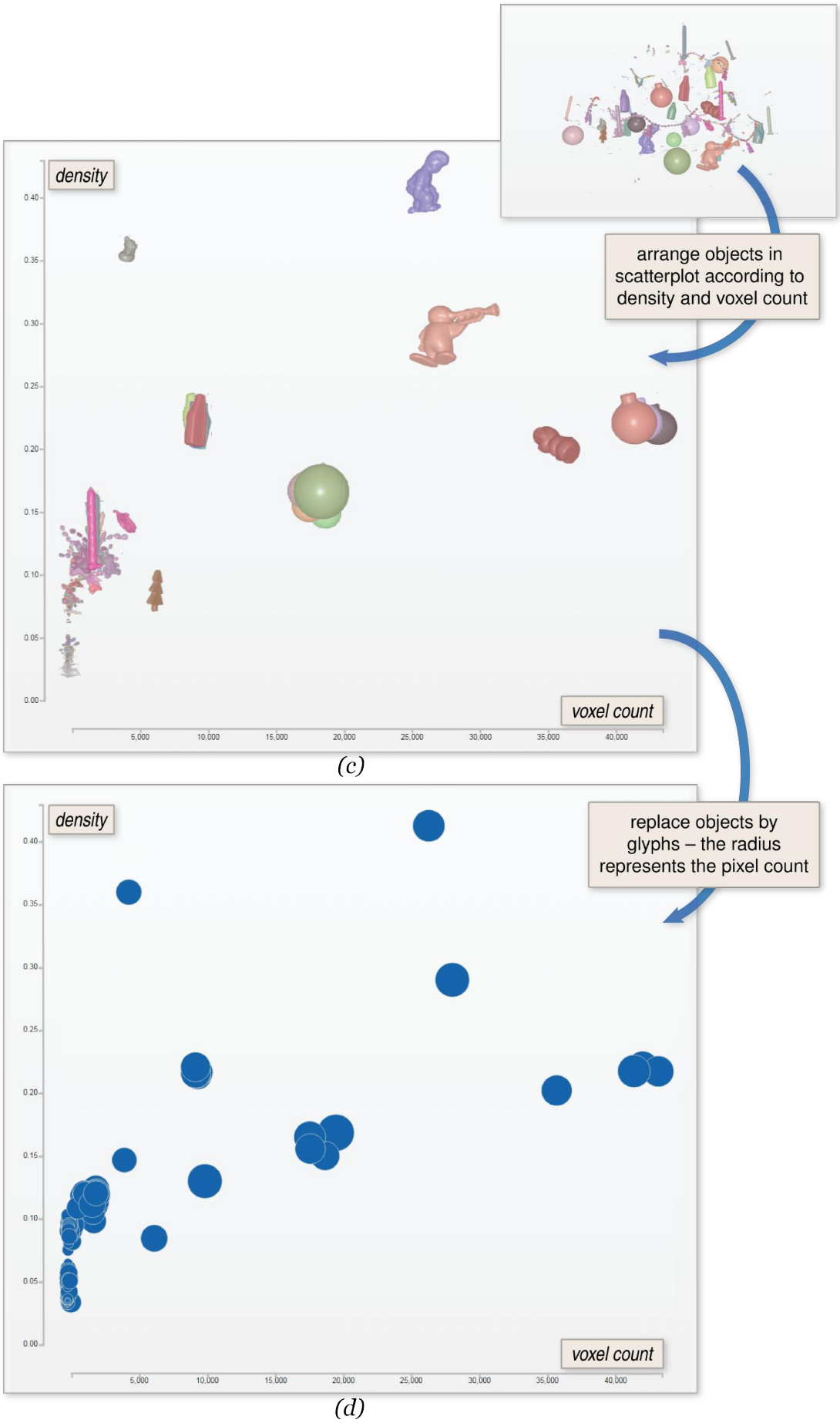


Figure 5.16: We look up the coordinates in the scatterplot for each object from the mapping functions we created. This allows us to move every object to its position in the scatterplot. We can also substitute the objects with glyphs, in this case circles, where the radius is defined by the objects' pixel count.

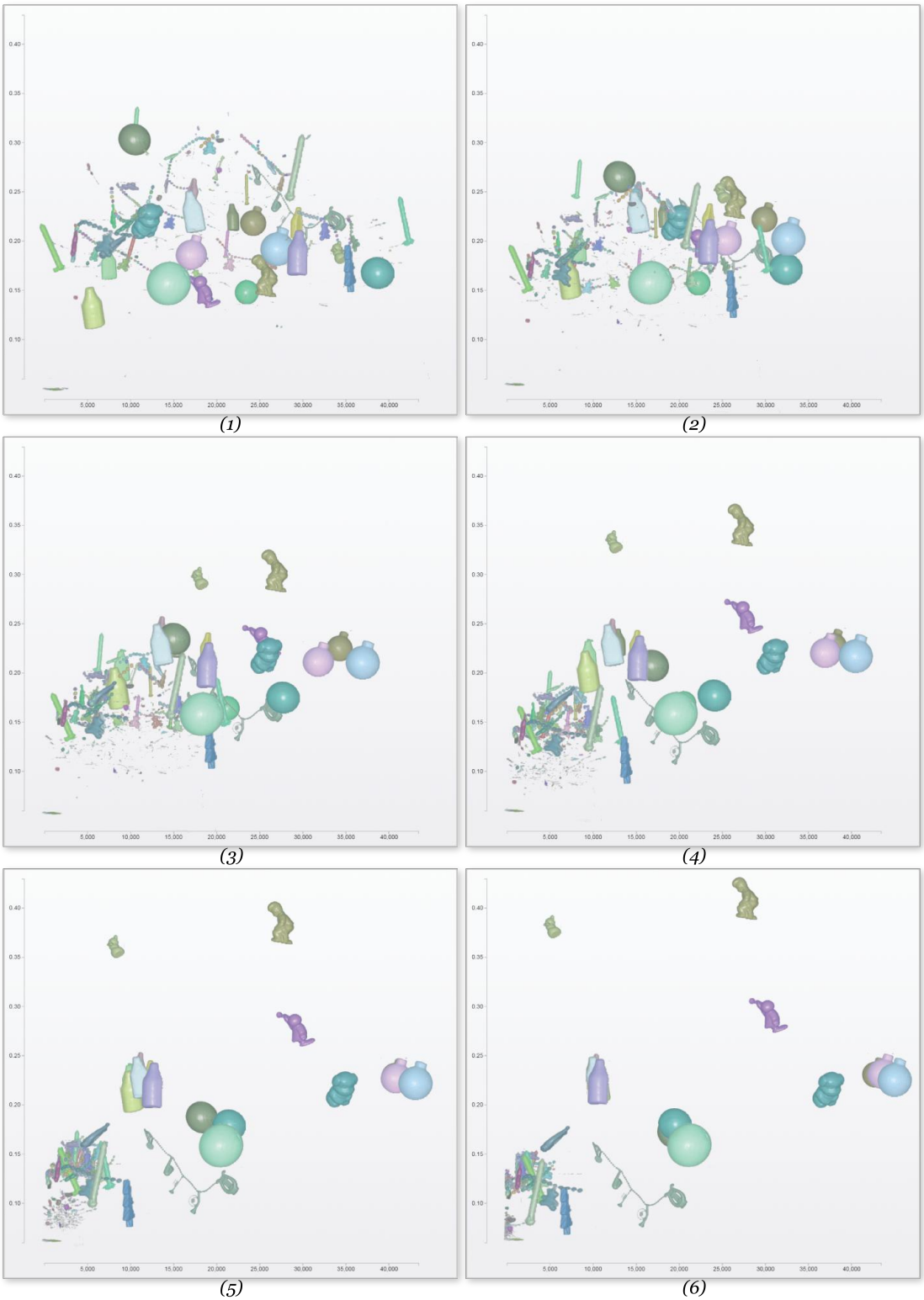


Figure 5.17: D3 allows for a smooth transition between the original arrangement (1) of the objects and the new visualization (6) in order to maintain the comprehensibility of the object rearrangement. Six timeframes from this transition are depicted.

5.4 Evaluation

Our approach surpasses existing techniques for remote visualization in that it provides a fully programmable environment for executing customized deferred visualization tasks.

In contrast to conventional remote rendering techniques, our approach does not require a re-rendering unless the viewpoint is changed. Whereas the Visualization by Proxy [TCM10c] technique only allows for certain settings of the visualization to be deferred to a later stage in the pipeline, our Volume Object Model permits the user to interactively modify the visualization according to any parameters that are transmitted in the metadata.

By performing the visualization per object, we facilitate selection, filtering and rearrangement tasks in a thin client. The use of the web browser as the environment for our client does not only allow the user to access the visualization platform-independently, but also the full toolset of modern web technology can be accessed for programming the visualization.

As shown in the scatterplot example, we can repurpose the object images to serve as glyphs in information visualization techniques. By transitioning smoothly from one visualization to the other, the arrangements of the objects in the different views stay transparent and comprehensible.

Through the reconciliation of information visualization techniques with three dimensional object representations, we provide the users with tools for obtaining novel insights into their data.

5.4.1 Performance

We tested the performance of our server-side Volume Object Model generation with multiple data sets at a viewport size of 1280×800. The performance currently depends heavily on choosing adequate values for the memory settings (e.g. the constant defining the maximum number of possible labels in the data set) in the OpenCL kernels.

Data Set	Size	Objects	First Pass	Packing	Second Pass
Three balls	128×128×128	3	3 ms	6 ms	2 ms
Human hand	244×124×257	2	11 ms	6 ms	16 ms
Small Christmas tree	128×124×128	283	85 ms	7 ms	20 ms
Christmas tree	512×499×512	1047	276 ms	12 ms	57 ms

On the client-side, we measured the timing for executing the customized function sequences we implemented. The execution time of the complete Christmas tree appearance modification example described in Section 5.3.2 for 1049 objects is 68.40 ms. The execution time of the complete scatterplot example described in Section 5.3.2 for 1049 objects is 53.43 ms. The calculation for the transitioning of the objects to their regular position as depicted in Figure 5.17 takes 14.5 ms.

5.4.2 Discussion and Comparison to State of the Art

The novelty of our approach lies in the design of a system that deliberately splits the visualization task into two stages. The first stage is computed on the server side and generates the intermediate representation of the data (i.e., the Volume Object Model). The second stage is computed on the client side and generates visualizations of the Volume Object Model.

The design of our system makes several assumptions about data sets, client and server architectures and the memory bandwidth. We assume that there is a powerful server for computation and a thin client for visualization. This assumption only makes sense if

(i) the data size is too large to be moved from the server to the client, or
(ii) the client does not have enough memory to hold the data, or
(iii) the client does not have enough computational power to interactively query and visualize the data set.

If any of the above conditions is met, a remote visualization approach might be a suitable choice. We demonstrate that our approach benefits the visualization system design under each of the above-mentioned conditions:

(i) In the case of large data sets that cannot be transmitted from the server to the client, we have to prove that the data transmission requirements of our approach are less than the data set size. We show that this is true even for medium sized volume data sets with a large number of objects. Instead of transferring the data set which consists of N voxels, we only transmit the Volume Object Model, a condensed representation of the volume, once per request. Although the memory savings depend on the number of objects, it is a weak assumption that the Volume Object Model is much more compact than the raw data. The Volume Object Model and image data are only transmitted for the K objects, where K is typically much smaller than N. The view dependent parts of the Volume Object Model (e.g., the object images) need to be transmitted per viewpoint and therefore might grow larger than the actual data for intensive investigations. However, this limitation is true for any remote visualization scenario and is not unique to our approach.
(ii) In case the client does not have enough memory to hold the entire data set, our approach is clearly beneficial. The memory that has to be allocated on the client side consists of memory for the Volume Object Model and of memory for the images of the objects. The memory requirements for the Volume Object Model are small compared to the whole data set, which directly follows from (i). The memory requirements for the image data are typically smaller than the memory requirements for the volume data. This fact is exploited in all remote visualization scenarios.
For instance, the memory requirement for the Human hand data set (11.39 MB) shown in Figure 5.1 is larger by a factor of 12 compared to the memory required for the corresponding Volume Object Model with its image data (0.92 MB). For the Christmas tree data set shown in Figure 5.2, the memory requirement (255.49 MB) is larger by a factor of 187 compared to the Volume Object Model with image data (1.36 MB). Clearly, this factor increases for larger data sets and becomes even more beneficial for in-situ

visualization scenarios.

(iii) In case the client does not have enough computational power to query a volume dataset for objects and to render the objects directly, we have to prove that the object property computation and object visualization is more expensive than the object level manipulation and final visualization on the client side. This requirement is easily shown for typical data sets, since the object level computations are only performed on the metadata of a relatively low number of objects compared to the number of all voxels in a volume. The visualization of the image data on the client side is also known to be less expensive, which is a heavily exploited property in billboard- or impostor-based real-time rendering approaches. These techniques replace complex models by a more compact image based representation when possible [AMHH02]. Our system is designed to be asymmetric and to perform the major workload on the server. For instance in the example of Figure 5.17, the server computes the volume object model once in 345 ms, while the client runs the object level animations with approximately 25 frames per second.

Our prototype implementation proves that our system design is a feasible approach and the assumptions hold true. Compared to other remote visualization systems, we propose an approach that defers object-level tasks to a later stage than the voxel-level tasks. This task separation offloads the compute intensive voxel-level operations to a server, yet allows deferred interaction and visualization on an object level in a thin client. The flexibility of the Volume Object Model is a key aspect to improve the applicability of deferred visualization approaches in the future.

Other remote visualization systems [ESE00, SME02, SDWE03] completely lack the flexibility to do deferred interactions and visualizations and solely transmit image data. Although the work of Tikhonova et al. [TCM10c] shows a related deferred visualization model, the focus of their system is on the reconstruction of volume rendered images under slight changes of the visualization parameters. We, in contrast, demonstrate object level operations on the client side that go beyond these capabilities and enable entirely new avenues for in-situ visualization. In fact, the work of Tikhonova et al. [TCM10c] is orthogonal to our approach and could be incorporated for even more flexible deferred visualizations.

CHAPTER 6

Conclusion and Future Work

In this thesis, we presented a novel approach for enabling deferred object-manipulation tasks on a segmented data set in an in-situ remote visualization environment. We describe how we apply the deferral concept to the visualization pipeline. For object manipulation and modification tasks in in-situ visualization scenarios, a complete re-rendering of the volume is too costly, especially if the viewpoint is not changed. It is advantageous to enhance the visualization with object-wise metadata that allows for certain object-related operations to be performed on the visualization without re-rendering.

Our system reduces the bandwidth requirements and the load on the server. In conventional remote visualization, a re-rendering would be necessary on every change of the visualization parameters. The Visualization by Proxy [TCM10c] approach only allows certain settings changes to be deferred. We provide an object-based volume representation that, at comparatively little additional computational cost and memory requirements, allows for interactive object visualization tasks without requesting additional information.

The construction of the volume using the Volume Object Model representation on the client side poses some constraints to the programmability of the visualization. These include viewpoint changes, adjustments to the segmentation algorithm and transfer function, shading and lighting modifications. We can integrate these tasks with our system, but we need to send a new streaming request to the server and re-render the data set.

The operations our approach can perform on the client based on the Volume Object Model include the modification of the objects' appearance (e.g. visually enhance specific objects according to certain properties) and the resizing and rearrangement of objects within the viewer (e.g. arranging all objects next to each other in order to avoid occlusion). To enable the system to perform object-related operations, it is necessary to provide metadata describing the objects. These object-wise properties need to be gathered during the first pass of the visualization process and are provided by the server. The pre-visualization step allows the user to prefilter the data set based on the metadata. After filtering on an object level, images

of a subset of objects are displayed in a second pass.

We show how a deferred visualization pipeline can be made programmable on the client-side for extensive flexibility. Our system makes use of fully developed web technologies to provide users with powerful tools for generating customized visualizations from their data.

Our approach is only suitable for segmented volumes since all tasks we defer to the client are object-wise operations. Data sets where segmentation algorithms are inapplicable cannot be investigated using our system.

For in-situ environments, calculations on the client side may get slow due to technological limitations. The HTML5 canvas is hardware-accelerated, but pixel-wise operations on the canvas are expensive and JavaScript does not scale very well in these situations. Therefore, for very large data sets with many complicated occlusion cases, the performance of the visualization may be impaired.

6.1 Outlook

In the future, we will work to overcome the limitations of the current implementation of our system. We currently transmit the complete Volume Object Model, that is, both the image data and the metadata, to the client simultaneously. For the data sets we tested our system with, this strategy was acceptable. Future work includes an integration of the pre-visualization step in that the filtered visualization data is requested only after the interesting subset of the data set has been selected. Our current approach to pack all render targets into one image buffer does not yet allow us to select a subset of objects for streaming. This will help solving performance issues for very large data sets.

A solution for the integration of flexible renderer settings within our application would be desirable. Additionally, we want to enable rotation of specific objects only and requesting a re-rendering for these while preserving the other objects as they are. Furthermore, we want to give the user the flexibility of specifying the parameters of the segmentation and visualization that are currently hardcoded in the session file in the request for the data. This can be achieved by encoding the parameters in a string and sending them in the request for the visualization. We also want to permit the user to specify the properties that should be included in the metadata.

The user interaction of our viewer is currently limited to rotation and zooming. All object interactions our framework offers need to be scripted in the console. It would be beneficial to provide handles for additional interaction metaphors which should also be programmable. Examples are click-and-drag operations or the display of information on mouseover. The programmability of our approach actually allows these interactions to be implemented in JavaScript in the current version of our client, but an implementation of common functions and basic interaction techniques in a library would be expedient to the usability of our client.

While we demonstrated that our approach works in principle on a small scale, we did not test our implementation on an HPC system. The adaptation and evaluation of our system for an in-situ scenario is planned for future work.

APPENDIX A

Appendix

A.1 Rectangle Packing Algorithm

Algorithm A.1: Rectangle Packing

```
Input: sorted_list ⟨Rectangle⟩ rectangles, Block root
Data: Rectangle: input format, Block: binary tree node
Result: added coordinates to rectangles
foreach rectangle in rectangles do
    Block block = search (root, rectangle);
    if found block then
        subdivide (block, rectangle);
    else
        block = grow (rectangle);
    end
    rectangle.coordinates = block.xy;
end
```

Algorithm A.2: Rectangle Packing: *search*

```
1  function search (Block block, Rectangle rectangle)
2      if block not used and rectangle fits in block then
3          return block;
4      end
5      if block used and has vertical child then
6          Block option_v = search (child_vertical, rectangle) ;
7      end
8      if block used and has horizontal child then
9          Block option_h = search (child_horizontal, rectangle) ;
10     end
11     Block found = choose better filling ratio from {option_v or option_h};
12     return found;
```

Algorithm A.3: Rectangle Packing: *subdivide*

```
1  function subdivide (Block block, Rectangle rectangle)
2      set block to used;
3      if rectangle.dimensions < block.dimensions then
4          attach preferable subdivisions from
           {vertical_subdivision() or horizontal_subdivision()}
           as children to block;
5      end
```

Algorithm A.4: Rectangle Packing: *grow*

```
1  function grow (Rectangle rectangle)
2      Block extension = choose preferable extension from
       (vertical_extension(), horizontal_extension());
3      attach extension as child to {topMost or rightMost} block;
4      adjust root.dimensions;
5      subdivide (extension, rectangle);
6      return extension;
```

Bibliography

[AMHH02] Tomas Akenine-Moller, Eric Haines, and Naty Hofmann. *Real-Time Rendering*. A. K. Peters, Ltd., 3rd edition, 2002.

[Ast15] Astrofra (Wikimedia Commons). Deferred rendering illustrations. `https://commons.wikimedia.org/wiki/Special:Contributions/Astrofra`, 2015. (Accessed on 20/09/2015).

[BD13] Poorna Banerjee and Amit Dave. GPGPU based parallelized client-server framework for providing high performance computation support. *International Journal of Computer Science & Technology*, 4, 2013.

[BG05] Stefan Bruckner and M. Eduard Gröller. VolumeShop: An interactive system for direct volume illustration. In *Proceedings of IEEE Visualization 2005*, pages 671–678, 2005.

[BH09] Michael Bostock and Jeffrey Heer. ProtoVis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

[Bin13] Roba S. Binyahib. Image-based exploration of iso-surfaces for large multivariable datasets using parameter space. Master's thesis, King Abdullah University of Science and Technology, 2013.

[Bly06] David Blythe. The Direct3D 10 system. In *Proceedings of ACM SIGGRAPH '06*, pages 724–734. ACM, 2006.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[Bun05] Michael Bunnell. Dynamic ambient occlusion and indirect lighting. In Hubert Nguyen, editor, *GPU Gems 2*, pages 223–233. Addison-Wesley Professional, 2005.

[CA79] Guy B. Coleman and Harry C. Andrews. Image segmentation by clustering. *Proceedings of the IEEE*, 67(5):773–785, 1979.

[CGM+06] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James P Ahrens, and Jean M Favre. Remote large data visualization in the paraview framework. In*Proceedings of Eurographics/IEEE-VGTC Symposium on Parallel Graphics and Visualization (EGPGV) 2006*, pages 163–170, 2006.

[Chi07] Hank Childs. Architectural challenges and solutions for petascale postprocessing. In *Journal of Physics: Conference Series*, volume 78, page 012012. IOP Publishing, 2007.

[CML11] Matthäus G. Chajdas, Morgan McGuire, and David Luebke. Subpixel reconstruction antialiasing for deferred shading. In *Proceedings of Symposium on interactive 3D graphics and games (I3D) 2011*, pages 15–22, 2011.

[Cod15] CodeMirror. CodeMirror JavaScript library. `https://codemirror.net/`, 2015. (Accessed on 20/09/2015).

[Com07] Darrell Commander. VirtualGL: 3D without boundaries–the VirtualGL project. `http://www.virtualgl.org/`, 2007. (Accessed on 20/09/2015).

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Proceedings of ACM SIGGRAPH '88*, pages 21–30, 1988.

[EE99] Klaus Engel and Thomas Ertl. Texture-based volume visualization for multiple users on the world wide web. In *Proceedings of Eurographics Symposium on Virtual Environments '99*, pages 115–124. 1999.

[EHK+06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[Eng09] Wolfgang Engel. Designing a renderer for multiple lights – the light pre-pass renderer. ShaderX series, pages 217–224. Charles River Media, 2009.

[ESE00] Klaus Engel, Ove Sommer, and Thomas Ertl. A framework for interactive hardware accelerated remote 3D-visualization. In *Proceedings of Eurographics Symposium on Data Visualization (VisSym) 2000*, pages 167–177. 2000.

[Gor11] Jake Gordon. Binary tree bin packing algorithm. `http://codeincomplete.com/posts/2011/5/7/bin_packing/`, 2011. (Accessed on 20/09/2015).

[GPB04] Rich Geldreich, Matt Pritchard, and John Brooks. Deferred lighting and shading. Game Developers Conference, D3D Tutorial Day, 2004.

[HBE13] Matthias Holländer, Tamy Boubekeur, and Elmar Eisemann. Adaptive supersampling for deferred anti-aliasing. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):1–14, 2013.

[HCL05] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems 2005*, pages 421–430, 2005.

[HH04] Shawn Hargreaves and Mark Harris. Deferred shading. Game Developers Conference, D3D Tutorial Day, 2004.

[HJ07] Jared Hoberock and Yuntao Jia. High-quality ambient occlusion. In Hubert Nguyen, editor, *GPU Gems 3*, pages 257–274. Addison-Wesley Professional, 2007.

[HS98] Gerd Hesina and Dieter Schmalstieg. A network architecture for remote rendering. In *Proceedings of 2nd International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT) '98*, pages 88–91, 1998.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proceedings of Eurographics 2005*, pages 303–312, 2005.

[JMP88] Tianlai Jiang, Michael B. Merickel, and Edward A. Parrish. Automated threshold detection using a pyramid data structure. In *Pattern Recognition, 1988., 9th International Conference on*, pages 689–692 vol.2, Nov 1988.

[JQu15] JQuery. JQuery JavaScript library. `https://jquery.com/`, 2015. (Accessed on 20/09/2015).

[Koo08] Rusty Koonce. Deferred shading in tabula rasa. In Hubert Nguyen, editor, *GPU Gems 3*, pages 429–457. Addison-Wesley Professional, 2008.

[Kor03] Richard E Korf. Optimal rectangle packing: Initial results. In *Proceedings of ICAPS 2003*, pages 287–295, 2003.

[Kor04] Richard E Korf. Optimal rectangle packing: New results. In *Proceedings of ICAPS 2004*, pages 142–149, 2004.

[KTM+02] Armin Kanitsar, Thomas Theußl, Lukas Mroz, Milos Sramek, Anna Vilanova Bartroli, Balázs Csébfalvi, Jirí Hladuvka, Stefan Guthe, Michael Knapp, Rainer Wegenkittl, Petr Felkel, Stefan Roettger, Dominik Fleischmann, Werner Purgathofer, and M. Eduard Gröller. Christmas tree case study: Computed tomography as a tool for mastering complex real world objects with applications in computer graphics. `https://www.cg.tuwien.ac.at/xmas/`, 2002. (Accessed on 20/09/2015).

[KWT88] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International journal of computer vision*, 1(4):321–331, 1988.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of ACM SIGGRAPH '87*, pages 163–169. ACM, 1987.

[LD12] Gábor Liktor and Carsten Dachsbacher. Decoupled deferred shading for hardware rasterization. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D) 2012*, pages 143–150, 2012.

[Li94] Stan Z. Li. Markov random field models in computer vision. In *Proceedings of the Third European Conference-Volume II on Computer Vision - Volume II*, ECCV '94, pages 361–370. Springer-Verlag, 1994.

[Liu77] Hsun K. Liu. Two-and three-dimensional boundary detection. *Computer Graphics and Image Processing*, 6(2):123–134, 1977.

[Ma09] Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[MB97] Cherilyn Michaels and Michael Bailey. VizWiz: A java applet for interactive 3D scientific visualization on the web. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 261–267. IEEE Computer Society Press, 1997.

[MB98] Eric N. Mortensen and William A. Barrett. Interactive segmentation with intelligent scissors. *Graphical models and image processing*, 60(5):349–384, 1998.

[McG10] M. McGuire. Ambient occlusion volumes. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 47–56. Eurographics Association, 2010.

[MDM02] Johan Montagnat, Eduardo Davila, and Isabelle E Magnin. 3D objects visualization for remote interactive medical applications. In *Proceedings of First International Symposium on 3D Data Processing Visualization and Transmission 2002*, pages 75–78, 2002.

[MHUnC12] Marc Manzano, José Alberto Hernández, Manuel Urueña, and Eusebi Calle. An empirical study of cloud gaming. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, NetGames '12, pages 17:1–17:2. IEEE Press, 2012.

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007.

[MJ12] Charles Marion and Julien Jomier. Real-time collaborative scientific WebGL visualization with WebSocket. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 47–50. ACM, 2012.

[MOM+11] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities (PDAC) '11*, pages 1–6, 2011.

[MPHK93] Kwan Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In*Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, pages 15–22, New York, NY, USA, 1993. ACM.

[Nag14] Omniah H. Nagoor. Image-based exploration of large-scale pathline fields. Master's thesis, King Abdullah University of Science and Technology, 2014.

[OS88] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations.*Journal of computational physics*, 79(1):12–49, 1988.

[Pan09] David Pangerl. Deferred rendering transparency. In Wolfgang Engel, editor, *ShaderX7: Advanced Rendering Techniques*, ShaderX series, pages 217–224. Charles River Media, 2009.

[Par12] Tony Parisi. *WebGL: Up and Running*. O'Reilly Media, Inc., 1st edition, 2012.

[Pla06] Frank Puig Placeres. Overcoming deferred shading drawbacks. In Wolfgang Engel, editor, *ShaderX5: Advanced Rendering Techniques*, pages 115–130. Charles River Media, 2006.

[PXP00] Dzung L. Pham, Chenyang Xu, and Jerry L. Prince. Current methods in medical image segmentation. *Annual review of biomedical engineering*, 2(1):315–337, 2000.

[Qt15] Qt. Qt application framework. `http://www.qt.io/`, 2015. (Accessed on 20/09/2015).

[RBGH14] Peter Rautek, Stefan Bruckner, M. Eduard Gröller, and Markus Hadwiger. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, 2014.

[RCMC00] Kirsten Risden, Mary P. Czerwinski, Tamara Munzner, and Daniel B. Cook. An initial examination of ease of use for 2D and 3D information visualizations of web content. *International Journal of Human-Computer Studies*, 53(5):695–714, 2000.

[RPSC99] Harvey Ray, Hanspeter Pfister, Deborah Silver, and Todd A. Cook. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[SBSO09] Jeremy Shopf, Joshua Barczak, Thorsten Scheuermann, and Christopher Oat. Deferred occlusion from analytic surfaces. In Wolfgang Engel, editor, *ShaderX7: Advanced Rendering Techniques*, ShaderX series, pages 445–453. Charles River Media, 2009.

[SDWE03] Simon Stegmaier, Joachim Diepstraten, Manfred Weiler, and Thomas Ertl. Widening the remote visualization bottleneck. In *Proceedings of the 3rd International Symposium on Image and Signal Processing and Analysis (ISPA) 2003*, pages 174–179, 2003.

[SH15] Shu Shi and Cheng-Hsin Hsu. A survey of interactive remote rendering systems. *ACM Computing Survey*, 47(4):57:1–57:29, 2015.

[Shi05] Oles Shishkovtsov. Deferred shading in S.T.A.L.K.E.R. In Hubert Nguyen, editor, *GPU Gems 2*, pages 143–166. Addison-Wesley Professional, 2005.

[SME02] Simon Stegmaier, Marcelo Magallón, and Thomas Ertl. A generic solution for hardware-accelerated remote visualization. In *Proceedings of Eurographics Symposium on Data Visualization (VisSym) 2002*, pages 87–96, 2002.

[SSF+15] Karan Sapra, Melissa C. Smith, F. Alex Feltus, Asher Sampong, Joshua A. Levine, and Anagha Joshi. G^3NA: A GPU optimized global gene network alignment tool. In *GPU Technical Conference*, 2015.

[SSW88] Prasanna K. Sahoo, Sasan Soltani, and Andrew K. C. Wong. A survey of thresholding techniques. *Computer Vision, Graphics and Image Processing*, 41(2):233–260, 1988.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. In *Proceedings of ACM SIGGRAPH '90*, pages 197–206, 1990.

[TCM10a] Anna Tikhonova, Carlos D. Correa, and Kwan-Liu Ma. Explorable images for visualizing volume data. In *Visualization Symposium (PacificVis), 2010 IEEE Pacific*, pages 177–184, March 2010.

[TCM10b] Anna Tikhonova, Carlos D. Correa, and Kwan-Liu Ma. An exploratory technique for coherent visualization of time-varying volume data. *Computer Graphics Forum*, 29(3):783–792, 2010.

[TCM10c] Anna Tikhonova, Carlos D. Correa, and Kwan-Liu Ma. Visualization by proxy: A novel framework for deferred interaction with volume data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1551–1559, 2010.

[Thi09] Nicolas Thibieroz. Deferred shading with multisampling anti-aliasing in DirectX 10. In Wolfgang Engel, editor, *ShaderX7: Advanced Rendering Techniques*, ShaderX series, pages 225–242. Charles River Media, 2009.

[thr15] three.js. three.js JavaScript/WebGL environment. `http://threejs.org/editor/`, 2015. (Accessed on 20/09/2015).

[Tik12] Anna Tikhonova. *Deferred Visualization and Interaction with Explorable Images* PhD thesis, University of California at Davis, 2012.

[Tre09] Damian Trebilco. Light-indexed deferred rendering. In Wolfgang Engel, editor, *ShaderX7: Advanced Rendering Techniques*, ShaderX series, pages 243–255. Charles River Media, 2009.

[TS15] Georg Tamm and Philipp Slusallek. Plugin free remote visualization in the browser. In *Proceedings of SPIE Conference on Visualization and Data Analysis*, pages 939705–1:15, 2015.

[TYCM11] Anna Tikhonova, Hongfeng Yu, Carlos D. Correa, and Kwan-Liu Ma. A preview and exploratory technique for large scale scientific simulations. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV) 2011*, pages 111–120, 2011.

[WBW96] Jason Wood, Ken Brodlie, and Helen 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 81–87. IEEE Computer Society Press, 1996.

[Wes91] Lee Alan Westover. *Splatting: A Parallel, Feed-forward Volume Rendering Algorithm*. PhD thesis, 1991. UMI Order No. GAX92-08005.

[WFM11] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV) 2011*, pages 101–109, 2011.

[Wir07] Oliver Wirjadi. Survey of 3D image segmentation methods. Technical report, Fraunhofer ITWM, 2007.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 1999.

[WPJR11] Andrew Wessels, Mike Purvis, Jahrain Jackson, and Syed Rahman. Remote data visualization through websockets. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 1050–1051, April 2011.

[YWG+10] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, (3):45–57, 2010.