Distributed Computing and Transparency Rendering for Large Displays

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

Vorgelegt von

Daniel M. Kauker

aus Lichtenstein

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. Anders Ynnerman
Tag der mündlichen Prüfung:	29.06.2015

Visualisierungsinstitut
der Universität Stuttgart

2015

Contents

List of Figures

Chapter 2

Chapter 3

Chapter 4

List of Tables

List of Abbreviations and Acronyms

API	Application Programming Interface
APU	Accelerated Processing Unit
CPU	Central Processing Unit
CSG	Constructive Solid Geometry
DDP	Dual Depth Peeling
DP	Depth Peeling
FTLE	Finite-Time Lyapunov Exponent
GPA	General Purpose Accelerator
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
MIMD	Multiple Instruction Multiple Data
OIT	Order-Independent Transparency
PPLL	Per-Pixel Linked List
PVLL	Per-Voxel Linked List
RS	Reduced Surface
SAS	Solvent Accessible Surface
SDK	Software Development Kit
SES	Solvent Exclusive Surface
SIMD	Single Instruction Multiple Data
VdW	Van-der-Waals Surface

Units

fps	Frames per Second
PiB	Pebibyte, 2^{50} Bytes
TiB	Tebibyte, 2^{40} Bytes
GiB	Gibibyte, 2^{30} Bytes
MiB	Mebibyte, 2^{20} Bytes

Abstract

Today's computational problems are getting bigger and the performance required to solve them increases steadily. Furthermore, the results are getting more detailed, so that methods for rendering, visualization, and interaction methods need to adapt. While the computational power of a single chip also increases steadily, the most performance is gained by parallelization of the algorithms. Although Graphics Processing Units are built and specialized for the task of graphics rendering, their programmability makes them also suitable for general purpose computations. Thus, a typical workstation computer offers at least two processing units, the Central Processing Unit and the Graphics Processing Unit. Using multiple processing units for a task is commonly referred to as "distributed computing".

One of the biggest challenges when using such heterogeneous and distributed systems is the variety of software and ways to use them for an optimal result. The first section of the thesis focuses on an abstraction layer to simplify software development on heterogeneous computing systems. The presented framework aims to encapsulate the vendor-specific details and the hardware architecture, giving the programmer a task-oriented interface which is easy to use, to extend, and to maintain.

Having the results computed in a distributed environment, the interactive visualization becomes another challenge, especially when semi-transparent parts are involved, as the rendering order has to be taken into account. Additionally, the distributed rendering nodes do not know the details about their surroundings like the existence or complexity of objects in front. Typically, the large scale computations are distributed in object space so that one node works exclusively on one part of the scene. As it is too costly to collect all computation results on a single node for rendering, those nodes also have to do the rendering work to achieve interactive framerates. The resulting parts of the visualization are then sent to specialized display nodes. These display nodes are responsible for compositing the final image, e.g. combining data from multiple sources, and show them on display devices. In this context, rendering transparency effects with objects that might intersect each other within a distributed environment is challenging. This thesis will present an approach for rendering object-space decomposed scenes with semi-transparent parts using "Per-Pixel Linked Lists".

Presenting these visualizations on large display walls or on a remote (mobile) device raises the final challenge discussed in this thesis. As the scenes can be either complex or very detailed and thus large in terms of memory, a single

system is not always capable of handling all data for a scene. Typically, display walls that can handle such amounts of data consist of multiple displays or projectors, driven by a number of display nodes, and often have a separate node where an operator controls which part of the scene is displayed. I will describe interaction methods where the user can directly control the visualization on a large display wall using mobile devices without an operator. The last part of the thesis presents interaction concepts using mobile devices for large displays, allowing the users to control the visualization with a smartphone or tablet. Depending on the data and visualization method, the mobile device can either visualize the data directly or in a reduced form, or uses streaming mechanisms so that the user has the same visual impression as a user in front of the display wall. With the mobile application, the user can directly influence any parameter of the visualization and can thus actively steer an interactive presentation.

In this thesis, I will present approaches for employing heterogeneous computing environments, from a single PC to networked clusters, how to use order-independent transparency rendering for local and distributed visualization, as well as interaction methods for large display walls and remote visualization devices. The approaches for heterogeneous computing environments make the development easier, especially in terms of support of different hardware platforms. The presented distributed rendering approach enables accurate transparency renderings with far less memory transfer than existing algorithms. For the interaction methods, the usage of ubiquitous mobile devices brings the described approaches to all types of display devices without the need for special hardware. Additionally, a concept for an integrated system containing the contributions of the thesis is proposed. It uses the abstraction layer as a middle ware for the computation and visualization operations in the distributed rendering environments. The user controls the application using the methods for mobile device interactions.

German Abstract
— Zusammenfassung —

Berechnungs- und Simulationsprobleme werden immer komplizierter, Algorithmen zur Lösung immer komplexer und es ist immer mehr Performance nötig, um die Probleme zeitnah zu lösen. Zugleich werden die Resultate dieser Berechnungen detaillierter, so dass auch die Methoden zum Rendern, zur Visualisierung und zur Interaktion angepasst werden müssen. Zwar steigt die Berechnungsgeschwindigkeit der einzelnen Chips stetig, der größte Performancegewinn ist heutzutage allerdings durch Parallelisierung von Algorithmen zu erreichen. Obwohl Grafikkarten auf grafisches Rendern ausgelegt sind, lassen sie sich aufgrund ihrer Programmierbarkeit gut für bestimmte Arten von generellen Berechnungsproblemen nutzen. Daher bieten heutige Workstations mindestens zwei Berechnungseinheiten, den Hauptprozessor und die Grafikkarte, und bilden damit ein „verteiltes“, heterogenes System.

Eine der größten Herausforderungen bezüglich solcher Systeme ist die Vielfalt an Software und die unzähligen Wege sie einzusetzen, um das optimale Ergebnis zu erzielen. Der erste Teil dieser Dissertation beschreibt eine Abstraktionsschicht, die die Entwicklung von Software für heterogene Systeme erleichtern soll. Das Framework kapselt die herstellerspezifischen Programmierschnittstellen, -details und die Hardware-Architektur und stellt dem Programmierer ein aufgabenorientiertes Interface bereit, das einfach zu verwenden, zu erweitern, und zu warten ist.

Wenn die Berechnungen des verteilten Systems abgeschlossen sind, ist die interaktive Visualisierung der Daten eine weitere Herausforderung – insbesondere, wenn semi-transparente Objekte enthalten sind. Typischerweise werden große, verteilte Berechnungen im Objektraum unterteilt, so dass jeder der teilnehmenden Knoten an einem Teil der gesamten Szene arbeitet. Es ist zu aufwändig, die Ergebnisse von allen Knoten einzusammeln, um sie durch einen einzelnen Renderknoten darstellen zu lassen. Daher müssen schon die Berechnungsknoten die Visualisierungsarbeit übernehmen, um (semi-)interaktive Darstellungen zu erhalten. Die Visualisierungsresultate werden dann zu spezialisierten Anzeigeknoten geschickt. Diese setzen die Szene zusammen und zeigen sie an. Der zweite Teil dieser Dissertation beschreibt Verfahren, um verteilt gerenderte Objekte, die teilweise transparent sind und sich auch überschneiden können, ohne visuelle Artefakte zu rendern. Dazu werden sogenannte „Per-Pixel Linked Lists“ eingesetzt.

Der dritte Teil behandelt, wie die so gewonnene Visualisierung auf einem großen

Display oder einem anderen (mobilen) Gerät präsentiert werden kann. Die Szenen können sehr groß und komplex sein, benötigen also viel Speicherplatz, den ein einzelnes Visualisierungssystem nicht immer zur Verfügung stellen kann. Große Displaywände, die mit solchen Datenmengen umgehen können, bestehen aus mehreren Monitoren oder Projektoren, die von mehreren Anzeigeknoten betrieben werden. Zudem gibt es meist spezielle Operatorrechner, mit denen die Visualisierung auf diesen Displaywänden gesteuert wird. Es werden Interaktionskonzepte vorgestellt, wie der Benutzer die Visualisierung auf einem großen Display nur unter Zuhilfenahme von mobilen Geräten kontrollieren kann und somit nicht den Umweg über einen Operatorrechner gehen muss. Dazu müssen die Visualisierungen auf dem mobilen Gerät auch angezeigt werden. Je nachdem, welche Daten gerendert bzw. welche Darstellung genutzt wird, können die Daten auf dem mobilen Gerät entweder direkt, in reduzierter Form oder mittels Streaming-Technologien mit der gleichen grafischen Qualität wie auf dem großen Display angezeigt werden.

In dieser Dissertation werden Methoden vorgestellt, wie heterogene Computerumgebungen, sei es ein einzelner PC oder ein ganzer Cluster, mit wenig Aufwand programmiert werden können; wie tiefenunabhängiges Transparenzrendering mit weniger Speichertransfers als bisher zur lokalen und verteilten Visualisierung genutzt werden kann; und wie mittels mobiler Geräte mit Displays interagiert werden kann, ohne dass spezielle Hardware nötig ist. Die Methoden zur Programmierung heterogener Systeme vereinfachen die Entwicklung und Wartung insbesondere im Hinblick auf die Unterstützung verschiedener Hardwareplattformen. Die vorgestellten Algorithmen zum Transparenzrendering liefern akkurate Darstellungen und benötigen weniger Grafikspeicher als bisherige Ansätze. Die Konzepte zur Interaktion mittels mobiler Geräte beschreiben, wie diese allgegenwärtigen Alltagsgegenstände mit jeder Art von Display benutzt werden können, ohne dass hierfür besondere Hardware benötigt wird. Als Abschluss werden diese einzelnen Themen zu einem Gesamtkonzept vereint. Die Abstraktionsschicht bildet das Framework der Programme zur Berechnung und Visualisierung. Die so gewonnenen Daten werden mittels des verteilten Renderingansatzes visualisiert und die mobilen Geräte zur Benutzerinteraktion eingesetzt.

CHAPTER 1

Introduction

With the performance of today's computer systems and ongoing trend of increasing performance, programmers and users are keen to exploit this performance gains. In the context of simulation and computation, these improvements are used to achieve higher frame rates, more detailed calculations, or increased input and output data sizes. Visualization systems can make use of the increased performance by increasing the display resolution, by rendering more realistic effects, or by being able to process the output of the computation faster and more detailed.

The central research questions of this thesis focus on computing and processing the ever increasing amount of data, visualizing the results without losing details and interacting with the visualization in an efficient and user friendly fashion. To be precise, this thesis will present methods to abstract the hardware and Application Programming Interfaces (APIs) for computational programs (see Chapter 3), distributed remote transparency rendering for presentation (see Chapter 4) and an extension for generalized rendering of structured and unstructured data (see Chapter 5), and show new approaches for the interaction with large display walls using mobile devices (see Chapter 6).

1.1 Motivation

The user wants to process more data, ideally in less time, and expects the new hardware to deliver adequate performance. The presented abstraction layer targets those expectations and aims to provide maintainable and extensible

functions for applications to make use of the performance deliverable by the hardware; now and for future releases. For the detailed output, rendering and visualization techniques are needed which can provide interactive framerates and allow the user to fully inspect the computation results. A concept which can provide this is transparency rendering. For displaying large amounts of data, sometimes complex displaying solutions like display walls are used. To make interactions easy and comfortable for the user, the thesis presents conceptual methods for using mobile devices for interacting with the visualization – on large display walls as well as in remote rendering scenarios.

Having the increasing computational power, users expect the ability to handle bigger data sets, faster computation times, and more accurate results. To satisfy these expectations, not only the hardware development needs to be continued but also the software development needs to advance. This holds for the computational part as well as for the visualization and interaction.

The computer systems and clusters used for computational tasks get bigger and more complex. One of the largest systems is the *Tianhe-2* [Dongarra, 2013] which has about 3 million cores, about 1.34 PB in main memory and a performance of about 33.68 petaflops/s.

Developing software for these systems is time-intensive and costly. When the underlying system architecture or APIs change, a certain effort is necessary to update the software and usually the process for optimization of the code starts again. The abstraction layer middleware, DIANA, presented in this thesis (cf. Chapter 3) aims to provide a layer between the hardware and the actual application. The abstraction layer acts as the connecting element for *algorithmic building blocks*, allowing an easy development of applications by composing readily available plugins or modules. Its most important design goal is maximum performance. Thus, it should not waste any time on communication, data handling, or other administrative tasks. On the other hand, the software needs to be maintainable and easy to understand. Thus, maintainability is the second design goal for the abstraction layer. The third design goal for the abstraction layer is extensibility. This enables programmers to create modules which can perform a specific task on a given hardware device, thus allowing the software to be adapted for future uses.

Instead of programming using the APIs itself, DIANA can be used as a foundation for computational programs. It is put between the (hardware vendor) APIs, and the application, and is used as an interface to or abstractor of the computation hardware and the data. Modules implement the API of a given hardware or product and are used by the actual application to do its work. The implementation of the modules is hidden from the application which can make

use of the provided functionality. It is even possible to use remote devices for distributed computing when the module implements network code. Thus, any application using the abstraction layer can make use of the available hardware, either locally or on a remote computer, when the employed module supports the given task.

Once the computation is finished, the data can be stored for later use, or be prepared for presentation and direct visualization in an in-situ scenario [Knoll et al., 2013]. When a distributed system was used for computation, the resulting data can be too large – either due to the data set resolution or the domain size – to be handled for visualization on a single node. Therefore, the distributed architecture also needs to be taken into account for the visualization system. It might be important for the understanding of the data that all information is visualized at once. The usage of transparency is an option to give the user a (proverbial) insight into the data. On one hand, this creates visual clutter, i.e. more data that might confuse the user, but on the other hand, can provide additional information, like showing multiple layers simultaneously. The application has to take care of the visual clutter, e.g. by providing functionality to remove layers or alter the color schemes.

The rendering process can either take place on the computation device(s) or on dedicated nodes. For large data sets that have been computed on a distributed system, a distributed visualization might also be beneficial because this avoids the transfer of the raw data. Instead, only the visualization result has to be transferred which in some cases can be much smaller and thus faster. Other reasons for a distributed visualization are a large resolution or a display system which consists of multiple display devices.

Rendering transparency in a distributed system is a challenging task. The main part of this thesis (cf. Chapter 4) focuses on rendering semi-transparent objects which overlap each other. The reason for this is that for the standard procedure, where color and depth buffers are used for remote visualization, it is not possible to compose overlapping semi-transparent objects as interleaving insertion is not possible. In a distributed computation system, the data set can be decomposed in object space. Note, that no clipping is used and instead, complete parts of the scene are distributed. An example for this is a simulation of a car with objects like body, frame, tires and suspension. In this case, the outer body completely encapsulates the frame and suspension. Other examples for this scenario are distributed mesh renderings, especially if the geometry is too big to store in the graphics memory of a single node, or comparative visualizations where multiple instances of an object are rendered with slight variations, e.g. at different time steps of a simulation.

When the rendering of the data contains very fine structures, a standard workstation setup with one or two monitors might not have enough resolution to show the whole visualization and simultaneously deliver the level of details needed to understand the data. Standard paradigms like panning and zooming [Cockburn et al., 2009] might not always be viable, e.g. when the whole scene or visualization is necessary, or additional data needs to be displayed together with the actual data set. Visual analytics applications [Kehrer and Hauser, 2013] often display one or more visualizations of the data and additionally present tables or graphs that provide extra information to the viewer. Especially when multiple users want to inspect a data set at the same time, display systems that can deliver a high resolution and provide enough space for multiple persons are necessary.

While driving these visualization and displaying systems is an interesting research topic by itself, parts of this thesis focus on user interaction with large displays (cf. Chapter 6). Usually, the user interaction for visualization is done directly via conventional input devices. The device is either connected to the rendering node directly or an application controls the input signals, and distributes them among the participating rendering nodes. It is cumbersome for large displays to walk around with the standard input devices like mouse or keyboard. For the research in this thesis, mobile devices like smartphones or tablets are used instead. They act as an input device and additionally, the running application can display context information either for interaction or as ancillary information on the data.

The proposed method aims at distributed visualization systems, e.g. for presentations or remote visualization scenarios, where the user can control the visualization with a mobile device. When giving a talk, the presenter wants to focus on the text and the audience, not on controlling the presented visualization. Using standard devices like smartphones or tablets, the user can intuitively interact with the visualization. Additionally, for collaboration scenarios, multiple users can use their devices and all work together with the data set. Here, priorities and permission have to be worked out but this topic is beyond the focus of this thesis.

1.2 Structure and Contribution

This thesis is structured as follows: Chapter 2 presents the current evolution of the hardware and the state of the art of the concepts, algorithms and software in the area of General Purpose Computation on Graphics Processing Unit (GPGPU), transparency rendering, visualization and user interaction.

Chapter 3 details the concept of a computation hardware abstraction layer and demonstrates an example application. With every new hardware or update, the new application programming interfaces for computation devices have to be updated in a program. Changing programming models might make it necessary to completely rebuild the application. The presented abstraction layer [Panagiotidis et al., 2011, 2012] aims to simplify and generalize the development for computational hardware by encapsulating the device/API relevant code. Pre-compiled plugins deliver the access to the new devices for the applications which do not even have to be recompiled. In general, the abstraction layer provides higher maintainability, flexibility, and extensibility than existing frameworks or applications without a significant loss in performance.

Chapter 4 presents extensions to the Per-Pixel Linked Lists (PPLLs) algorithm for rendering Order-Independent Transparency (OIT) in distributed environments. With the data sizes increasing further, it is harder to keep track of the details which might play an important role in the visual analysis [Krone et al., 2014b]. Possibilities are to use rendering techniques like transparency rendering for blending multiple layers or larger displays up to wall-sized gigapixel powerwalls. Therefore, algorithms which can handle the amount of data have to be used. The PPLLs rendering algorithm is one of them, allowing a generic visualization of scenes with semi-transparent objects [Kauker et al., 2013b] and to be used in distributed computation and visualization environments [Kauker et al., 2013a]. The concepts and implementation details for the extensions are described together with performance evaluations and an outlook for further optimizations to this approach.

Chapter 5 extends the concept of PPLLs into a generalized rendering approach using Per-Voxel Linked Lists (PVLLs). PVLLs convert a scene geometry, volumes, and implicit surfaces into a voxel structure. The approach renders the scene from the three main axes, thereby capturing all fragments within. Rendering a standard volume data set with the PVLLs algorithm, only the visible voxels are stored, removing completely transparent sections and thus saving memory on sparse volumes compared to traditional volume rendering. Having all fragments available additionally enables global effects rendering like reflection, refraction, and shadows.

Chapter 6 describes interaction concepts for mobile devices as input devices and second screens in visualization systems and for large displays. As said before, with increasing data sizes it is harder to keep a general view on the visualization on one hand, but also to focus and keep the contextual information visible for the user, especially when the data is projected on wall-sized display devices. Here, mobile devices can come to help by displaying detailed or contextual information, additional visualizations, or serving as a intuitive input

device, especially in a more advanced visualization environment than the typical workstation setup. Single-user and collaborative interaction concepts are presented in this chapter together with an outlook of how this topic might evolve in the future.

Chapter 7 summarizes the contributions of the thesis on a higher level. Furthermore, it gives an outlook to a system combining the presented techniques.

Non-referenced co-authored Papers

Memory Saving Discrete Fourier Transform on GPUs

Authors: Daniel Kauker, Harald Sanftmann, Steffen Frey, and Thomas Ertl
Conference: Proceedings of the IEEE International Conference on Computer and Information Technology (pages 1152–1157), [2010]

This paper shows an alternative method to compute the two-dimensional Discrete Fourier Transform. While current GPU Fourier transform libraries need a large buffer for storing intermediate results, the presented method can compute the same output with far less memory. This will function by exploiting the separability of the Fourier transform. Using this scheme, it is possible to transform multiple rows and columns independently. The presented approach can compete with the timings of the two-dimensional transform provided by the NVIDIA CUFFT library but consumes at the same time far less memory, thus enabling the transformation of much bigger data sets on the GPU.

Evaluation of Visualizations for Interface Analysis of SPH

Authors: Michael Krone, Markus Huber, Katrin Scharnowski, Manuel Hirschler, Daniel Kauker, Guido Reina, Ulrich Nieken, Daniel Weiskopf, and Thomas Ertl
Conference: EuroVis 2014 Short Papers (pages 109–113), [2014a]

This paper presents a GPU-accelerated visualization application that employs methods from computer graphics and visualization to analyze SPH simulations from the field of material science. To this end, the iso-surface that separates the stable phases in a fluid mixture is extracted via the same kernel function that was used by the simulation. The application enables the analysis of the separation process using interactive 3D renderings of the data and an additional line chart that shows the computed surface area over time. This also allows for validating the correctness of the simulation method, since the surface area can be compared to the power law that describes the change in area over time. Furthermore, the iso-surface that is based on the simulation kernel is compared with an established method to extract smooth high-quality SPH surfaces. The comparison focuses on demonstrating the applicability for data analysis in the context of material science, which is based on the resulting surface area and how well the two phases are separated with respect to the original particles. The evaluation was carried out together with experts in material science.

CHAPTER 2

Fundamentals

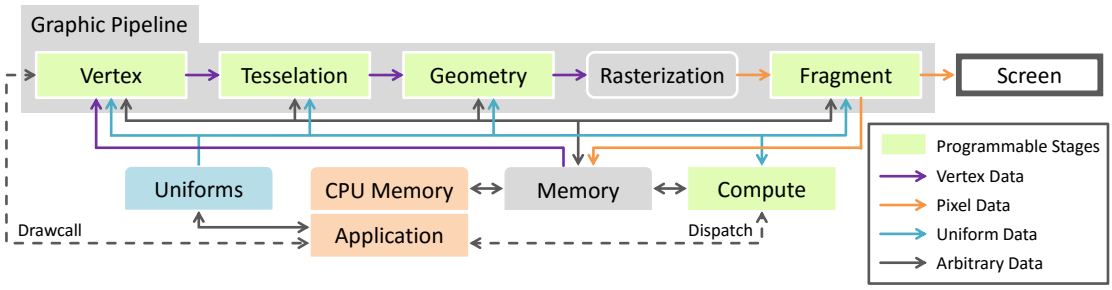
This chapter describes the state of the art at the time of writing. It defines the hardware situation, common terms, and standard approaches for visualization as a base line for the following chapters.

2.1 Personal Computer Hardware

In the last five to ten years, the trend even on low-end consumer products goes to the use of multi-processors. Here, the standard von-Neumann-architecture [Godfrey and Hendry, 1993] is still used, but parallelism is introduced. While Central Processing Units (CPUs) are very flexible in their execution model, allowing different threads to execute different operations using the Multiple Instruction Multiple Data (MIMD) pattern on a relatively low number of physical cores. Graphics Processing Units (GPUs) use the more strict model but do feature more than thousand parallel cores, allowing each thread to execute the same operation on different data in a lockstep manner, the so-called Single Instruction Multiple Data (SIMD) pattern. GPUs can easily outperform CPUs on tasks which fit this model, e.g. matrix multiplication with larger dimension sizes.

2.1.1 Graphics Processing Units

GPUs are a hardware component dedicated to rendering and graphical operations. A GPU in the context of this thesis consists of a chip housing a high number of parallel *Shaders* (2000 and more), and a dedicated graphics memory sub-system. The so-called *Unified Shader Model* (OpenGL) or *Shader Model 4.0*



▲ **Figure 2.1** — Simplified OpenGL pipeline with the programmable stages (green boxes). The arrows denote the data flow between the CPU and application side, the GPU memory, and the stages within the pipeline.

(DirectX) feature programmable shaders that can be used for the different stages of the rendering pipeline. While there were dedicated hardware sections with diverging capabilities for different parts of the rendering pipeline in earlier hardware, today's graphics chips use one common shader architecture for all parts. This flexible allocation model was introduced with the NVIDIA GeForce 8 series or ATI Radeon HD 2000 series GPUs in 2006.

The rendering pipeline consists of different stages (cf. Figure 2.1), most of which are at least partly programmable by the application. The application delivers the input data in form of so-called *primitives*, typically points, lines, triangles, or quads, to the *Vertex Shader*. Here, each vertex of the input data is transformed according to the model, view, and projection matrices. The *Tessellation Shader* is used for subdivision of the input model. In the *Geometry Shader*, each incoming primitive can be processed by a user-defined program. A typical use case for the geometry shader is the generation of proxy geometry, e.g. for *Raycasting* of implicit surfaces like spheres. Here, point vertices are uploaded to the GPU and the geometry shader generates a quad from the input vertex and a given radius. This saves bandwidth and memory but the computation on the GPU does cost performance. For the following stages, the input is the same as if four vertices were uploaded. After clipping and rasterization – the process of converting the geometry to fragments – , the *Fragment Shader* is called for each generated fragment. In this programmable shader, the fragment can be colored, textured, and lit, or removed, i.e., not blended into the final output. The final image is then directly sent to the screen or can be stored for later processing in the GPU memory.

Typically, the APIs used for programming GPUs are *OpenGL* or *DirectX*. Current versions are OpenGL 4.4 and DirectX 11. In the context of this thesis, the OpenGL notations will be used.

2.1.2 General Purpose Accelerators

General Purpose Computation on Graphics Processing Unit (GPGPU) is the concept of using GPUs not only for graphical operations but for general computations, too. This paradigm is enabled by the possibility of programming the vertex shader or fragment shader of the GPUs [Maughan and Wloka, 2001]. With the introduction of the unified shader model and the NVIDIA *CUDA* and *OpenCL* Application Programming Interfaces (APIs) using the GPUs for arbitrary computation tasks became even more popular.

The user can program *Compute Shaders*, which are actually the same hardware as the graphic pipeline shaders, but behave a little different in terms of input and output. While the graphics shaders have a defined input and output type, the compute shaders directly access the graphics memory via buffers of arbitrary content. The loss of the graphics related restriction allows the GPU to be used for other purposes of computation outside of the rendering pipeline. Additionally, compute shaders have access to shared memory. It is accessible by all concurrent threads. Barriers are used to avoid race conditions during execution.

The benefit of using GPUs for computation is that the highly parallel architecture can quickly execute SIMD-tasks. With hundreds or thousands of parallel shaders, the GPU can outperform a standard CPU on highly parallel tasks like matrix multiplication. The drawbacks of the GPU are the comparatively small memory of 12 GiB, whereas large CPU systems can access 128 GiB and more, and the comparatively small bandwidth to the host CPU. The single CPU is usually very good at a wide range of tasks whereas the GPU needs to strictly follow the SIMD-pattern to reach its peak performance.

Typically, a certain number of GPU threads are grouped into so-called *warps*. The performance of the GPU can only be maximized when all threads in a warp follow the exactly same instruction in a lockstep manner [NVIDIA Corp., 2014]. When branching occurs (like in `if`-cases or `for`-loops of different length), some threads might need to wait for other threads working on other instructions. Inactive threads, e.g. the threads in the other `if`-case, do the same operations but the writing operation is masked out, preventing them from writing variables. Thus, the whole execution takes as long as the union of all instructions executed by any thread combined.

Accelerated Processing Units (APUs) combine the different processors from CPUs and GPUs on one chip. This reduces the performance compared to discrete, high-end graphic processors but allows both sub-units to use the same but slow memory, effectively avoiding costly memory transfer operations. APUs are often found in small or embedded systems, e.g. mobile devices.

CUDA can only be used with GPU devices by NVIDIA, whereas OpenCL can

perform on any GPU, CPUs, APUs, and other specialized hardware. Currently, CUDA has a stronger user base on GPUs and is better supported than OpenCL due to the commercial interest of NVIDIA. The OpenGL compute shaders are closely related to OpenCL but are used from a graphics context from within OpenGL applications which makes the development and integration into graphical applications much easier.

2.2 Rendering and Visualization

Rendering is the process of converting raw data into a final high-quality visual representation, usually mimicking the real world to some extent. Visualization aims to present the raw input data in a way that can more easily be interpreted by the viewer than the data itself.

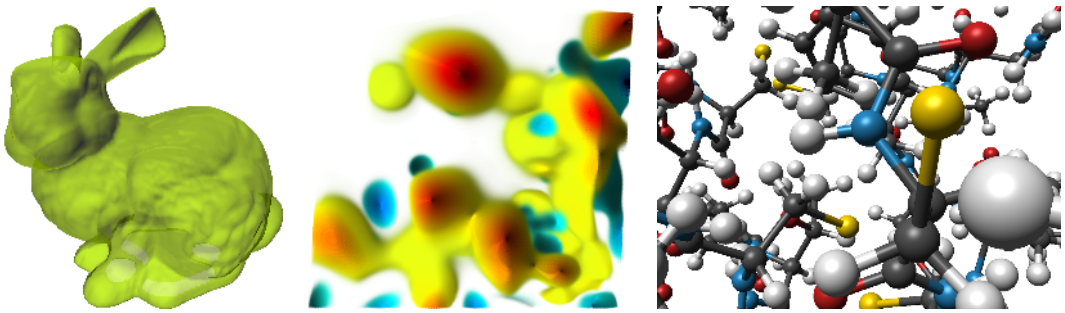
2.2.1 Rendering

In the area of graphics rendering, the raw data typically consists of primitives like points or triangles forming a mesh, volumetric data, or other representations. The result of the graphics rendering process is an image which is to be shown on a display device. Basically, it is a projection from the (higher dimensional) data/attribute space into the (two dimensional) screen space. On the screen, each pixel of the final image only consists of a color value. Blending operations and a depth buffer are used to overcome the limitation that each screen pixel can only hold one color value.

Blending can take place *front-to-back* or *back-to-front* and is subject to the implementation. It describes how the existing and new color values are combined and put into the framebuffer. Thus, in the context of this thesis, it is only referred to as *blending* unless otherwise noted. The distinction has no impact on the performance or capabilities of the presented methods. Usually, blending is not commutative, making the order of blending operations important for the composition of the final image.

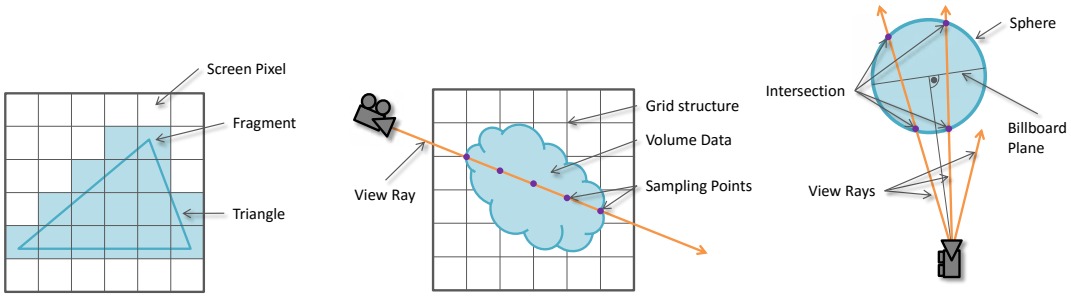
The depth buffer is used to store the image-space depth of fragments, typically the top-most one. This enables composition of unordered scenes with fully opaque objects. When a new fragment requires insertion on top of the currently top-most one, the current color in the frame buffer and depth value in the depth buffer are replaced, otherwise the new fragment is discarded and the top-most one stays.

For scenes with transparency enabled, the opaque geometry is usually rendered in a first pass with an enabled depth buffer. In the second pass, the semi-



(a) Mesh Rendering (b) Volume Rendering (c) Implicit Surface Rendering

▲ **Figure 2.2 —** The Stanford Bunny mesh (a), a volume from geostatistic calculations (b), and the ball-and-stick representation of a protein rendered using ray casting (c).



(a) Mesh Rendering (b) Volume Rendering (c) Implicit Surface

▲ **Figure 2.3 —** Schematics of mesh rendering using rasterization, volume rendering, and implicit surfaces.

transparent objects are rendered ordered by depth using the respective blending mode. They are applied only if their depth value lies over the opaque fragments already drawn before. After this, it is no longer possible to insert fragments behind the semi-transparent fragments, as they are already blended and the color information can no longer be separated.

Mesh Rendering

Mesh rendering (see Figures 2.3a and 2.2a) is one of the rendering methods used in this thesis to visualize data sets. A mesh is usually made of quads or triangles which are loaded onto or generated on the GPU. On the GPU, the vertices defining a primitive are passed through the vertex shader followed by the tessellation shader and the geometry shader. While the tessellation stage is used to subdivide the input, the geometry shader can apply modifications

the input data topology or spawn further primitives which are also passed on. Then, the primitives are rasterized, that is one fragment is generated for every pixel in the view port, and the fragments are then fed into the fragment shader. Here, the color and depth of the fragment can be modified. Finally, fragment tests, e.g. alpha or depth tests, are applied before it is passed on to the frame buffer where it is ready to be displayed.

Volume Rendering

Volume data is another data representation and widely used in scientific and medical visualizations. A scalar volume is (typically) some kind of three-dimensional array or grid where each cell, a *Voxel*, holds one or multiple data values. Other topologies, like unstructured grids or analytical/procedural formats, exist but are out of scope of this thesis. Volume data can be obtained from medical imaging devices like CT scanners, from geological measurements, mathematical computations, or simulations, only to name a few.

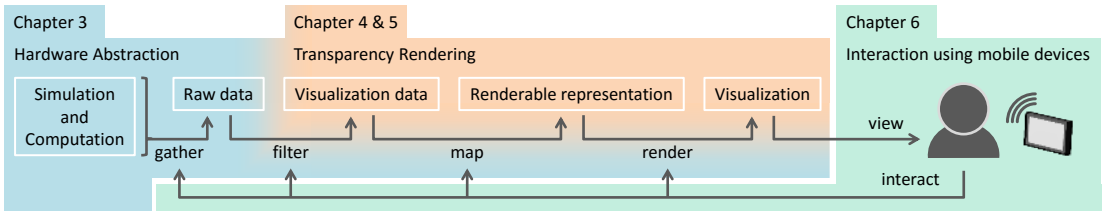
When rendering the volume [Stegmaier et al., 2005a], rays are traced from the camera through each pixel of the screen and through the volume in small steps (see Figures 2.3b and 2.2b). The actual traversal strategy depends on the algorithm used and its implementation. For each step, the data value for the current position is calculated, e.g. by interpolation from its neighbors. Next, a *Transfer Function* is applied, converting the value into a color, which is then blended onto the screen.

The transfer function maps the current data value into the color space for rendering. It is typically implemented using a lookup table which can be interpolated to find the exact output value for a given input. Other methods for coloring a volume are for example the *Gradient Mapping*. Here, the local gradient of the volume is used instead of the data value. The gradient value can also be used as a normal for lighting computations. The resulting color value is then blended onto the frame buffer or collected along the ray for later blending or processing.

Alternatively, the voxel value or local gradient of the volume can be used to generate an iso-surface [Krone et al., 2012]. The iso-surface can then be rendered as a mesh.

Raycasting of Implicit Surfaces

Implicit surfaces are a mathematical representation of an object, e.g. a plane or a sphere [Gumhold, 2003; Reina, 2008] using very few parameters. During rendering (see Figures 2.3c and 2.2c), the intersection of the viewing ray from



▲ **Figure 2.4 —** Abstract visualization pipeline in context of the chapters and the contribution of this thesis.

the camera with the object is calculated. The result is a pixel-accurate visualization of the given object. In contrast, meshed objects need to have a very fine resolution which costs graphics memory and performance to deliver the same visual quality. The backside of implicit surfaces is that a mathematical expression has to be known for the object. This is true and relatively easy to compute for simple objects like a sphere, a cylinder or a torus, but also for higher-order data [Üffinger et al., 2010; Sadlo et al., 2011] Triangles from a mesh could also be rendered as implicit surfaces but the sheer number (models might use hundreds of million triangles) would make this approach inefficient.

2.2.2 Visualization

Visualization is the process of converting raw data into meaningful visual representations. The raw data is often given in the form of lists, tables, or large and unordered chunks of numbers. The process of visualization uses colors, shapes, graph relations, geometric figures, highlighting, contextual additions, and other methods, to convert the data into an image. The goal of a visualization is that a viewer can easily interpret the image and understand the underlying data. A very popular visualization is a weather map where simulation outputs are turned into a colored map and are shown together with additional icons and temperature information. Rendering is one part of the visualization process, e.g. generating the underlying image. On this image, highlighting and color transformation is applied then.

The *Visualization Pipeline* (cf. Figure 2.4) defines the typical workflow of a (scientific) visualization application. First, the data has to be generated, for example by calculations, simulations, or by measurements. In the weather map example, the data is the result of a simulation. The raw data is then *filtered*, if necessary. For a coarse scale weather map of an entire country fewer data samples are necessary than for a large, high-scale view of a specific region. This filtered data is then *transformed* into some kind of ordered data structure, e.g.

into a regular two-dimensional grid or array. The result is *mapped* to graphical primitives. There exist a plethora of methods to map the data structures to graphical primitives. For the weather map example, the two-dimensional array can be used as-is for the rendering step. Regular grid data can be visualized using volume rendering, iso-surfaces can be extracted as a geometry mesh, and particles can be rendered as spheres, only to name a few. The graphical primitives are then *rendered* by the graphic system and displayed for the viewers to inspect and analyze. In case of the weather map, the array values are used as input for a transfer function which maps temperature to color, and the resulting color is displayed or printed.

The user can typically influence all of these steps, e.g. by selecting ranges of data, the mapping function, and the rendering method with all of their respective parameters. When the data generation or acquisition in form of a computation or simulation is also interactively controlled by the user, it is often referred to as *interactive steering* [Mulder et al., 1999; Wright et al., 2010].

In the context of this thesis, the presented abstraction layer (see Chapter 3) can be useful in the data generation, filtering and mapping. A method suitable for mapping and rendering the data, especially in a distributed environment, is presented in Chapter 4. Finally, the methods for user interaction with the focus on large displays and mobile devices is discussed in Chapter 6.

2.2.3 MegaMol

MegaMol [Grottel et al., 2015] was developed as visualization framework for large numbers of particles within the SFB716[1]. In this thesis, MegaMol was used for the remote and multi variant rendering (cf. Section 4.6) as well as for the user interaction using mobile devices (cf. Chapter 6).

It is a framework which provides windows, modules which do the actual rendering, modules for loading and processing data, and it also handles user input. The modules communicate using a user-defined interface which allows easy replacement of modules, e.g. exchanging a volume renderer for an iso-surface renderer connected to the same data loading module when the interface is compatible. The modular design makes it ideal for scientific development as the modules do not influence each other and can be developed and tested independently. It is also possible to use MegaMol with tiled display devices like the powerwall with the respective modules.

[1] SFB716 "Dynamic Simulation of Systems with Large Particle Numbers", http://www.sfb716.uni-stuttgart.de/ (accessed 29.06.2015)

2.3 Distributed and Remote Rendering

Distributed rendering [Brodlie et al., 2005, 2004] is a term for generating the final image on multiple independent nodes. In the context of this thesis, a *node* describes a computer participating in a cluster. All nodes are connected by a network infrastructure, e.g. Ethernet or Infiniband.

Rendering using a distributed network environment on multiple nodes and displaying it on an additional node is challenging [Makhinya, 2012]: The main challenge is the transport of the data to the displaying machine, regardless of it being parts of the final image or intermediate data, ideally at interactive frame rates, and compositing it without artifacts into a final visualization. In a distributed or remote rendering scenario, the *render application* produces the image or intermediate data and the *display application* shows the final image to the viewers. The continuous communication between the renderer and the display application demands an infrastructure which is capable of handling the required bandwidth for the data transport and has a low latency to provide a reasonable interactive user experience.

The main drawbacks of distributed or remote rendering is the network aspect. The network, typically also shared by other applications, has to deliver enough bandwidth and guarantee a low latency for an interactive and fluent user experience. Even if the network is exclusively reserved for the visualization system, normally it is not possible to scale the network architecture as easily as upgrading the rendering nodes, i.e. adding additional nodes.

Meligy [2008] gives a state of the art report about general parallel and distributed visualization. For distributed rendering, there are various systems and frameworks available, for example, DRONE [Repplinger et al., 2009] or Equalizer [Eilemann et al., 2009]. IceT [Moreland, 2011] is a scalable parallel rendering framework for applications to render remotely to a large display. There are also commercial and industry-grade frameworks that support remote visualization, e.g. RemoteViz[2] for the Open Inventor framework by FEI Visualization Sciences Group and the Visualization Platform[3] offered by Dragon HPC. Those systems are far more sophisticated in terms of distributing the work load or managing the nodes than the approach presented in this thesis. However, their transparency handling is still framebuffer-based, making the handling of distributed, intersecting objects unfeasible.

For a remote rendering scenario, rendering servers are run and controlled by a client application. A *server* is a software application which runs on a node in the

[2] http://www.vsg3d.com/open-inventor/remoteviz (accessed 29.06.2015)
[3] http://dragon-hpc.com/remote-visualisation.php (accessed 29.06.2015)

cluster. It interacts with other servers or the *client* by network communication. The client is also an application but is controlled by the user i.e. it has a (graphical) user interface and allows the user to tweak parameter settings. The rendering result can either be displayed on the rendering servers directly, can be transferred to the client application and displayed there, or can be transferred to dedicated display nodes.

Both remote and distributed rendering are closely related. The only difference is whether the image is transferred as a whole (remote rendering) or the display node have to perform post-rendering operations like compositing (distributed rendering). Hybrid systems can, for example, combine both approaches by using nodes for compositing the output of the render server(s) and forwarding the result to the displaying devices.

Distributed rendering can be used with a *power wall* visualization system where one or more devices are used to display the whole image space. As it might not be feasible for the user to interact with the large display, a client application can be used to steer the visualization on the large display.

In a *remote rendering* scenario, a device does not need the computational power for rendering the scene. Here, the user interacts with the device which sends the controlling information to the render server. The server then sends the visualization information back, either in a composited image or just (partial) raw data, depending on the application. Use-cases for this scenario might be a smart phone or tablet, used in a meeting or presentation, showing the visualization without the need of having a powerful laptop, workstation, or even rendering cluster at hand.

There are two different decomposition variants for distributed rendering: image-space (or screen-space) decomposition and object-space decomposition. The first variant partitions the screen and each participating render node works on one or more of these sections. As these sections do not overlap and only one node works on a portion of the screen, it needs to know the complete scene at this point. Therefore, the complete scene needs to be distributed among the participating render nodes. This might not always be feasible, e.g. if the scene is too large. For example, in a big in-situ simulation, the output sizes would be far too large to transmit for interactive rendering. This correlates to the sort-first approach for rendering as the partitioning of the data set takes places before the rendering.

The latter variant divides the scene in object-space. Here, one node renders its designated partition of the scene. This can lead to overlapping screen-space sections where more than one node contribute to one pixel on the screen. For this case, the compositing algorithm needs to be aware of this and has to be

able to blend multiple render results onto the same pixel in the correct order. Depending on where the sorting is done, either on the render nodes after the rendering process directly before sending or on the display nodes, this can be referred to as sort-middle or sort-last, respectively.

As an example, Section 4.6 details two different applications which make use of the distributed rendering using Per-Pixel Linked Lists (PPLLs).

2.4 Large Displays

In the context of this thesis, large displays are a setup of one or more display devices, e.g. monitors or projectors, which are driven by one or multiple display nodes. Typically, a single monitor or projector of today has a resolution of at least 1920×1080.

The specific display wall setup used for research in this thesis is the VISUS powerwall [Müller et al., 2013]. It is a stereo back-projection wall and consists of five projectors per eye. The projectors are positioned in portrait-mode next to each other. Each projector has a resolution of 4096×2400 and four input ports. The total resolution is 10800×4096 with an overlap of 300 pixels between the projectors. It is driven by one display node per projector, two graphics cards each, and each card is connected to two projector inputs.

No special interaction techniques are available to the presenter until now. When using the powerwall, a separate operator node in the back of the room is used to steer the visualization. The main reason is that the operator needs to see the power wall and has to control the presented material while the speaker talks facing the audience. Chapter 6 will detail approaches to give the speaker (or the audience) a direct influence on the visualization.

2.5 Mobile Devices

Today's mobile devices have a touch-enabled 4-inch to 10-inch display, typically featuring a resolution of 1280×720, 1920×1080 or 2560×1440, which corresponds to about 200 to 400 ppi. The devices typically have no hardware buttons except for power, volume control, or the camera shutter, if available. High-end devices feature a double or quad-core processor and two or more Gigabytes of main memory. They usually connect via WiFi to the local wireless LAN to communicate with other services.

Possible interactions on the touch screen are panning, zooming, and rotating using one or two fingers. Additionally, the touch screen can display standard

interaction elements like buttons, sliders, etc., enabling the user to also control parameters provided by of the application.

Although the used devices feature a 3D-enabled graphics chip which can run OpenGL ES, the graphical capabilities of the devices are limited and the graphics chips are very battery consuming. OpenGL ES, an OpenGL variant for embedded systems, provides a reduced API for graphics programming. This allows the chip vendors to develop more efficient hardware, as it only needs to support a reduced function set. As mobile devices do not have the room for large chips, nor the power to drive them, and, especially for the flat devices, cannot dissipate the heat, the mobile CPUs performance cannot be compared to today's workstation computers. These limitations and the fact that the data sets used can be in the size of Gigabytes, which is too much for transferring and storing them on the device, are the reasons to either use remote rendering or reduced data sets for the visualization on the devices itself.

CHAPTER 3

Computation Hardware Abstraction

As stated in the introduction, today's computer systems continuously offer more performance and the users exploit the performance by increasing the size and complexity of the problems. This chapter describes DIANA, a *D*istributed *I*n *A N*ode *A*bstraction layer for computational hardware. The goal of DIANA is to provide easy access to the ever-changing environment of the hardware in a node and in a homogeneous or heterogeneous cluster system, thus simplifying the development process. It is designed to be integrated into an application once and then supply independent plugins which deliver access to the hardware. Thus, the application does not need to be recompiled when a new API is deployed by the vendor or new hardware has to be used. Only the new plugin needs to be implemented, reducing the maintenance effort and development costs and time.

The plugins are a kind of building blocks for the programmer. They include access to a hardware component or deliver a compute function which can actually execute code on the hardware. The programmer uses DIANA to select the functions he needs and then execute them on the available devices, always having the possibility to exchange the hardware when the appropriate plugins are available.

The presented system also allows remote computations by using the same interface, effectively elevating standard single-node applications into cluster-ready applications. DIANA is therefore a scalable abstraction architecture. The performance cost of the abstraction is kept at a minimum, making it in no way inferior to the actual implementation without the middleware. As an example

case, a distributed visualization of a Finite-Time Lyapunov Exponent (FTLE) field using DIANA is presented.

Parts of this chapter are based on the publications "DIANA: A Device Abstraction Framework for Parallel Computations" [Panagiotidis et al., 2011] and "Distributed Computation and Large-Scale Visualization in Heterogeneous Compute Environments" [Panagiotidis et al., 2012].

3.1 Motivation

GPUs were always designed with a strong focus on performance due to the need for faster and more realistic graphics. Formerly, configurable fixed function pipelines were used on GPUs, but for some time now programmable compute units, so called *shaders*, are prevalent. Initially, shaders were used for graphics exclusively, for example for the transformation and lighting stage, texturing, or post processing. Due to hundreds of parallel processing cores on GPUs and the resulting high peak floating point operations per second, shaders were quickly adopted for general purpose computations. Since then, the concept of GPGPU has become a driving force in the development of new graphics hardware and a major aspect in high performance computing (HPC). Moreover, there are other specialized hardware platforms suitable for parallel computation, for example Intel's Xeon Phi.

Programming data-parallel GPUs is different to task-parallel CPUs in several ways. First, data residing in the main memory needs to be transferred to and from the GPU memory in order to access it, while the CPU has direct access. This non-uniform memory access within one node is similar to distributed memory processing with interconnected nodes. Second, GPUs were designed for graphics, that means for different memory access patterns, like texture lookups, where data locality is given. In contrast, CPUs normally access memory in a very random manner with a smaller performance penalty than GPUs. Finally, GPU cores are combined in so-called multi-processors that execute the same code, making branching costly. Nonetheless, using GPUs for some operations has proven to deliver speed-ups by several orders of magnitude and should not be easily disregarded. However, the kernels for these operations do need to be written and optimized by hand and for each hardware architecture again. Therefore, it is obvious to implement a system that takes care of the environment and only to exchange the parts that are in contact with the actual API or hardware.

In general, hardware vendors supply their own, distinct API for their GPUs or compute units. So, in order to access the various parallel computation hardware,

developers have to use different interfaces and libraries. This is complex and time-consuming for a variety of reasons, like detecting available hardware, using diverse interfaces, incompatibilities between hardware, and different tool-chains. Even when developers choose to specialize their applications by supporting only one API, they face problems like differing hardware capabilities (e.g., double-precision support), and changing APIs. Additionally, it is sensible to support different devices, because of changes in hardware (e.g., better performance, higher energy efficiency, lower costs) and suitability for computations [Göddeke et al., 2007; Brodtkorb et al., 2010]. Furthermore, software that is already in long-term production is not easily adapted to new hardware and the GPGPU paradigm. For example, tested and productive code needs to be changed to use GPU APIs, code (kernels) for GPUs needs to be written, and interfaces are needed to decide whether to call CPU or GPU code.

Nowadays, clusters are typically homogeneous regarding hardware as well as software. While this simplifies development, deployment, and debugging of parallel applications, it also dictates APIs and platforms (e.g., GPUs vs. CPUs, NVIDIA vs. AMD, CUDA vs. OpenCL). Furthermore, extending the infrastructure with heterogeneous hardware often necessitates widespread changes in software. On the one hand, using homogeneous infrastructures is the predominant approach in distributed memory processing and parallel computing, as seen on the various cluster systems, e.g. the Tianhe-2. On the other hand, this is hard to maintain in today's rapidly changing field of high performance computing because of steadily increasing performance and memory, and decreasing power consumption of CPUs and GPUs, making hardware upgrades and thus software adjustments mandatory.

3.2 Previous Work

An overview of node-level parallelization is given by Brodtkorb et al. [2010]. They conclude that applications cannot rely on a single architecture alone but need to be developed with general heterogeneous systems in mind. This means that applications need to support different compute units locally, as well as remotely, to maximize scalability.

CAPS HMPP [Dolbeau, 2007] is a commercial solution that provides compiler directives for calling functions on accelerator devices. It allows for automatic generation of CUDA and OpenCL code through their compiler backends as well as user-supplied code. While HMPP and DIANA are conceptually similar, DIANA is more intrusive because it requires explicitly allocating memory, transferring data, and executing commands on a device. On the one hand, this increases the development effort. On the other hand, this allows for better control and

fine-tuning, for example by including more detailed and relevant information (e.g., data locality, transfer rates, estimated duration) for the scheduling.

CUDASA [Strengert et al., 2008] provides extensions to CUDA that handle communication and scheduling in multi-GPU and GPU-cluster systems using MPI. By using new keywords and an additional compilation step, computations are scheduled automatically for execution on remote nodes. These poll a head node for work when they are idle which may lead to a bottleneck, for example when many nodes poll simultaneously. DIANA, in contrast, handles communication to local and remote devices using various APIs automatically, without imposing a specific scheduling. This allows applications to decide how to distribute and access data, especially in a decentralized way as demonstrated in the example application (see Section 3.4). Furthermore, arbitrary compute units are supported through plugins without changes to existing tool chains.

Domonkos and Jakab [2009] describe a programming model for GPGPU-based parallel computing. They support clusters using MPI, and CUDA and OpenMP on the local node for the actual computation. While DIANA also supports these, it additionally features a flexible plugin model, allowing for easy extensibility of hardware APIs and computation kernels.

rCUDA [Duato et al., 2010] is similar to CUDASA by providing a client-server system for CUDA that communicates using sockets. Currently, rCUDA 3.1 exposes the CUDA 4 API, except for graphics-related methods, through a wrapper library that is loaded on clients as drop-in replacement for the real CUDA library. It provides support for the runtime API and the CUBLAS library but not for other libraries like CUFFT. Though one can use multiple rCUDA servers, it is unclear how an application uses a specific GPU on a server. Data transfers are handled implicitly but kernel launches need to be configured by manually building the call stack. DIANA goes beyond this approach not only by hiding a specific API but by providing access to arbitrary APIs (e.g., CUDA and OpenCL kernels, CUBLAS) for use through well-defined interfaces. Furthermore, all remote devices are exposed to the clients, providing full control of their use.

Barak et al. [2010] present two approaches to distributed heterogeneous computing. First, VCL provides an implementation of the OpenCL specification that exposes OpenCL devices of remote nodes. Second, MGP provides an API and source directives for easy partitioning and distribution of tasks on different backends, for example VCL. DIANA also allows the transparent use of devices but is not restricted to a specific API, providing access to any compute hardware with the respective plugin.

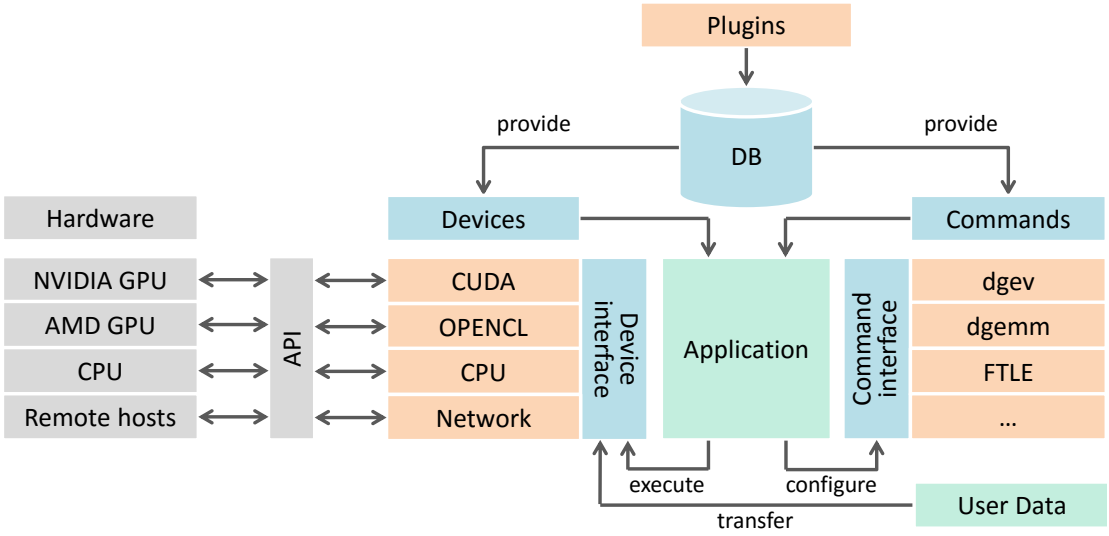
PaTraCo [Frey and Ertl, 2010] supports automatic scheduling and distribution of

problems to multiple nodes with different compute units. Computations need to be declared in passes and pass-blocks. Every node constructs a task-graph with computations as vertices, and transfer and execution duration as weights on the edges. The nodes then determine the critical path with regard to minimizing transfers in order to minimize the total duration for processing the whole graph. The edge weights can be set initially or guessed by PaTraCo and are updated every iteration, thus dynamically adapting to interconnects and workloads. However, the partitioning of computations into passes and pass-blocks can impose rather large changes to existing applications. Furthermore, PaTraCo only allows for scheduling of a fixed number of different compute architectures that need to be implemented at compile time. DIANA improves upon this by providing access to arbitrary devices through plugins loaded at runtime and a query mechanism for selection of available devices. Applications can use any dynamically loaded device in the same manner, thus requiring no source changes to support additional compute units. DIANA is also less intrusive since computations do not need to be expressed as passes and pass-blocks, and the scheduling is left for the application to decide.

Rossbach et al. [2011] present PTask, an abstraction layer for operating systems to manage GPUs as compute devices. Their integration with the operating system allows PTask a high performance and little overhead, as the objects can directly be managed by the kernel. They demonstrate PTask using a gestural interface and an encrypted file system as an example. In contrast to DIANA, PTask is designed as an integral operating system component. DIANA is designed on top of existing operating system and hardware driver APIs to be integrated on the application level.

Bourgoin et al. [2014] present SPOC (Stream Processing with OCaml), an OCaml based GPGPU programming framework. It allows low-level and high-level programming of the computation devices either by directly using CUDA or OpenCL, and by the integration of a OCaml language extension for GPGPU. While these systems are both similar to DIANA, their programming model is closer to the actual parallel programming style. DIANA hides the details from the programmer, offering only interfaces that have to be used instead.

The remote execution of DIANA resembles modern RPC systems like SmartGridRPC [Brady et al., 2010] in the sense of automatic serialization and transport of data, and invocation of procedures on remote nodes. While SmartGridRPC uses agents for service discovery, DIANA provides a local database – populated on startup by querying remote nodes – for access to remote compute units and operations. Contrary to the automatic load balancing of computations and reduction of communication in SmartGridRPC, DIANA requires developers to manually trigger computations and transfers. This allows for fine-grained



▲ **Figure 3.1 —** Overview of the DIANA architecture. Plugins (orange) provide commands and devices to DIANA (blue). An application (green) initiates the data transfers and selects the commands and devices needed for computation and DIANA provides the interfaces for execution on the hardware (gray).

scheduling optimization based on the knowledge of developers about their applications and problem domain. Internally, DIANA minimizes communication and the amount of transferred data where possible, for example by sending buffer contents only when necessary, or using special API calls on remote nodes to copy data between local compute units.

3.3 System Overview

In principle, the core component of DIANA is a database and it provides abstract interfaces for hardware and functions to be executed (see Figure 3.1). The following section describes the internals and how an application can make use of them.

3.3.1 Concepts

As the goal was to design a common high-level layer for multiple computation devices, three main components needed to be abstracted: the computational *Devices* itself, an interface for the abstract *Memory Objects*, and a description of the work to be done, the *Commands*. Basically, DIANA is a SQL database holding the plugins, and a set of management and interface functions to populate

the database, retrieve its content, and execute the kernel functions on the computation hardware.

Database Storage

The main part of DIANA is a SQL database backend for storing all information about devices, commands, and the internal management. During startup, the database is populated with the information provided by the plugins. Each plugin delivers the data it retrieved from the specific API (e.g. CUDA for the available devices) about the devices or the information about the commands to DIANA which then puts them into the database. Upon execution of a command, the database is queried for the issued command, e.g. the matrix multiplication. The query may contain specific device capabilities necessary for execution like the support of double precision floating point operations. Additionally, a matching device is queried from the database.

Devices

The device interface holds all information necessary to run a command on a device and all information about the capabilities of the device. Its implementation uses the standard interfaces provided by the operating system for the CPU device and the manufacturer specific APIs for all other hardware devices. The main ability of a device is to run a command. Even the device-specific commands for memory management like allocation, copy, put, get, set, and free are implemented as commands executed by the device.

Commands

A command mainly consists of a *run* method which is called when the command is executed. It handles the API specific execution of a function, e.g. calling the CUDA functions to run a given kernel. The command also handles the parameters necessary for the execution, e.g. the memory objects to compute on or the like.

Implementing a new command means to provide an abstract interface in form of a C++ class, a kernel function, and defining the capabilities of the command. The interface is what the programmer for the application is working with and the implementation is one instance of this interface. This ensures that multiple commands, e.g. one for OpenCL and one for CUDA, can be used with that one interface defining them.

To run a command on a device, the parameters for the command need to be set and then the command and the device are passed to DIANA which handles the

execution of the selected command on the given device. The application can implement an asynchronous command management, querying the command for completion (or failure), so it knows the current state of the command and whether it can continue with the computations.

Memory Objects

Another important component is the abstraction of the memory objects on each device. As each API can have its own memory management, the buffers also need to be abstracted. From the application point of view, the memory objects are managed using the device-specific memory commands.

3.3.2 Using DIANA

Upon start, DIANA loads all available plugins and queries the found devices for their capabilities like double-precision computation, available memory and the like. Usually, the first step for an application is to allocate memory on a computation device and initialize the memory. Therefore, a device is chosen and the database delivers a compatible command for the memory operations.

Then, the application selects a computation command from the database and requests a device which can execute this command. Here, data locality, computation capabilities or performance may play a role. These selectors may either be automatically derived by DIANA, e.g. when a given command needs a special capability, or by the user, e.g. selecting a device with a given feature.

The application then configures the command, sets all parameters and buffers, if required. Finally, DIANA executes the command on the selected device.

Multithreading is used to keep the impact on the actual application as low as possible. Callbacks or fences can be used to signal the completion of a computation request or memory operation.

3.3.3 Support for Remote Devices and Computations

DIANA supports abstraction of compute units on a node-level. For further scalability, DIANA automatically provides access to compute units on remote hosts. A small application (the DIANA server) runs on each remote host, listening for incoming compute requests from applications. Applications use DIANA exactly as before; remote devices are automatically discovered by the network plugin and are used in the same way as local devices. For the programmer, the remote devices are accessed like local devices with all the memory transfers handled by DIANA.

Application Side

The network plugin sends a discovery message to known servers when it is loaded and receives a list of all devices and commands that are available on each server. When registering remote devices and commands in the application-side DIANA database, the network plugin appends a unique identifier for each server to the server-side identifiers, thus making them unique application-wide.

Internally, remote devices and commands are proxies that implement the same interface as their local counterparts. The proxies serialize and forward all calls to the target server, handling all communication automatically. Thereby, only scalar values (e.g., integers, floats) and globally unique identifiers for devices, commands, and buffers are transmitted. Note that sending and receiving of buffer contents are explicit memory operations, triggered by the application. Since the applications' network plugin generates all required identifiers for further communication, there is no need to wait for an answer from the server, except when querying the state of a command. The transportation layer ensures the end-to-end communication and the integrity of the transmitted data.

The network plugin does not link nor require any information for the remotely used target hardware or APIs. If the plugin is not loaded, there will be no automatic communication to hosts running the DIANA server. In any case, other device plugins (e.g., CUDA, OpenCL) behave exactly as before. As a result, applications can use devices without explicit distinction and knowledge of their physical location.

An application uses the same calls for both local and remote devices for memory allocation and release, data transfer, command execution, and result retrieval. Furthermore, the constraints for using remote devices also apply to local devices, since those might operate asynchronously. Therefore, it should never be assumed that drivers or APIs ensure barriers or execution order. Instead, applications need to query if commands are finished or encountered an error. When accessing data, an application needs to wait for the transfer to finish.

There is no implicit or automatic scheduling of any operations in DIANA since partitioning of the problem or data management is highly application dependent. Only the network communication is handled automatically by the network plugin. This leaves all responsibility for balancing computational workloads among available devices to the application which can employ any desired scheduling strategy, e.g. work-stealing. Ultimately, when a network command is finished, the application releases it and the network plugin tells the server to free all corresponding remote resources.

DIANA Server

On startup, the server loads the DIANA plugins for access to local devices and available commands thereon. The server then waits for requests from the application-side network plugins to discover devices and commands, execute operations, or query command states. The discovery simply returns the local database of the devices and commands. When a command request is submitted, the server deserializes the parameters, constructs the corresponding local command, and queues it for execution. All submitted commands are non-blocking on the client side to maximize parallel operation. Applications only block when querying the command state.

Memory operations behave in the same way as computations: execution followed by query of the state for completion or error. The only exception is the transfer of data from a remote device, where the data is returned together with the notification of command completion. Subsequently, the network plugin on the application side copies the result into a local buffer. Commands and other resources are held until the application issues a release.

DIANA servers can also communicate with each other, without routing traffic through the node that executes the application. In such a case, the usual device-to-device transfer is realized by a copy between remote nodes. As each device has a unique id, the servers know where to send the data and if the destination is within their own domain or on another server. However, the application is in control and the DIANA servers only receive the copy command with the source and destination device identifiers.

Use Cases and Scenarios

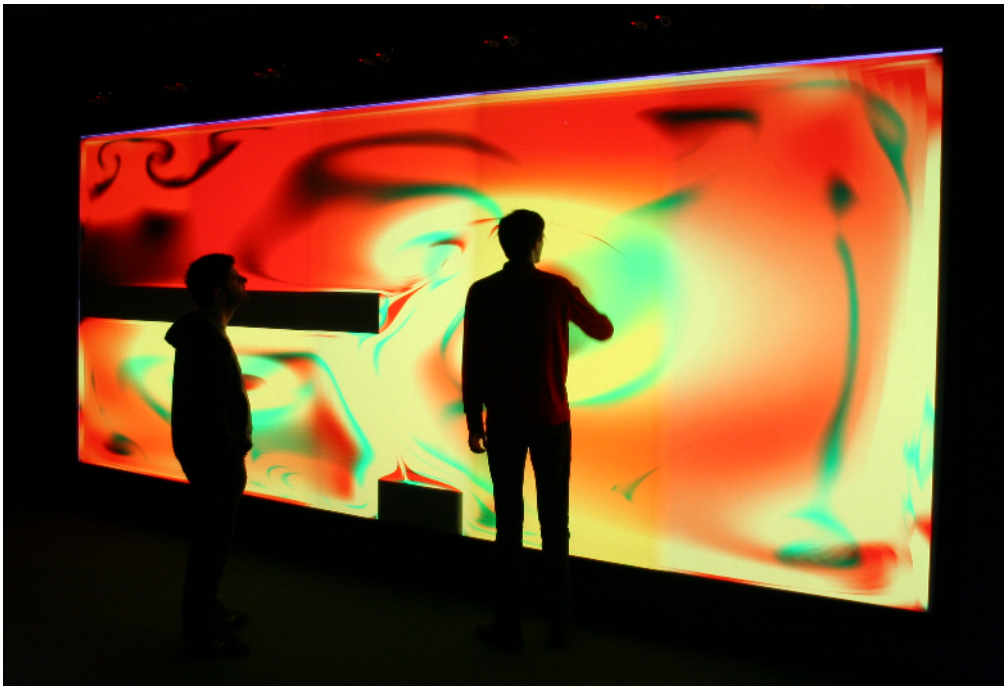
With DIANA it is possible to supply the same operation (e.g., FTLE computation) multiple times for the same and for different compute units. Applications can choose from the available operations in order to optimize performance or efficiency. For example, applications can operate on small data sets locally when equipped with less performant devices, or offload larger data sets to more powerful remote nodes.

Using this feature also enables seamless switching between operation implementations to evaluate different platforms and increase portability. For instance, developers can start using CUBLAS on NVIDIA hardware and then later choose to use clAmdBlas on OpenCL platforms without any changes to their application. Furthermore, one can provide specifically tuned variants of operations that are more performant for given problem sizes than standard versions in libraries. The user has to specify selectors or conditions for the command to be used for the current input data.

3.4 Large-Scale Finite-Time Lyapunov Exponent Field Visualization

The presented system can be utilized for computation and visualization. The following section explains an example of using DIANA for visualization of a FTLE field.

As an example of the capabilities of DIANA, it is extended for computation and visualization of the FTLE [Sadlo and Peikert, 2007] in a distributed heterogeneous environment. It demonstrates how DIANA can be used with multiple different nodes and compute units to generate a large-scale visualization of the computationally expensive algorithm.



▲ **Figure 3.2 —** Visualization of a FTLE field computed by distributed DIANA instances and presented on the VISUS power wall.

The system consists of two parts: a DIANA plugin and a display application. The display application triggers the computation, fetches the results from DIANA, and composes the resulting image which is shown on the attached display (see Figure 3.2). Since each pixel of the FTLE visualization requires the computation of a forward and a reverse trajectory in the vector field, it lends itself well for parallelization. Each trajectory is computed independently and is defined by its start point, start time, and duration of advection. The viewport of the display

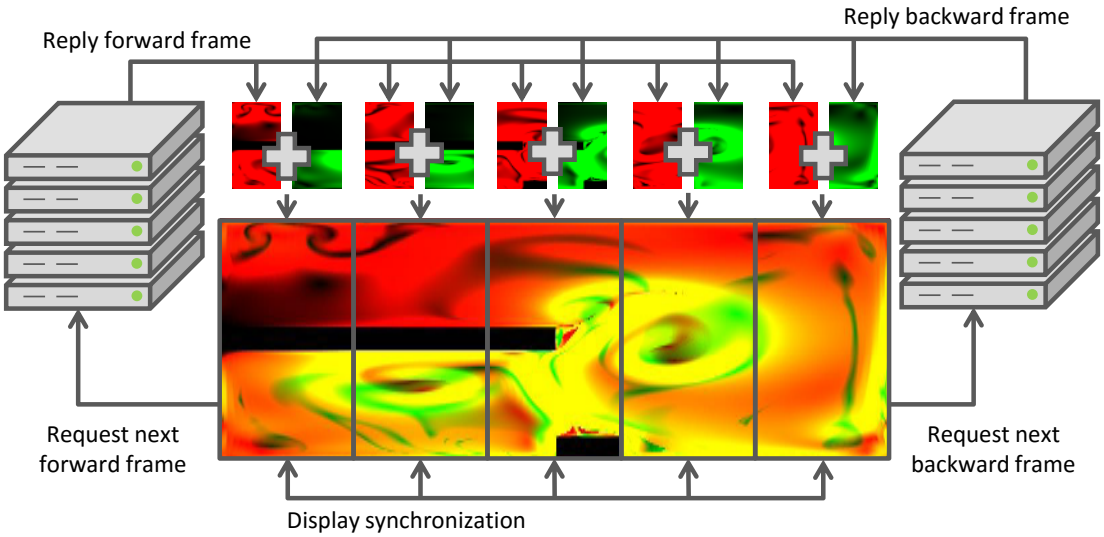
is partitioned into as many parts as there are compute nodes assigned to the application. Thus, the application builds the final image from parts of equal width and height computed by statically assigned compute nodes. As each pixel is exactly as expensive as every other pixel, no balancing issues occur in this scenario and, thus, no scheduling is necessary here.

The VISUS powerwall (see Section 2.4) together with a cluster of 64 nodes is used for visualization (see Figure 3.2). Each cluster node contains two NVIDIA GTX 480 GPUs, and the result is displayed on five 4096×2400 projectors resulting in a 40 mega-pixel display on a powerwall. Simultaneously, the same frame is computed at lower resolution on commodity hardware of workstations (NVIDIA GTX 465/480 and AMD RADEON HD 5870) for use on a smaller auxiliary display, in this case the operator node of the powerwall. All displays are synchronized with each other, resulting in a consistent view on the powerwall and the operator node.

DIANA Plugins for FTLE Computation

The plugin is the second component of the system. It computes two FTLE fields, one resulting from forward and one from reverse particle advection. The retrieval of time-dependent vector field data is treated as a black box, that means, DIANA assumes no details regarding its internals or runtime characteristics. Consequently, one can switch between simulations, if they provide correctly formatted data, without changing anything in DIANA or the application. For the tests, pre-calculated, time-dependent CFD data with a resolution of 101×101 grid was used as input.

Commands on devices consist of two parts in DIANA: the abstract interface and the device-specific implementation. The abstract interface for computing the FTLE requires some device buffers as well as parameters describing the compute domain. The *computeFTLE* command for this example case provides an implementation of this interface for CUDA and for OpenCL, each residing in a separate DIANA plugin (for development reasons only). Note that these mostly call the respective API for execution on local hardware. The actual transfer of data and the remote invocation is handled by the network plugin (see Section 3.3.3). The implementation requests the necessary vector field data from the simulation black box and passes it to the kernels for both forward and reverse integration. The results are then stored in separate buffers which are sent back to the display nodes.



▲ **Figure 3.3 —** Each display application requests a forward and a backward frame and merges them to the final image. The display nodes stay in sync to deliver a consistent image.

Display Application

The FTLE evaluation domain is mapped to a display that may contain multiple distributed display nodes, each having its own local viewport. Thus, the display application needs a list of other nodes to synchronize the display with, the number n of display nodes it shares the global viewport with, and the index i of its local viewport relative to the others. In this scenario, the synchronization list contains the five powerwall nodes and the operator node. For the display nodes of the cluster n is set to 5 and i is set to 1..5, so every display node shows one fifth of the final image (see Figure 3.2). The auxiliary display uses $n = i = 1$, thus the whole FTLE domain is shown in one window. The list of synchronized nodes needs to be the same for all application nodes, to provide a synchronized view. Additionally, the display application receives a static assignment of DIANA servers to compute the FTLE field.

As described in Section 3.3.3, the application-side network plugin discovers remote devices and commands on startup. Then the application queries the local database for implementations of the *computeFTLE* interface and subsequently for devices that can execute it. After starting the computation and synchronization threads, it sets up an OpenGL context and enters the rendering loop.

The computation thread requests result frame f from every assigned remote device using f as time step parameter for the FTLE command (see Figure 3.3). Thereby, every device computes a part of the image. The resulting buffers

containing the forward and reverse FTLE are retrieved after each device of this display node finished its computation in parallel. Those parts are then merged to two coherent buffers and cached for display. Finally, the application broadcasts to all display nodes that it is ready to display frame f and continues by requesting frame $f + 1$.

The synchronization thread waits for messages about completed frames from other display nodes. Upon receiving one, it increments the node counter for that frame. When all nodes have signaled the availability of the frame, it notifies the render loop to use that frame and all previous frame parts are discarded. Note that the chronological order is maintained as the frames are requested in ascending order and frames are discarded only when a newer frame is ready on all display nodes. Thus, every frame is computed sequentially, transferred to the display node, and displayed in the correct order.

The render loop simply maps the scalar field output of the FTLE computation to a texture that is rendered on a full screen quad by using the forward FTLE as the red channel and reverse FTLE as green channel; the blue channel is set to 0.0 and alpha to 1.0 (see Figure 3.2).

Scalability and Universality

The cost of the FTLE computation depends on the output resolution and integration length. The amount of data a display node receives is constant and only depends on its total display resolution. Thus, the time needed to generate a single frame decreases if more nodes are added, since every node needs to compute smaller images. Note that the prototype is not restricted to nodes of the cluster to create an image on the powerwall. One can easily use nodes outside the cluster as long as DIANA is installed, a suitable network connection is available, and they support CUDA or OpenCL when the plugins for those APIs are provided. However, as the views are synchronized, the slowest node sets the pace as each node only can display the new frame when it is available to all other nodes.

The setup described here can be applied to accomplish any distributed computation and visualization scenario. For development, one can use local commodity hardware in workstations. Even when displaying the result on a single monitor, one can simply use multiple windows to verify if the tiles fit together. The whole system can then be easily deployed on multiple machines or a cluster.

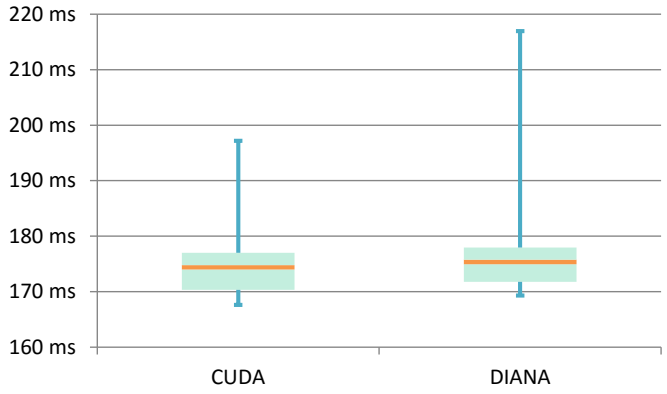
3.5 Summary

All benefits of the presented abstraction layer are achieved by trading-in performance. Every layer between the calling software and the executing hardware does cost performance. However, what matters when using the software is not only the performance itself but also the maintainability and flexibility. The system has to be well-balanced with respect to these goals.

3.5.1 Results

DIANA provides an interface to different device APIs, Software Development Kits (SDKs), and libraries. Since there are some layers of indirection (e.g., database lookups, virtual function calls) calling an API via DIANA should always be slower than calling that APIs directly. Therefore, a major goal was to minimize the time spent in DIANA while still having high extensibility and flexibility. In order to measure this overhead, we constructed the following test: initialize CUDA, set up the device, allocate memory on the device for a vector, transfer it from main to device memory, modify it on the device, transfer it back to main memory, free the memory on the device, and shutdown the device and CUDA. What is measured is the time it takes the CPU to call CUDA and DIANA respectively, since the intention was not to measure the performance of the computation or the hardware. This test is repeated 10,000 times with a vector containing 10,000,000 floats, resulting in 40 MiB to transfer. The test system was an Intel i7 920 with 12 GiB main memory and a NVIDIA GTX 465 with 1 GiB device memory.

In the initiation phase, only a context for CUDA is created, while DIANA loads a plugin and initializes it, thereby registering one CUDA device (that creates the context) and one command. The setup phase does not apply to CUDA, since the created context is already active. In contrast, DIANA needs to query the database for the CUDA device; its context will be activated inside of the command that modifies the vector. Memory operations (allocation, put, get, free) directly map to the respective CUDA functions and to the buffer methods of DIANA. Computation is split into two distinct parts, lookup and execution. In the lookup phase, an application determines which kernel to execute depending on available hardware. While a real application supporting multiple devices cannot skip this, no specific checks of hardware capabilities are assumed for the test. Therefore a CUDA kernel file is loaded directly and the function pointer for the kernel is retrieved. The execution phase creates the call stack and launches the kernel. These two phases are different in DIANA. In the lookup phase, the database is queried for the command and it is instantiated. In the execution phase, the application sets the parameters of the command and executes it



◀ **Figure 3.4 —** From top to bottom: maximum, third quartile, median, first quartile, minimum values of $\sum^*$.

	Init	Setup	Alloc	Put	Lookup	Exec	Get	Free	Term
DIANA	96.8	0.1	0.2	70.8	0.4	26.4	71.5	5.5	75.3
CUDA	89.1	0.0	0.1	70.7	25.8	0.0	71.5	5.5	71.7
Δ	7.7	0.1	0.1	0.1	-25.4	26.4	0.0	0.0	3.6

▲ **Table 3.1 —** Median time in milliseconds the CPU needed to call CUDA and DIANA. Note that the execution time is not measured on the GPU.

	$\sum$	$\sum^*$
DIANA	345.0	174.8
CUDA	331.8	173.6
Δ	13.2	1.2

◀ **Table 3.2 —** $\sum^*$ is the sum without Init, Setup, and Terminate from Table 3.1. Note that the execution time is not measured on the GPU.

directly. The command then executes all steps of CUDAs lookup and execution phase as described above. The termination phase releases all CUDA resources while DIANA unloads the plugin (thereby releasing devices and commands) and shuts down the database.

Table 3.1 shows the median times (rounded to the first decimal) of all phases for CUDA and DIANA, and their difference. Figure 3.4 shows the distribution of values for $\sum^*$ of Table 3.2 (total time without the phases Init, Setup, and Term). Here, $\sum^*$ equals the orange lines inside the boxes. Table 3.1 shows the median times (rounded to the first decimal) of all phases for CUDA and DIANA, and their difference.

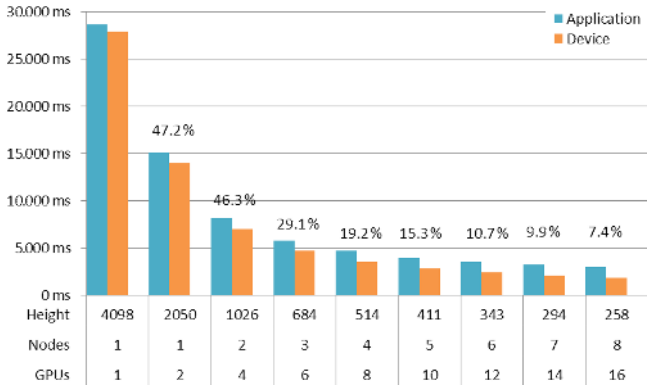
The difference for the computation can be accounted to two factors. First, the lookup used for CUDA is very trivial and even though DIANA's command lookup is fast due to the in-memory database, there is still some time needed.

Cluster Nodes	GPUs	Height (pixel)	Time GPU (ms)	Time App (ms)	Speedup (%)
1	1	4098	27940.1	28660.9	-
1	2	2050	13993.9	15145.6	47.2
2	4	1026	7021.0	8129.3	46.3
3	6	684	4677.6	5765.7	29.1
4	8	514	3533.1	4659.6	19.2
5	10	411	2826.1	3949.0	15.3
6	12	343	2361.7	3527.0	10.7
7	14	294	2030.7	3178.8	9.9
8	16	258	1791.5	2944.7	7.4
9	18	229	1595.3	2753.5	6.5
10	20	206	1438.0	2580.4	6.3
11	22	188	1302.5	2490.2	3.4
12	24	172	1188.7	2391.9	3.9
13	26	159	1101.0	2307.9	3.5
14	28	148	1030.0	2237.0	3.0

▲ **Table 3.3 —** Average times of the FTLE computation duration in milliseconds on the cluster nodes (GTX 480 using CUDA) for different image heights depending on the number of GPUs used. Additionally it is shown how long the display application takes from issuing all commands to receiving that all have finished and the resulting speedup when using more GPUs. The image is always 2400 pixels wide.

Second, the command parameters are stored in a hash-map that is filled by the caller. This map is then read inside the command and used to build the call stack for CUDA. Therefore, DIANA is overall a little slower in both phases than CUDA. Overall, the small overhead of using DIANA is acceptable since it provides many benefits.

As a benchmark, the average computation duration and the total time in the display application for collecting all parts of a FTLE image on the display nodes of the powerwall was measured. Thereby, the number of cluster nodes a display node uses for computing its part of the whole display is increased for ten successive frames. When using n GPUs, each GPU needs to compute an image 2400 pixels wide and $(4096/n) + 2$ pixels high. The output resolutions tested are the same in both cases and correspond to the resolution of the partial images of the powerwall display nodes depending on the amount of compute nodes used (up to twelve). Note that using an additional cluster node adds two GPUs



◀ **Figure 3.5 —** Performance and speedup (percentage over the bars) when using multiple GPUs for the computation.

to the system. Two additional pixel rows are needed to avoid artifacts between the partial images because the FTLE involves gradient computation.

Table 3.3 and Figure 3.5 show the average computation times depending on the number of GPUs (respectively cluster nodes), the total time the display application requires from starting the commands on all nodes to receiving the acknowledgment that they have finished, and the speedup when increasing the number of used GPUs. As can be seen, the speedup decays very fast, dropping below 20% when using four instead of three nodes (respectively 8 instead of 6 GPUs), below 10% using more than six nodes (12 GPUs), and drops to less than 4% when using more than ten nodes (20 GPUs). This correlates to the change in the output resolution when adding more GPUs. For example, each GPU needs to compute 170 lines less when using four instead of three nodes (514 lines instead of 684), and only 49 lines less when using seven instead of six nodes (294 lines instead of 343). For the latter, it still results in a difference of about 331 ms for the computation but suffices only for about 135 ms in the display application. Therefore, even though the computation is getting faster with more nodes and GPUs, the overall system performance is not increasing accordingly due to latency and overhead in communication. The difference between computation time and the time spent in the display application is varying throughout the different resolutions between $1 - 2$ seconds and can be accounted to overhead in DIANA and network communication.

3.5.2 Outlook and Conclusion

As the evaluation has shown, DIANA meets the goals without losing significant amounts of performance and still provides flexibility and maintainability. The system also scales within the tested cluster systems. What cannot be solved by the presented system is the fact that each hardware needs optimized kernel codes to achieve its designed maximum performance. This topic is subject to

ongoing research [Magni et al., 2013; Grauer-Gray et al., 2012]. Integrating automatic kernel optimization into DIANA would again reduce the amount of work, and thus costs, when upgrading to newer accelerator hardware.

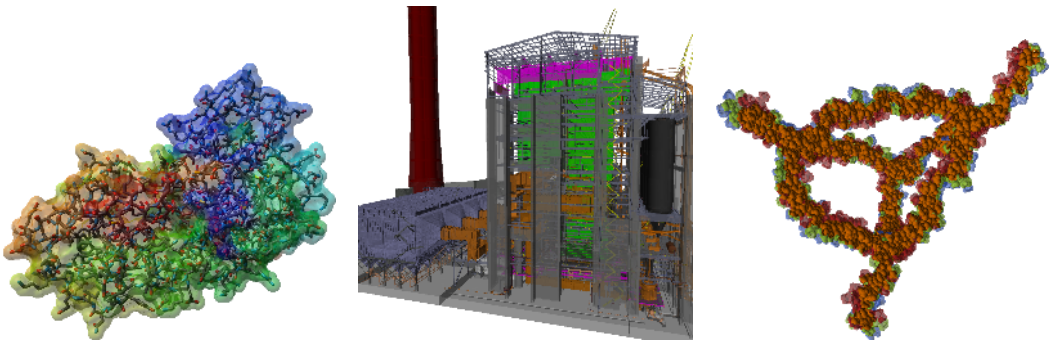
DIANA provides common interfaces and automatisms to simplify the development for different hardware and its usage. With DIANA, the continuous development and support for upcoming accelerator hardware is reduced to the implementation of a device plugin and a command plugin, completely removing any update work on the main application.

With OpenCL, CUDA, high performance cards from NVIDIA, AMD and Intels Xeon Phi currently on the market, there are three competitors for high performance computation. Having the option to switch to another hardware platform with minimal implementation cost is always an advantage over specialized solutions which cannot be upgraded easily. This is especially the case, when the performance drawback is as small as in the case of DIANA presented here.

The recent trend in hardware development is going to more specialized chips which are combined within one die, let it be APUs on standard workstations, or fast performance chips with an added low-powered companion core in the mobile processor section. This leads to a broader spectrum in processors available for a computer configuration and requires yet another configuration or implementation of the respective API. Using an abstraction layer, all these details can be hidden and the performance of the underlying hardware can be exploited with low effort.

CHAPTER 4

Transparency Rendering



(a) 1VIS protein rendering with transparency effects

(b) UNC power plant mesh rendering

(c) Visualization of comparative multi-variant renderings

▲ **Figure 4.1 —** PPLLs OIT renderings of a molecule surface combined with inner structure representation (a), the UNC power plant (b), and a comparative visualization of three different states of a hydrogel simulation (c).

Transparency is an important visual effect but requires additional effort when rendering, typically sorting geometry or multiple render passes. It might also require more memory that could be filled sparsely and fragmented, depending on the rendering technique. To achieve high frame rates, approximations and heuristics can be used but thus, the scene is not rendered exactly. The methods presented in this chapter aim to accurately render the complete scene but without any heuristics or simplifications.

Parts of the techniques and algorithms presented in this chapter have previously

been published in "Rendering Molecular Surfaces using Order-Independent Transparency" [Kauker et al., 2013b] and "Evaluation of per-pixel linked lists for distributed rendering and comparative analysis" [Kauker et al., 2013a].

4.1 Motivation

Using transparency in rendering and visualization can make the image more realistic, e.g. when rendering glass to look through. It can also be used to add additional information to a visualization, e.g. make multiple layers visible at once in volume data or visualize internal structures behind a semi-transparent surface mesh. Transparency allows the viewer to look behind layers in an image, thus allowing him to gain a better understanding of the scene or data. The best perception of transparent objects can be achieved in interactive scenes where the user can move the camera around a given object. Here, the different depth layers are not only shown as a blended color with the other layers, but the movement can help the viewer to get a better understanding of the scene.

Rendering more content can also increase the visual clutter in an image that might make it difficult to spot important details or even might overcharge the viewer. Care has to be taken to design the application in a way that it is possible to tune the parameters so that the viewer can still see the details, e.g. by altering the transparency values, color schemes, or by tweaking the visible layers.

In the context of this thesis, the main focus for the transparency rendering is when it is used for multiple objects which overlap each other, e.g. objects simulated by a distributed system. Example use cases are rendering of molecular surfaces (cf. Section 4.6.1), comparative visualizations (cf. Section 4.6.2), e.g. of multiple instances of a given data set, each representing a slight variation, or the rendering of unaltered object-space decomposed models (cf. Section 4.6.3), e.g. meshes which were segmented in logical units.

The algorithms used here are built upon the PPLLs concept introduced by Yang et al. [2010]. Figure 4.1 shows some example images rendered with PPLLs. There exist a plethora of other algorithms for transparency rendering, but the PPLLs approach is the most suitable and flexible as explained in the following Section 4.2.

4.2 Previous Work

For rendering scenes with semi-transparent objects there exist a plethora of algorithms. Basically, they can be divided into so called *sort-first* and *sort-middle/sort-last* methods.

For sort-first methods, the primitives with semi-transparent sections are sorted by depth first and then the rasterizer generates the fragments which are blended in order. An example algorithm in this category is the Painter's algorithm [Foley et al., 1990]. The drawback of these methods is that it is necessary to sort the geometry before rendering. This step typically has to be done for each frame as the viewpoint might change or the transparent objects move around in an animated scene.

In contrast, sort-last algorithms render the primitives in arbitrary order and sort the fragments before compositing. This category is also referred to as Order-Independent Transparency (OIT) rendering as the depth ordering is not important here and done implicitly by the algorithm. Algorithms like Depth Peeling (DP) [Bavoil and Myers, 2008], and especially the PPLLs algorithm which is discussed in this chapter, are representatives of this category.

The following sections will detail different standard transparency rendering algorithms. For a more complete overview on the topic, Maule et al. [2011] gave a survey of current raster based transparency techniques.

4.2.1 Painter's Algorithm

Rendering semi-transparent scenes is challenging as it requires the fragments to be sorted before blending them into the final image. One way of doing this is the Painter's algorithm [Foley et al., 1990] which orders polygons by depth before drawing. This implicitly guarantees a correct rendering of the scene (when no cycles are in the scene).

The Painter's algorithm usually separates the opaque and non-opaque primitives. The opaque primitives can be rendered with the depth-buffer enabled, thus guaranteeing a artifact-free opaque image. Only the semi-transparent primitives have to be sorted by depth and are then blended into the scene. Using the depth-buffer ensures that the semi-transparent fragments are only visible in front of the opaque scene. Thus, the Painter's algorithm produces one final image containing all fragments blended into it.

This makes it impossible to insert fragments between two already blended fragments which is important for distributed rendering, especially, when the objects are interleaving each other.

Recent transparency rendering algorithms using the Painter's algorithm are published for example by Zhang and Pajarola [2007], and Chen et al. [2012]. Zhang et al. do single-pass deferred blending using point-based rendering, also achieving transparency effects. Chen et al. use view point dependent presorted meshes and render using the painter's algorithm.

4.2.2 Depth Peeling

The Depth Peeling (DP) algorithm by Everitt [2001] renders the scene multiple times and in each pass the front layer not yet extracted is "peeled" off the object. Therefore, it requires n render passes to fully capture an object with n depth layers. A faster version is the Dual Depth Peeling (DDP) [Bavoil and Myers, 2008] which can peel the front and back layer at a time. Fragments that have been captured in earlier render passes are rejected.

Regarding remote or distributed rendering, this algorithm would require the transfer of $2 \times n$ full sized textures containing the color and depth values for each layer. This is not feasible, as the size of the network transfer is crucial for this scenario.

There are several optimizations of the basic concept, for example peeling multiple layers per pass [Bavoil and Myers, 2008; Liu et al., 2009], offering z-fighting awareness [Vasilakis and Fudos, 2011], or using only constant memory [Bavoil and Enderton, 2011], but they still consume too much traffic for distributed transparency rendering.

4.2.3 k-buffer

The *k-buffer* [Bavoil et al., 2007] concept is a generalization of the z-buffer. Instead of only storing the depth value for a given pixel location, k different values can be stored for each pixel. This allows for storing multiple fragments, thus enabling various effects like OIT, depth-of-field, and volume rendering. Recently, Zhang [2014] presented a memory-hazard-aware variant of the k-buffer avoiding read-write-conflicts.

The stencil routed A-Buffer by Myers and Bavoil [2007] uses multisample textures to store multiple fragments for one pixel location in one texel. Stencil-testing on sample-level allows the shader to write only one sub-element of the texel at a time.

However, the drawback of the k-buffer is that it allocates the full memory for k depth layers. The depth complexity of most scenes is distributed very inhomogeneously, wasting memory for elements which might never be used. In the context of distributed transparency rendering, this drastically increases the amount of transferred data although there might be no information. Additionally, as the k-variable is fixed for a given render pass, it has to be adapted to scenes with higher (or lower) depth complexity.

4.2.4 Per-Pixel Linked Lists

A algorithm which overcomes the drawbacks of the transparency algorithms explained beforehand is the Per-Pixel Linked Lists (PPLLs) approach. This algorithm is the basis for the transparency rendering algorithms presented in this thesis.

Yang et al. [2010] presented this rendering algorithm as an alternative to texture-based OIT rendering methods to overcome several limitations of aforementioned OIT transparency approaches and implemented this using DirectX. Knowles et al. [2012] call this concept *layered fragment buffer* and discuss different memory layouts. The idea is based on the *A-buffer*, introduced by Carpenter [1984] as part of the REYES CPU renderer. It captures all fragments and their attributes of a scene instead of blending them onto the final image consecutively. Wittenbrink [2001] presented the *R-buffer*, a hardware architecture design implementing the A-buffer technique.

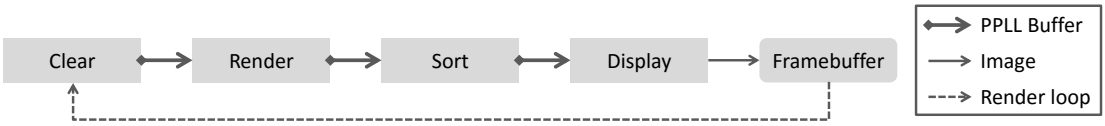
Technically, PPLLs became feasible when GPUs were able to read and write to arbitrary global memory positions and not only (limited) texture space. In 2010, DirectX 11.0 and the OpenGL 4 extensions NV_shader_buffer_store [Brown, 2012] and NV_shader_buffer_load [Brown et al., 2010] made those operations available to GPU shader programs.

PPLLs are created by appending each generated fragment to a list local to the according output position so that each pixel does have its own linked list of fragments. The resulting buffer is tightly packed with per-pixel list elements, but each list is scattered and not stored contiguously in memory.

After rendering the fragments and storing them in the list, they might not be correctly ordered by depth. Thus, they have to be sorted prior to displaying, either forward or backward in terms of depth ordering. Subsequently, the fragments in the list can be blended by simply traversing the list and applying the appropriate blending operation [Porter and Duff, 1984].

Salvi et al. [2011] use heuristics to reduce the number of captured fragments by merging fragments which are very close to each other or do not contribute much to the scene, e.g. due to a high transparency value. However, this still eliminates the possibility of inserting further data between the merged fragments on the display node in the merging step. Therefore, this optimization does not apply to this work.

Pixel Shader Ordering provided by Intel is an extension to DirectX for the Haswell integrated graphics system. This makes it possible to order memory access by the GPU based on the pixel location which can also be used for OIT effects [Fife et al., 2013].



▲ **Figure 4.2 —** The stages of the PPLLs algorithm pipeline.

The PPLLs approach does not require explicit sorting, is able to insert fragments between already rendered depth layers and has a tightly packed memory structure. The algorithm provides artifact-free composition of interleaved objects when rendered with transparency effects and the memory can be accessed en-block for transfer. Thus, it is best suited for rendering object-space decomposed scenes with semi-transparent models in a distributed environment.

As can be seen in the evaluation, the PPLLs approach is slower than the other rendering techniques for single node rendering. However, the framerates are still highly interactive and having all fragments available allows distributed rendering with a reasonable amount of network traffic.

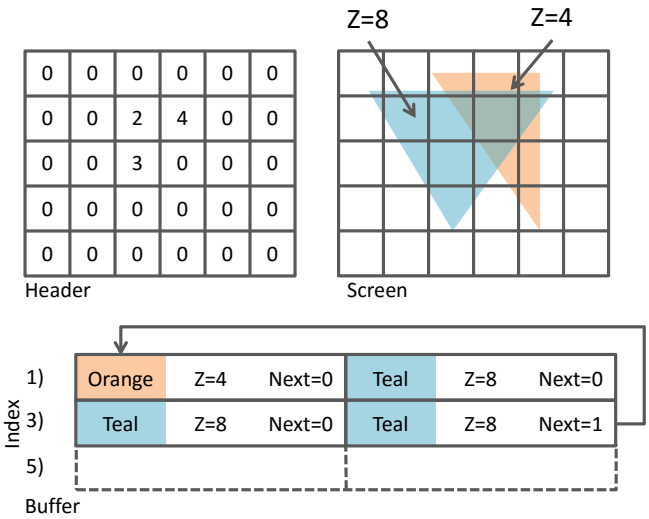
4.3 Per-Pixel Linked List Rendering Algorithm

The PPLLs algorithm collects and stores all fragments for later processing. This is achieved in a single render pass. The final image is created by sorting – either all fragments in the scene or locally for each per-pixel list – and then compositing each pixel's fragments according to the sorting order. After the data has been sorted, it is also possible to render the scene with depth-of-field effects (see Section 4.5.1) or to perform operations like Constructive Solid Geometry (CSG) on the fragments (see Section 4.5.2).

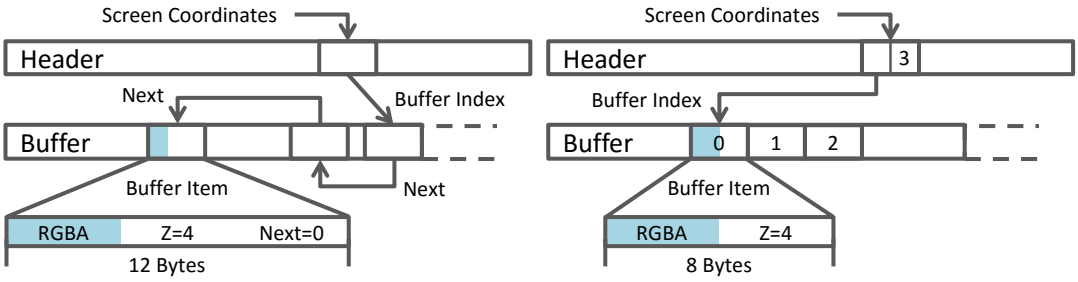
The whole PPLLs algorithm consists of four rendering stages: *clear*, *render*, *sort*, and *display* (see Figure 4.2). It utilizes a global atomic counter to avoid race conditions on parallel writing operations and two buffers. The *header buffer* has the size of the final image and stores the entry indices for each list, pointing to the second buffer. The *data buffer* stores the elements of the PPLLs or the per-pixel arrays in an unordered fashion.

First, the *clear* shader resets all data structures, i.e. the global atomic counter and the header buffer. It is not necessary to reset the data buffer as it is newly built from ground up in each frame and the atomic counter indicates its length. Then, the *render* shader is used to store the fragments as mentioned above.

This results in unordered lists for each pixel, which need to be sorted according to fragment depth for blending. Sorting can be done either in a separate step or directly before blending. The approach described here uses a combination



▶ **Figure 4.3 —** Two triangles and the data structure of the PPLLs header and buffer. Only full fragments are shown for simplification.



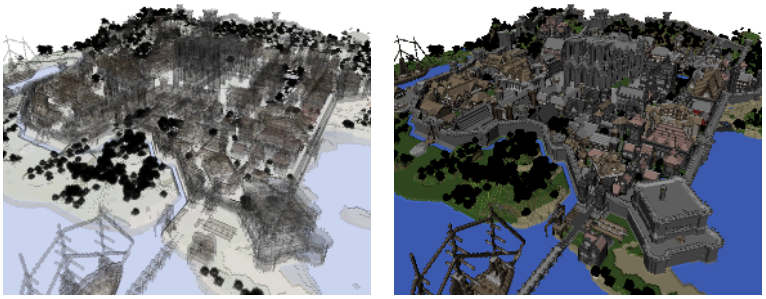
(a) Per-pixel linked lists (b) Per-pixel arrays

▲ **Figure 4.4 —** The buffer setup for PPLLs (a) and the array variant (b).

of both: for long arrays or lists, the data is sorted in global memory using an optional *sort* shader. Short arrays and lists are cached in a local array in the *display* shader and sorted directly prior to blending. Since everything is re-generated every frame, the cost of sorting the fragments can be assigned to any of the stages. The display shader also composes the depth-sorted data front to back and sends them to the OpenGL framebuffer. Figure 4.5 shows the Rungholt city model rendered with the PPLLs method.

Data is collected and processed using OpenGL graphics rendering and compute shaders, and the `GL_SHADER_STORAGE_BUFFER`s introduced in OpenGL 4.3. For each pixel of the viewport, the header buffer contains the entry index to the data buffer which stores the fragment attributes (cf. Figure 4.3 and Figure 4.4a).

An element of the data buffer contains the color and depth per fragment and may store additional attributes (e.g. fragment normal or texture coordinate). Per-



(a) Semi transparent (b) Opaque model

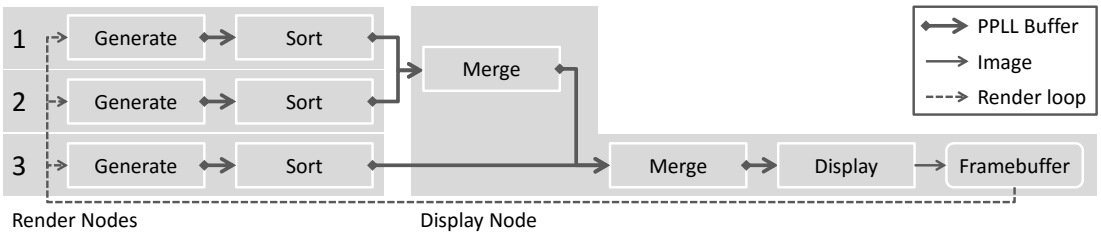
◀ **Figure 4.5 —** Semi transparent (a) and opaque (b) Rungholt city model rendered with PPLLs.

pixel lists are created during rendering by appending the attributes of a fragment to the list in the fragment shader. A global atomic counter which contains the index of the next free element in the data buffer is incremented in a thread-safe manner (`atomicCounterIncrement`). Existing fragment shaders and rendering methods can easily use PPLLs by replacing assignments like `gl_FragColor = color;` and `gl_FragDepth = depth` with `store(color, depth)` which appends the data to the fragment list of the current pixel.

This method can be used for rasterized geometry or implicit surfaces. For volume rendering, it is possible to insert every sample along the sampling ray into the PPLL buffers. However, this might lead to a very heavy memory usage and thus should be avoided. Lindholm et al. [2013] suggest to use a certain number of iso-surfaces extracted from the volume instead. The iso-surface renderings could than again be stored using the simple shader code replacements mentioned above.

When a scene additionally contains opaque fragments, such as the sticks in Figure 4.1a, they can be rendered into a separate framebuffer object first. Transparent fragments that lie behind opaque ones can be discarded during rendering, shortening the resulting lists.

As an alternative method for storing the fragments for each pixel, per-pixel arrays could be used, similar to a k-buffer but with a varying k for each pixel. Here, the number of fragments per array is counted in a first render pass, so the entry index for each pixel location can be calculated using a prefix sum (cf. Figure 4.4b). In a second render pass, each fragment is stored in the respective array within the data buffer. The arrays are created in the same way as the PPLLs by using an atomic per-pixel counter that contains the index of the next free element in the per-pixel array. Thus, the algorithm only differs for the storing method, the remaining parts work like in the PPLLs algorithm.



▲ **Figure 4.6 —** Setup of multiple renderers creating PPLLs buffers for one renderer. As the stages all do produce the same type of buffers, the merge step can easily be integrated in the existing pipeline (cf. Figure 4.2). The clear and render stage are pooled into the *Generate* stage for simplification.

Conceptual Extension for Multiple Renderers

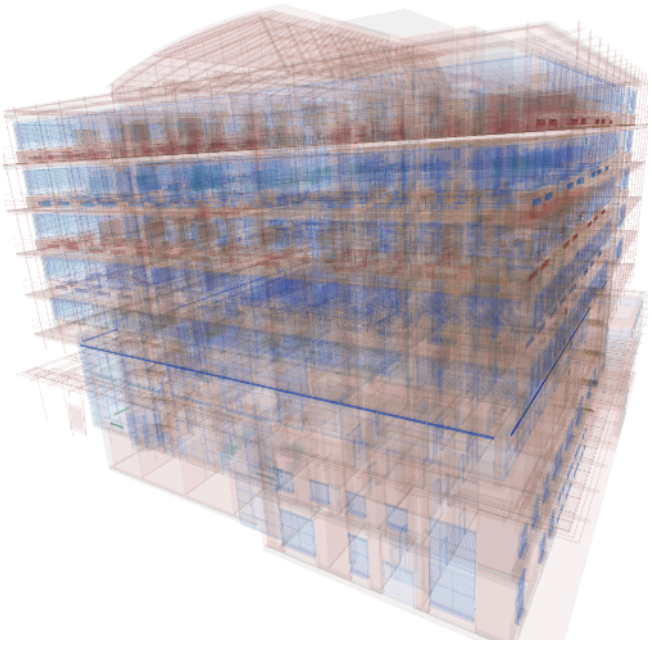
PPLLs have the ability to store all generated fragments of a scene for a given view. As the sorting of the fragments takes place right before blending in the last stage, it is possible to combine multiple buffers, even from different nodes, without the risk of rendering artifacts.

The so called *merge* step collects all buffers and is placed right before the *display* step. This method allows the PPLLs approach to be used for single node rendering as well as for parallel or distributed rendering. Figure 4.6 depicts the setup for three renderers working for one display node. As all stages use the same interface, the buffers, they can easily be reconfigured, allowing to merge multiple outputs for one (or multiple) display nodes. This also allows for more complex setups, e.g. a pyramid-like scheme with mixed hierarchical render-sort-merge-steps. For example, in a system with multiple render nodes, before sending the PPLLs buffers over the network, the merge-steps could be used to filter out fragments which are guaranteed to be invisible in the final image. This could be achieved by sharing the depth-buffers of the opaque renderings with all render nodes, effectively reducing the amount transferred and thus increasing the overall performance.

The next section details and evaluates the implementation of the distributed rendering using PPLLs using the described approach.

4.4 Distributed Transparency Rendering

Using the flexibility of the PPLLs, it is possible to render object-decomposed scenes which include semi-transparent models in a distributed environment without clipping. For distributed rendering, an arbitrary number of render nodes is connected to a number of display nodes via some sort of network.



◀ **Figure 4.7 —** View on the MPI Informatics Building model, rendered on eight render nodes. Each holds an equal-sized part of the model. The color represents the decomposition.

Here, a $1 : n$ correspondence between display and render nodes is assumed for simplicity. For a $m : n$ relation, the display nodes have to be synchronized. This can be achieved via a simple protocol where each display node broadcasts the maximum available frame number. Using the broadcast, each display node also knows the state of all the other nodes and the newest frame available to all nodes is displayed simultaneously.

Render nodes do the actual rendering work of the scene. Their output is then transferred to the display nodes which are connected to the output device(s). They compose the received data into a final image. Note that the methods described here are also applicable when using multiple graphics cards in a single node.

With the PPLLs algorithm, it is possible to render a scene that has been decomposed in *object space*. In the context of this thesis this means that the geometry of the object is decomposed without clipping or bricking. More precisely, the data is divided into geometry subsets which may intersect and overlap each other. That is, multiple render nodes may contribute to the same output pixel on the final image.

The challenges for distributed rendering, especially when all fragments of a scene are captured, are the vast amount of data sent and the compositing step on the display node as there will be more than one fragment per pixel location in transparent scenes, thus increasing the overall fragment count. An algorithm to overcome this will be presented later (see Section 4.4.3).

4.4.1 Merging Transparent Renderings from Multiple Render Nodes

The merge operation can take place either in image-space, i.e. pixel-wise on the rendered images. Or it can take place in object space where the actual 3D objects and their dimensions or positions are taken into account.

Merging the buffers from the render nodes in image-space means putting the data at the respective position of the output image. When a render node generates a simple pre-composed image, no other node may render on the same pixels as no other fragment can be inserted any more behind the generated fragment or between the flattened fragments. Thus, for object space decomposition where the objects may overlap in image-space, the Painter's algorithm is not able to generate artifact-free images without further effort. To use the algorithm, the geometry of the object has to be sorted before distributing it among the render nodes.

In the following, only rendering of object space decomposed scenes without bricking or clipping is discussed. This implicitly requires a sort-last approach. Here, the objects can intersect and overlap each other as seen in Figure 4.7: this particular decomposition is based on the structure of the MPI Informatics Building Model [Havran et al., 2009] which is already provided in sections. For distributing the model, these sections have been combined into meshes of equal size in terms of memory usage.

A complementary concept is image-space decomposed rendering. Here, every render node is assigned an area in the output image. This eliminates the need for a merging step, making the approach trivial. However, it does not allow for distribution of the models in the scene. Depending on the camera, one render node might have to render a large section of the scene and thus it would need the complete data set available. Especially for scenes that do not fit into local GPU memory like the MPI Informatics Building Model (see Figure 4.7) the image-space decomposition is therefore not feasible.

Furthermore, for the multi-variant analysis of proteins (see Section 4.6.2), each variant is simulated and rendered on a different node. For image-space decomposition, each render node would need to know all of the data sets in its view-space, resulting in a significant communication overhead that can be avoided in the object space decomposition approach. Therefore, object space decomposition is the desired approach for the applications using distributed transparency rendering (see Sections 4.6.2 and 4.6.3).

As described in Section 4.2, texture-based OIT implementations are not suitable for distributed rendering. Depending on the details of the algorithm, the

available graphics memory, and the capabilities of the graphics card, the result might be limited to a certain number of depth layers. This might have negative influence on the visual quality of the final image or produce incorrect results, especially if all fragments need to be available for complex operations like in the CSG algorithms.

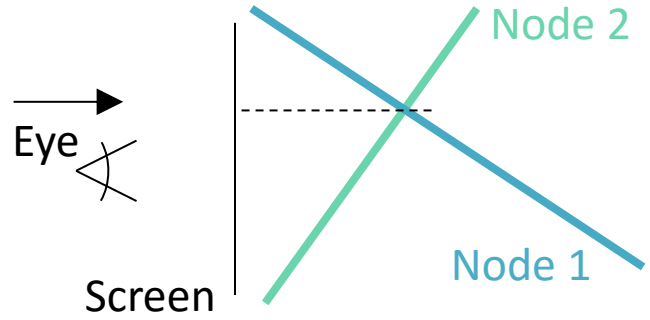
Whereas DP can handle scenes with an arbitrary number of depth layers for local rendering, the *k*-buffer uses a pre-defined constant and implementations like the bucket depth peeling [Liu et al., 2009] have a number of pre-defined buckets for each output pixel which are limited by hardware constraints. These constants directly impact the maximum number of depth layers that can be captured. Even if all fragments are captured, some of the output buckets or memory might not be used, when the scene does not have uniform depth complexity. This may waste valuable graphics memory and network bandwidth. An example for this is detailed in Table 4.1 and Figure 4.9: even for the simple dragon scene it is obvious that there is a high variation in the depth complexity for the whole image.

When rendering in a distributed environment, all of those textures need to be transferred over the network to the display node. Once the display node has collected all textures, they have to be merged for blending. As for the render nodes, the limiting factor is the number of attachable textures per render pass. When all textures can be attached at once, the sorting can be realized as a single merge sort step at the pixel level because there is potentially no globally valid sort order for the textures.

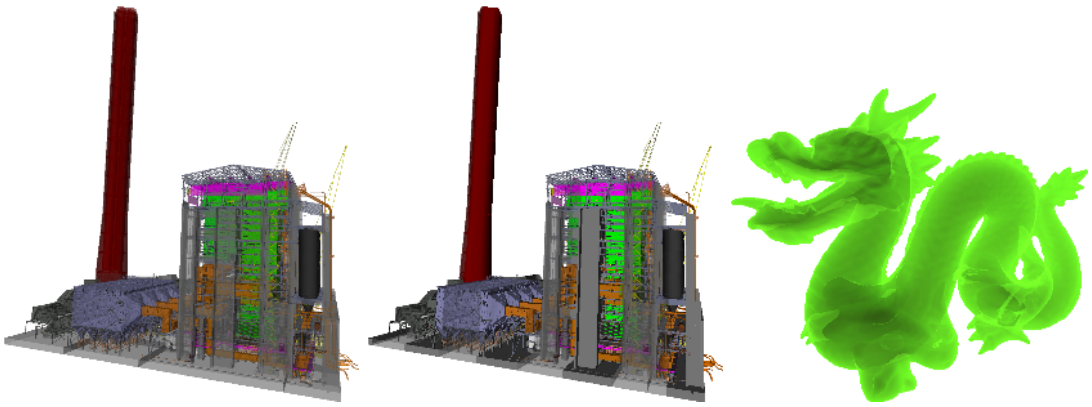
Having multiple textures, merging the received data is far more complex: The merging algorithm needs to determine a global order for each pixel to compose the image correctly. For an arbitrary scene, the problem is that the order of the textures might be different for each output pixel, making this step very complex. This is the case if two or more polygons that are rendered by different nodes intersect each other in object space. Figure 4.8 demonstrates this. Due to the limited number of texture slots on the GPU, this operation has to be performed by the CPU which is very costly. When the merging and sorting is complete, the collected data can be blended and displayed.

4.4.2 Using Per-Pixel Linked Lists for Distributed Transparency Rendering

The PPLLs approach is an ideal candidate for using it with distributed rendering. As explained above, with this approach it is possible to insert fragments between existing ones because all fragments are collected and kept until they are displayed.



▶ **Figure 4.8 —** A minimal example demonstrating the sorting problem for two nodes. Above of the dotted line, node 1 is in front, below of the dotted line, node 2 provides the front layer.



(a) UNC power plant 1 (b) UNC power plant 2 (c) Stanford Dragon

▲ **Figure 4.9 —** PPLLs examples of the UNC power plant rendered with fully semi-transparent colors (a), with opaque stripes (b), and the Stanford Dragon (c).

For PPLLs approaches, opaque and semi-transparent fragments can be treated differently. As recommended by Yang et al. [2010], opaque fragments are rendered first by discarding all semi-transparent fragments in the first geometry rendering pass. The output consists of two textures containing the color (tex_c) and depth values (tex_z) of the opaque fragments. Alternatively, the opaque fragments can simply be stored in the list, too.

When rendering semi-transparent fragments, the depth texture generated in the first render pass can be used to discard all fragments behind the nearest opaque fragment. The discarded fragments are not visible in the final image either way. Discarding them immediately results in less fragments stored, thus leading to faster sorting and blending, and saves memory bandwidth during the transfer.

After the PPLLs have been generated on the render nodes, they are sent to the display node together with the color and depth texture of the opaque fragments.

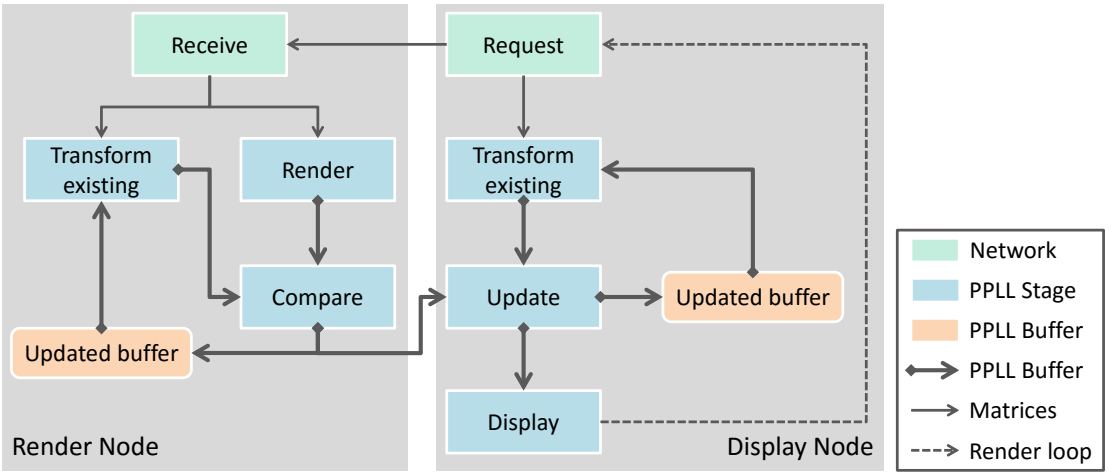
When all data is collected, the textures are composited by taking over the minimal depth value of all opaque fragments and the respective color to tex_z and tex_c of the display node. In the case of unordered lists, they are simply concatenated, forming a longer list which is sorted thereafter. For already sorted lists, they are combined in a list-sorting step: Given two ascendant ordered lists, append the smaller value to the new list and advance the pointer of the respective list. This is repeated until the end of both lists is reached. Thus, the newly generated list is also sorted and the merge step can be executed again with the next buffer, if available. tex_z can again be used to discard fragments which are behind the opaque fragment. Having the per-pixel lists sorted by depth, they can be blended and displayed on the screen. This method is suited best for object space decomposition as the step can either be executed on the render node or on the display node. The complexity is equal to a simple list operations for the each output pixel.

When using PPLLs for distributed rendering, the following data has to be transferred for each frame from the render nodes to the display node:

- Color and depth texture, if the scene contains opaque fragments.
- Header buffer with the entry for each per-pixel list.
- Data buffer containing the actual list elements.

Each list element contains a next-pointer and its payload. In this case, the payload consists of two 32-bit values: the depth, and the color of the fragment or its normal. Transferring the normal to the display node makes it possible to recolor the fragment using per-pixel lighting. This requires the display node to have access to the scene lighting data but this can easily be shared in a setup step before the actual rendering takes place. In a shader, the color is usually represented as four 32-bit floating point values. To reduce the size, each color channel is stored in 8-bit integers in this approach as this is what most output devices can display. Here, the quantization can be justified by the data compression ratio of 75%. On the display node, the color is used for deferred shading and for blending the final image.

When storing normals instead of colors, the normal vector is converted to spherical coordinates, thus only two values need to be stored. Both components can be stored in a 16-bit floating point value [Gross and Pfister, 2007]. Again, the precision reduction is justified by the compression ratio of 66%. When receiving a normal from the render nodes, the display node uses it to compute the per-pixel lighting in the blending stage.



▲ **Figure 4.10 —** Instead of sending the complete rendering, only updates are transmitted which are used in combination with the last frame to compute the new image on the display node.

4.4.3 Two-sided Fragment Transformation

In the approach described above, all fragments generated on the render nodes are sent entirely to the display node (see Figure 4.6). It is obvious and proven by numbers (see Section 4.4.4), that such large frame updates have a negative influence on the overall performance of the system.

When a frame is rendered, both the render node and the display node hold buffers containing the fragments of the current view. To reduce the amount of memory transferred, differential updates can be used. For a new view, the render node creates a new buffer which holds the complete scene for this render node. Additionally, the render node uses the buffer from the last frame and transforms the fragments into the new view. Then, both buffers are compared and only fragments which are new or do no longer exist are sent to the display node. The display node then uses its buffer from the previous frame and the newly received differential buffer to update its main buffer and displays it. Therefore, the render node always knows the state of the display node buffers. Periodical updates will help to avoid rounding error effects in the display node buffer. When there is no movement or camera transition, no update is requested from the render nodes. These idle moments can also be used to request a full update so the user experience is not disrupted. Figure 4.10 shows the work flow of the two-sided fragment transformation algorithm.

When using multiple models with different transformation matrices, e.g. in a scene with multiple animated models, the same scheme can be used. However,

instead of merging the results into one buffer, each model has its own buffer. This is necessary because the transformation algorithm needs the correct matrix to transform the fragments of a given model back and forth. With other fragments than the ones belonging to the matrix in the same buffer, wrong results will occur. When all render nodes have sent their updates and the buffers of each model were updated, all of these buffers are merged into a final buffer and then the final buffer is displayed.

Implementation Details

The standard transformation for a vertex v from object space to camera space works as follows:

$$v_{camera} = M_{view} * M_{model} * v \tag{4.1}$$

In this approach, each PPLLs entry stores the complete xyz-coordinate of the fragment and the normal vector in object space. This is necessary to minimize the rounding errors of the matrix multiplications and to keep the transformation chain as correct as possible in comparison to the newly rasterized fragments. As the fragments are stored in camera space, the depth ordering still applies here. Additionally, the entries contain the normal vector in object space, encoded in memory saving 2×16 bit angular notation. This allows for comparing the fragment also for the normals and therefor enables an independent deferred shading on the displaying node, similar as described beforehand (cf. Section 4.4.2).

When the existing fragments from the last frame are transformed, the inverse view matrix M_{view}^{-1} and inverse model matrix M_{model}^{-1} from the last frame are used to get the stored fragment position in camera space v_{camera} back into the object space v:

$$v = M_{view}^{-1} * M_{model}^{-1} * v_{camera} \tag{4.2}$$

Then, equation 4.1 is applied to v again, using the new model transformation and thereby putting it into the new camera space of the current view.

Case-by-Case Analysis

For comparing the actual rendering output buffer with the transformation buffer of the previous frame, the render node advances both lists for a given pixel. Fragments are seen as equal if the depth value and normal vector are within a given epsilon value. There are four cases (cf. Figure 4.11a):

1. If both fragments are the same, the display node has the correct fragment and is able to transform it to the correct location. Thus, the fragment is not transferred.

2. If a fragment is only available in the newly rendered buffer, it is a new fragment which cannot be put there by a transformation on the display node. It has to be taken over into the update buffer to be sent to the display node.

3. If a fragment is only available in the transformed buffer from the last frame but not in the newly rendered buffer, it has to be discarded. A special flag is used so the display node can distinguish between new fragments and fragments marked for deletion.

4. If two fragments exist at the same position but with different content, the newly generated fragment wins and is sent to the display node.

Upon receiving, the display node traverses the buffer from the last frame and the newly received buffer. Again, there are four cases (cf. Figure 4.11b):

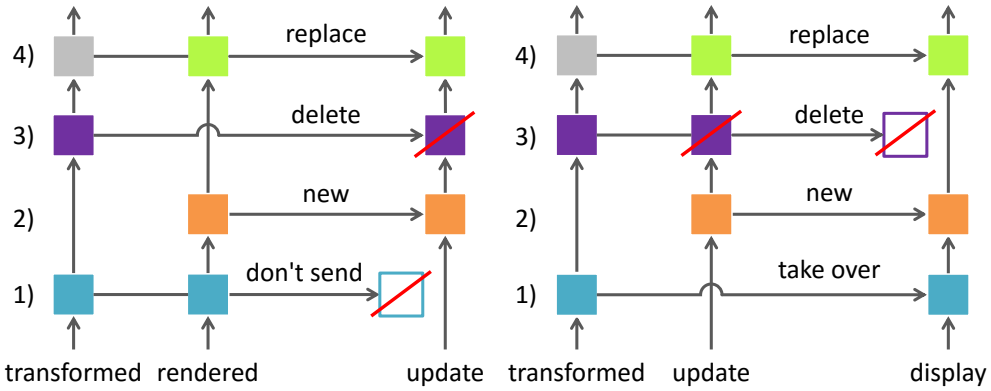
1. If a fragment is available in the transformed buffer but not in the update buffer, it is left out to save space (case 1 from above).

2. If a fragment is in not in the previous buffer but only in the newly received one, the fragment is new and has to be taken over (case 2 from above).

3. If a fragment is in the last frame and in the new buffer and has the deletion-flag set, it is to be removed (case 3 from above).

4. If two fragments exist at the same position but with different content, the newly generated fragment wins and replaces the corresponding fragment on the display node (case 4 from above).

For this method, it is necessary to store the precise xyz-coordinate of the fragment to correctly transform it, making the fragment slightly larger in comparison to the standard approach. This is due to the fact that when the fragments are transformed, they do not fall exactly into the pixels of the screen but may have a slight offset. For a second transform, this offset has to be taken care of to minimize the error that could accumulate rapidly otherwise, falsifying the final image.



▲ **Figure 4.11 —** All four example cases of how fragments are compared and taken over into the update buffer on the render node (a), and how the fragments are used on the display node (b).

4.4.4 Discussion

Table 4.1 gives an overview of different OIT algorithms and their performance for two different meshes in two different configurations. All benchmarks were run on Intel Xeon E5620 nodes with an NVIDIA Geforce GTX 480 using a view port of 1024×1024 pixels. The scenes corresponding to the table columns are shown in Figure 4.9. As said above, the significant disadvantage of the texture-based methods is the memory consumption which is especially problematic when transferring the data over the network. Additionally, the rendering performance itself is partially very low for those methods. The calculations for the memory usage are based on the fact that for compositing, the display node needs every layer and that it is not feasible to combine them on the render node. Note that the stencil routed k-buffer provides a limited amount of depth layers, resulting in the relatively small but constant memory usage.

The bucket DP approach is also faster than the PPLLs, but its limited depth layer capacity uses more memory than the PPLLs for the complete scene. Additionally, the PPLLs can provide reasonable performance while keeping the memory overhead at a minimum. Note that for opaque fragments, as seen in the *UNC 2* case, the amount of data which needs to be transferred decreases further for the PPLLs. The reason is that opaque fragments are stored separately and non-opaque fragments behind the nearest opaque fragment are skipped during rendering. The other approaches do not benefit from separating the opaque fragments as those approaches are not modified for the evaluation. Additionally, the texture-based approaches still need a full-sized texture for each transparency

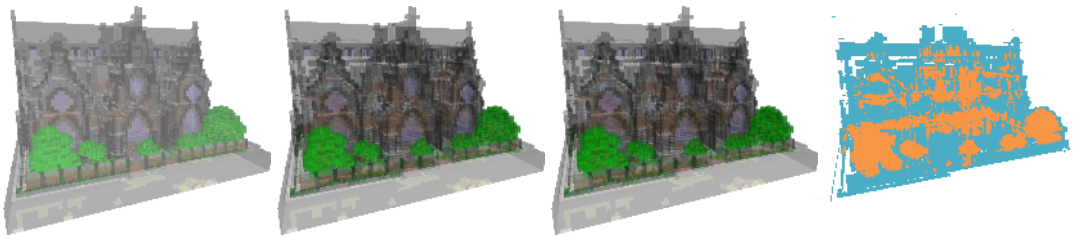
Test case	UNC 1		UNC 2		Dragon	
Polygon count		12.7M		12.7M		871k
Opaque fragments		0		1.9M		0
Transparent fragments		14.7M		4.7M		1.1M
Maximum depth layers		234		234		12
	FPS	Memory	FPS	Memory	FPS	Memory
Raw/Unsorted	44.1	-	43.9	-	434.3	-
Bucket DP[1]	8.6	256.0	8.6	256.0	72.6	96.0
Adaptive Bucket DP[1]	4.8	256.0	4.9	256.0	32.9	96.0
DDP[2], all layers	0.3	1,872.0	0.3	1,872.0	30.3	96.0
DDP[2], 32 layers	2.3	256.0	2.3	256.0	30.3	96.0
k-Buffer[3]	13.9	32.0	13.9	32.0	60.7	32.0
PPLLs[4]	4.3	180.0	8.2	94.9	56.5	17.3

▲ **Table 4.1 —** Frame rate and memory (in MB) needed for distributed rendering for different cases: UNC power plant with full transparency (see Figure 4.9a) and a large number of opaque fragments (see Figure 4.9b), as well as a fully transparent Stanford Dragon (see Figure 4.9c). [1]) Liu et al. [2009], [2]) Bavoil and Myers [2008], [3]) Bavoil et al. [2007], [4]) Yang et al. [2010]

layer which is empty for the opaque or unpopulated sections, thus wasting valuable graphics memory and network bandwidth.

For most of the algorithms, the amount of data that has to be transferred from the render nodes to the display node is enormous. Other algorithms can only provide a limited number of depth layers which reduces the visual quality. This might become an issue when the scene is very complex. Figure 4.12 shows the house model with different number of depth layers. While the alpha value is saturated at 12 layers for this example, the overall layer count is 32. Having all layers available might not be important for the final image, but it makes a difference for advanced effects, e.g. the CSG operations. The evaluation has shown that PPLLs can provide the full visual quality at a comparatively moderate memory usage. Compressing the data before sending them over the network can further improve the performance.

Table 4.2 shows the performance in FPS when using compression algorithms compared to no compression, averaged for a full rotation of the UNC powerplant (see Figure 4.9a) and the Stanford dragon model (see Figure 4.9c). Using the large UNC model, the GPU-based GFC [O'Neil and Burtscher, 2011] compression algorithm does not improve the performance, whereas it speeds up



(a) 3 layers (b) 8 layers (c) all layers (d) comparison layers

▲ **Figure 4.12 —** Renderings with different depth layer counts of (a) 3, (b) 8 and (c) all depth layers. For the given view, the alpha value is saturated at 12 layers, the full depth count is 32. (d) shows the difference of 3 and 8 layers (teal), and 8 and all layers (orange).

Model	None	LZO	GFC
Dragon	2.8	2.4	4.1
UNC	0.4	0.3	0.4

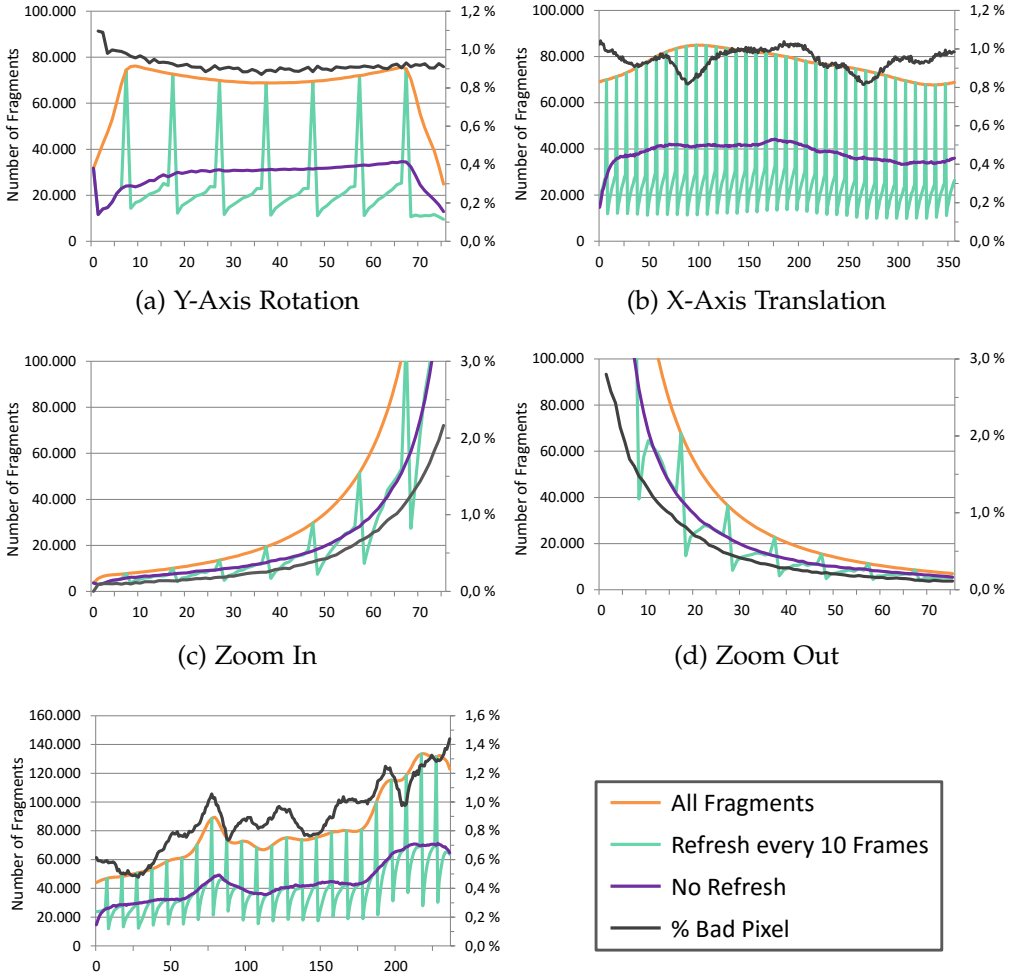
◀ **Table 4.2 —** Performance in FPS for two different compression algorithms LZO [Oberhumer, 2011] and GFC [O'Neil and Burtscher, 2011] compared to the plain network rendering for the power plant building (four render nodes) and the dragon model (one render node).

the dragon by about 45%. LZO [Oberhumer, 2011] was chosen as an alternative CPU-based compression algorithm for its high speed and lossless compression [O'Neil and Burtscher, 2011]. In this case, the computation overhead of the compression and decompression weighs far more than what is gained through the faster data transfer.

To evaluate PPLLs in a distributed rendering environment, two applications are presented later. PPLLs are small in size, flexible and easy to handle, still providing a reasonable performance. A visual analysis of multiple remotely rendered variants of a protein data set (see Section 4.6.2) and a simple mesh renderer (see Section 4.6.3) are used as proof of concept.

Two-sided Fragment Transformation Evaluation

Figure 4.13 shows the evaluation of the approach using the two-sided fragment transformation. For this, different viewing operations where used: rotation (cf. Figure 4.13a), translation (cf. Figure 4.13b), zoom (cf. Figure 4.13c and Figure 4.13d), and a combination (cf. Figure 4.13e) of all of them together. Using this approach, the fragments to be transferred could be reduced to about 40 % to 60 % of the total fragment count.



(a) Y-Axis Rotation

(b) X-Axis Translation

(c) Zoom In

(d) Zoom Out

(e) Complex Movement

▲ **Figure 4.13 —** Evaluation for different object and camera movements using the two-sided fragment transformation. Without refreshing (purple), the fragment count is at about 40 %-60 % of the total fragment count per frame (orange). The intermediate approach with updates every 10 frames (teal) converges towards the non-refreshing approach in between the full updates. Additionally, the percentage of bad pixels compared with the original image is shown (black).

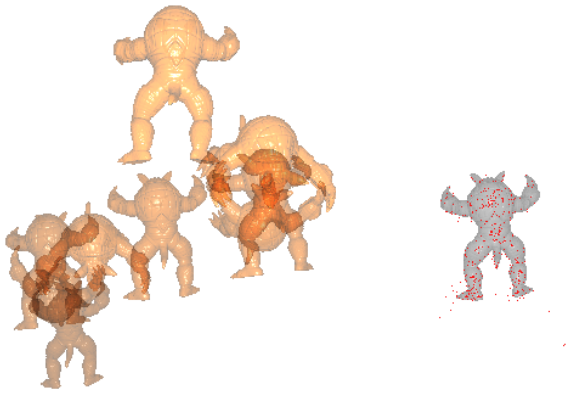
Refresh	per Frame		per 10 Frames		no Refresh	
Operation	FPS	Frag.	FPS	Frag.	FPS	Frag.
Full Y Rotation	5.51	65.4k	6.48	23.3k (35%)	5.88	28.4k (50%)
X Axis Movement	4.90	76.6k	5.98	26.7k (36%)	4.63	38.3k (43%)
Zoom In	7.36	43.4k	7.04	24.6k (57%)	7.22	23.9k (55%)
Zoom Out	5.30	72.9k	5.46	43.9k (60%)	5.34	45.1k (62%)
Complex Movement	4.81	79.3k	4.97	40.6k (51%)	4.67	43.6k (55%)
(large ϵ)	4.76	79.3k	4.15	70.7k (89%)	3.87	75.0k (95%)

▲ **Table 4.3 —** Frame rate and fragment count in thousands for the evaluated scenarios. The values in brackets denote the fragment percentage compared to the full refresh for every scene (“per Frame” columns).

Table 4.3 displays the average frames per second and average fragment count for the evaluated scenarios. The best average performance is delivered when a complete refresh is sent every ten frames. For one frame, there might be a higher network traffic, but for the first subsequent frames, the traffic is significantly lower. This also saves computation time for the transformation process for the full update and the subsequent frames. When no refresh is performed, the erroneous fragments have to be detected by the transform shader and removed in a delta update.

Another effect that can be seen is when the epsilon used for the identification is chosen to loosely (last row of Table 4.3). Here, the update mechanism sends far too many fragments. For the first difference after a full update, many fragments wrongly survive. Those wrong fragments have to be removed in the subsequent frame, causing more network traffic, and in addition, the correct frames have to be transferred, too.

As seen in Figure 4.13, the image quality is relatively stable for standard view transformations. However, it sums up when performing multiple view changes. In the example shown, only the difference frame was used and the image is compared to the full update for each frame. Figure 4.14a shows multiple steps of the complex movement (cf. Figure 4.13e) as an overlay. Some faulty pixels can be seen (cf. Figure 4.14b) which result from rounding errors during the transformation and are responsible for the not-perfect image quality. Additionally, when a transformed fragment is used instead of the newly rasterized one, the interpolated data, in this case the normal vector, might not exactly fit. However, this might not be visible to the viewer, especially if the perfect picture is not shown, but still causes a bad pixel. Using full updates can remove the errors, effectively starting from a perfect frame again.



(a) Multiple time steps (b) Highlighted bad pixels

▶ **Figure 4.14 —** Collage of multiple steps of the Armadillo model (a) performing the complex movement (dark to light color, cf. Figure 4.13e). Close view on the bad pixels (red) of a frame during the animation, original image converted to gray scale for better contrast (b).

4.5 Advanced Rendering Techniques using Per-Pixel Linked Lists

Using PPLLs and having all fragments available enables the realization of advanced rendering effects like depth of field or CSG. While the former is a nice effect and can be used for attention guiding, the latter is useful for combining models and the creation of complex objects without artifacts. Both techniques are explained in the following.

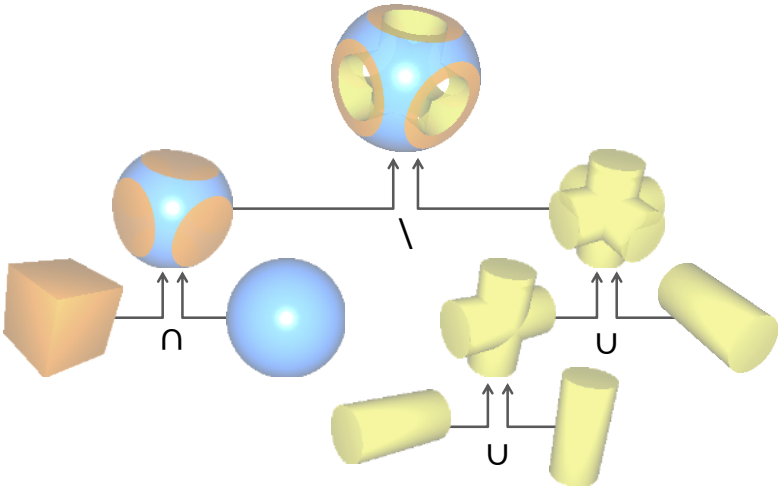
4.5.1 Splatting for Depth of Field Effects

Instead of rendering a full screen quad, the PPLLs data can be displayed directly using splatting [Westover, 1990]. For this purpose, each element stores the full 3D position and color, allowing a fast sorting of all fragments. Because of the flexible extensibility of the PPLLs data structure, those attributes can simply be added to the elements of the data buffer. After rendering the scene into the PPLLs buffers, the data is mapped to CUDA and sorted using the Thrust library. This destroys the structure of the list buffer, effectively converting the data into an array of vertices ordered by depth. The complete buffer is mapped as a vertex buffer and rendered as points. This can be extended, for example to create a depth-of-field effect (see Figure 4.15), using a geometry shader that creates quads whose size depends on the distance to the focal point.

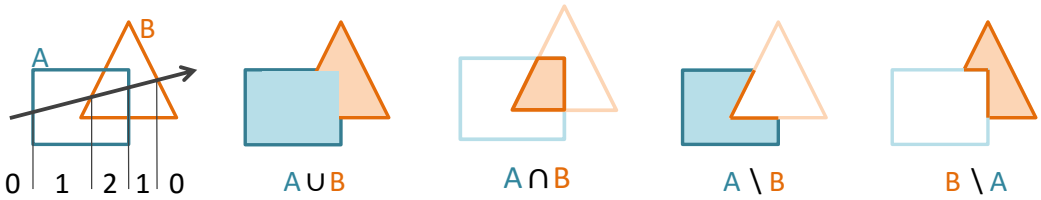
4.5.2 Constructive Solid Geometry

Using the PPLLs approach, all fragments in the scene are available for compositing. This makes the implementation of CSG operations possible. For example,

◀ **Figure 4.15** — Depth-of-field example with a Stanford dragon and two Buddha figurines. The focus plane is on the left Buddha and on the tail of the dragon.



◀ **Figure 4.16** — CSG operation example of a cross made of three joined (∪) cylinders which is subtracted (\) from the intersection (∩) of a cube box and a sphere.



(a) No Operation (b) Union (c) Intersection (d) A without B (e) B without A

▲ **Figure 4.17 —** Schematics of the different CSG operations and the counter values for the traversing ray. The shaded figure is the result of the CSG operation.

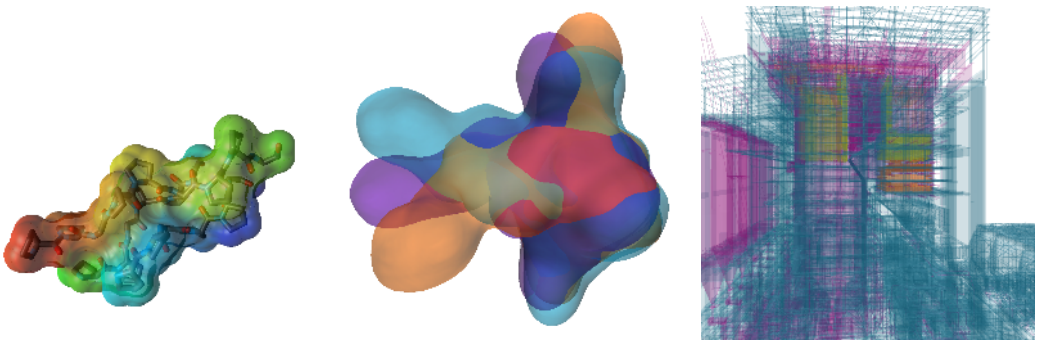
the *union* operation (cf. Figure 4.17b) can be implemented as follows: when composing the sorted fragments, the layers are counted and a fragment is rendered only if the counter is incremented from 0 to 1 (entering the object) or decremented from 1 to 0 (leaving the object). This represents the outer border and no internal structure or overlapping geometry is composed into the final image. The counter increases for fragments facing the viewer and decreases for fragments facing away. The orientation of a fragment is determined in the *render* stage and stored as algebraic sign of the depth value. Therefore, all depth-dependent checks are performed on the absolute depth value for consistency. The molecular surface rendering (cf. Section 4.6.1) uses CSG to correctly discard fragments which are inside the surface but should not be displayed.

The two other basic CSG operations – *intersection* (cf. Figure 4.17c) and *subtraction* (cf. Figures 4.17d-e) – can be implemented analogously (see Figure 4.16). For the intersection, the fragments are rendered only when the counter is incremented from 1 to 2 or decremented from 2 to 1. In case of the subtract operation, the fragments are rendered when the ray enters or leaves the first object, and when it enters the second object. For the latter case, the normal vector has to be inverted for rendering as the inside of the second object is the outside of the resulting object. More complex CSG operations, e.g. *XOR*, are logical combinations of these three basic operations.

For rendering more complex objects with more than two parts, the CSG operations are segmented into binary operations. Only the final result is displayed after the CSG operations are completed.

4.6 Applications

The following sections will detail three possible applications for the PPLLs algorithm. First, rendering molecular surfaces is explained (cf. Figure 4.18a and Section 4.6.1). Here, the OIT technique is used for an artifact-free rendering of



(a) 1A3I Protein (b) 2JQW Protein (c) UNC power plant

▲ **Figure 4.18 —** PPLLs examples molecular surface rendering (a), the multi-variant visualization of different states of a protein derived from a simulation (b), and a distributed rendering of the UNC power plant (c). Here, the colors indicate the data distribution with one color per render node.

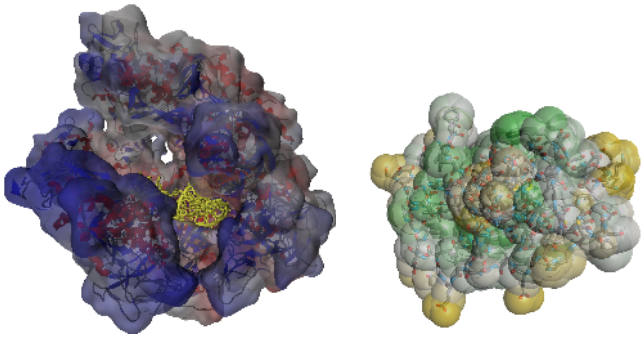
semi-transparent molecular surfaces consisting of ray casting implicit spheres and torus objects. The transparency of the surface enables e.g. a combined visualization of the surface and other representations like the ball-and-stick visualization, thereby delivering more information and context for the users and domain experts.

Section 4.6.2 extends the molecular surface rendering and uses the distributed PPLLs approach detailed above (cf. Section 4.4) to deliver a multi-variant analysis for molecular surfaces. Here, each render node creates the same view of a slightly different variant of the same object and the display node merges the results, augmented with information about common equal and non-equal parts of the variants.

The third application (cf. Figure 4.18c and Section 4.6.3) is a distributed mesh renderer, e.g. for models which are too large for a single graphics card. Another purpose is to use it for in-situ simulations with object decompositions which intersect each other. The display node receives the buffers from the render node and displays the final image. The use-case is that a simulation node computes the output and also directly creates the PPLLs buffers for visualization. For the presented application, a static mesh is used that is loaded at startup.

4.6.1 Rendering Molecular Surfaces using Order-Independent Transparency

Many fields of research like chemistry, physics, and biomedicine work with molecular data. This data can originate from purely computational sources (e.g.



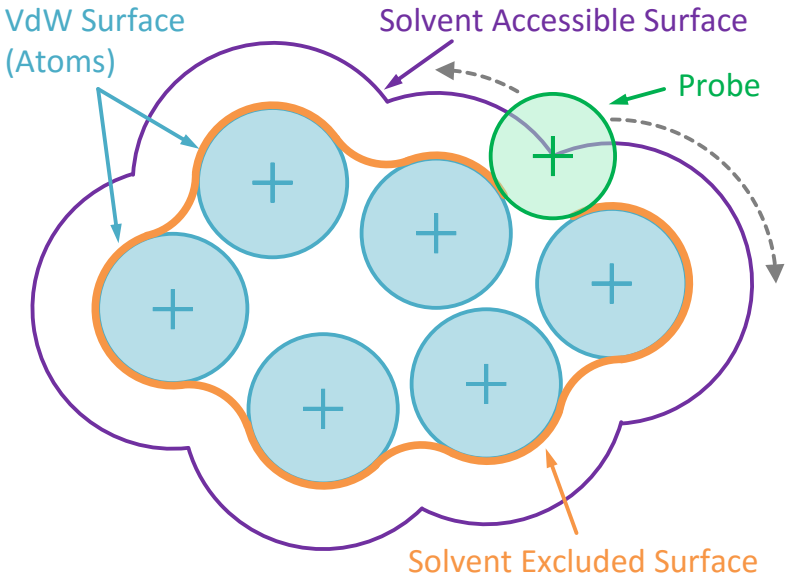
(a) 1I6H Protein (b) 1AX8 Protein

▶ **Figure 4.19 —** (a) Transparent Quicksurf surface with internal cartoon representation and DNA (yellow parts) and (b) SAS representation with internal ball-and-stick visualization.

molecular dynamics simulations, normal mode analysis, or fitting calculations) or from measurements (e.g. x-ray crystallography or scanning tunneling microscopy). High-quality visualizations are an important tool for the analysis of these data sets. Many different molecular models have been developed to that end. Among these models, molecular surface representations are widely used. They are beneficial for many applications since they show the boundary of a molecule with respect to a certain property, in this case the accessibility for other molecules. Naturally, transparent surfaces do not fully obscure other parts of the scene. That is, users can see through objects and spot features or changes on the backside immediately without changing the perspective. Furthermore, users are able to see inside molecules. This makes transparent molecular surfaces especially suited for complex analysis tasks where the user wants to see the molecular surface but also another, sparse representation of the molecule like a stick model (cf. Figure 4.19). Multiple nested representations of the molecule can be used to provide more information to the user.

Molecular surfaces are defined as a set of implicit surface patches (see Section 4.6.1 below). Thus, they can be rendered using GPU-based ray casting [Gumhold, 2003; Reina and Ertl, 2005]. This results in higher visual quality and faster rendering while additionally saving the computation time and memory for the triangulation. However, for transparent rendering, it poses an even greater problem than triangulated surfaces. The implicit surface patches might not only overlap and generate multiple fragments which have to be sorted according to their depth, but there are also parts of these surface patches which have to be removed to obtain a correct final image.

The following sections detail how this method can be used to render transparent molecular surfaces in real-time and compare the PPLLs approach to a freely available existing DP OIT approach from the NVIDIA SDK.



◀ **Figure 4.20** — Schematics of the SAS (purple) and the SES (orange), both defined by a probe (depicted green in a sample position) rolling over the molecule atoms (teal).

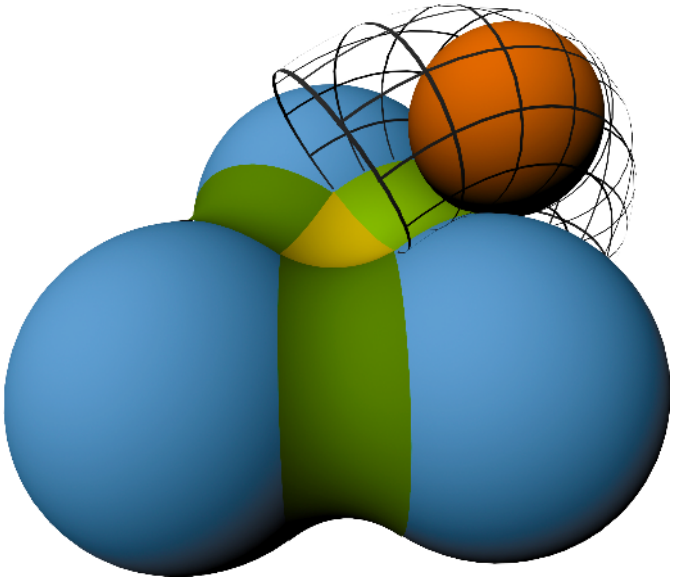
Molecular Surface Definitions

There are several definitions for molecular surface models, which are based on physical properties of the atoms.

The Van-der-Waals Surface (VdW) shows the extent of a molecule. All atoms are represented by spheres with the corresponding VdW radius of their element, which is derived from the distance between non-bonded atoms. Figure 4.20 shows a 2D schematic of the VdW surface (colored teal).

A fundamental drawback of the VdW surface is that it has no correlation with the surrounding medium (e.g. solvent molecules or ligands). The Solvent Accessible Surface (SAS) [Richards, 1977] includes this information. It is defined by the center of a spherical probe (colored green) which rolls over the VdW surface of the molecule. Thus, the SAS represents the surface that is directly accessible for solvent molecules the size and form of the probe. Figure 4.20 shows a 2D schematic of the SAS (colored purple). The SAS closes small gaps and holes in the VdW surface, which cannot be entered by solvent molecules.

The Solvent Exclusive Surface (SES) (colored in orange) is the topological boundary of the union of all possible probe spheres that do not intersect any atom of the molecule [Richards, 1977; Greer and Bush, 1978]. The SES can also be defined by a spherical probe rolling over the VdW surface. Here, the surface of the probe traces out the SES (in contrast to the SAS, where the probe center is used). Figure 4.21 shows a schematic of the SES. The SES consists of three types of geometrical primitives: convex spherical patches, concave spherical triangles,



▶ **Figure 4.21** — 3D schematics of the SES depicting the spheres (blue), the spherical triangle (yellow) and the toroidal patches (green) defined by the probe (orange).

and saddle-shaped toroidal patches. The spherical patches are the remainders of the VdW spheres. They occur when the probe is rolling over the surface of an atom and touches no other atom. The toroidal patches are formed when the probe is in contact with two atoms and rotates around the axis connecting the atom centers (see Figure 4.21). The inward-looking, saddle-shaped patch of the resulting torus is part of the SES. While rolling, the probe traces out a small circle arc on each of the two VdW spheres. These small circle arcs are part of the boundaries of the aforementioned spherical patches. Spherical triangles occur if the probe is simultaneously in contact with three VdW spheres. Here, the probe is in a fixed position, meaning that it cannot roll without losing contact to at least one of the atoms. Please note that the probe can in theory be in contact with more than three atoms, however, these cases can be divided into multiple atom triplets for simplicity.

Similar to the SAS, the SES closes small gaps and holes, but it does not inflate the molecule. The extent of the VdW surface is retained. Therefore, the SES is the most versatile molecular surface and suitable for many applications and analysis tasks like docking or solvation. Metaballs [Blinn, 1982] is another molecular surface definition modeling the electron density surface. For some images in this chapter, Metaballs are depicted using the QuickSurf [Krone et al., 2012] method and the representation was exported from the freely available Visual Molecular Dynamics tool [Humphrey et al., 1996].

Molecular Surface Visualization

Molecular models can be rendered using ray tracing by sending primary rays from the view point into the scene. For every hit with an object, secondary rays are generated to capture reflection and refraction. Ray tracing generates very accurate images, but is computationally demanding. With the performance of current CPUs and GPUs, real-time ray tracing of complex scenes has become possible [Wald, 2004]. This, however, requires elaborate acceleration structures. BALLView [Moll et al., 2006], a molecule viewer, offers such real-time ray tracing [Marsalek et al., 2010]. Although ray tracing – including secondary rays – achieves interactive frame rates, point-based GPU ray casting [Reina and Ertl, 2005] has a considerably higher performance. The performance of ray tracing further decreases when using transparency. In the presented approach, effects which might require secondary rays are not used, thus GPU ray casting is sufficient.

The computation and rendering of the VdW surface and SAS is quite trivial, since they only consist of spheres. If the surface is drawn opaque, the interior parts of overlapping spheres do not have to be removed since they are not visible anyway. However, for transparent renderings, the interior parts have to be removed. If the spheres are composed of triangles, the intersections for each triangle have to be computed. Laug and Borouchaki [2002] presented methods for high-quality meshing of molecular surfaces. Today, implicit surfaces like spheres are commonly rendered using high-quality GPU-based ray casting. Clipping the interior parts would require neighborhood information. Since this would result in a plethora of additional intersection tests in some cases, available tools often do not clip the interior parts, which results in a cluttered representation when using transparency.

The computation of the SES is more complex and costly. Connolly [1983] presented the analytical equations to compute the SES. Based on his work, several methods to accelerate the computation of the SES have been introduced. Edelsbrunner and Mücke [1994] presented the *α-shape*. Sanner et al. [1996] developed the Reduced Surface (RS) algorithm. Krone et al. [2009] later used this RS algorithm for dynamic simulation data and they were the first to render the SES using fast GPU-based ray casting of the individual patches instead of triangulations. Varshney et al. [1994] described a parallelizable algorithm based on Voronoi diagrams. Totrov and Abagyan [1995] proposed a contour-buildup algorithm for the SAS, from which the SES can be derived. This algorithm was recently shown to be parallelizable on multi-core CPUs [Lindow et al., 2010] and GPUs [Krone et al., 2011]. Parulek and Viola [2012] presented an implicit representation for the SES using a ray casting approach. With the exception

of Parulek and Viola [2012], all aforementioned methods compute the patches of the SES. The technique best suited for the application may depend on different factors, e.g. hardware capabilities.

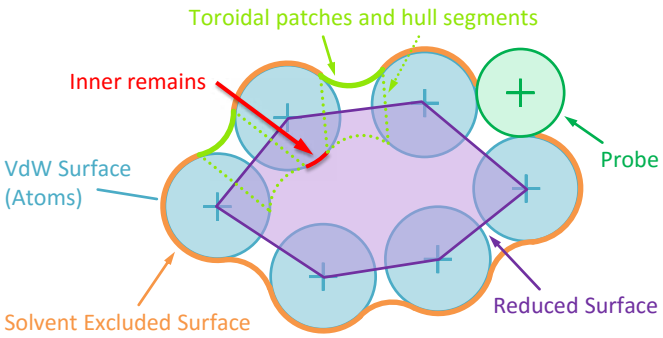
Transparent Molecular Surface Rendering

The VdW and the SAS representations consist only of spheres. As these spheres intersect each other, all parts of a sphere which are inside another sphere have to be removed since they would be visible when using transparency. The PPLLs method allows for removing these interior parts using the CSG *union* operation explained in Section 4.5.2. That is, all sphere fragments are stored in the data buffer during rendering. During the compositing stage, interior fragments are discarded to clear the internals.

As explained in Section 4.6.1, the SES consists of three types of patches: concave spherical triangles, torus segments, and convex spherical patches. The spherical triangles can be ray casted straightforwardly, since they have no interior parts which have to be clipped. Krone et al. [2009] showed that the outer part of the torus, shown as wireframe in Figure 4.21, can be removed using a sphere-intersection test. The interior parts of the torus (shown as dotted lines in Figure 4.22) can be removed by intersection tests with two planes. Simply speaking, a pie slice is cut out of the torus ring. The spherical patches are the parts of the VdW spheres that are accessible by the rolling probe. The parts that are interior to the small circles traced out by the rolling probe while forming a torus have to be removed. These parts can be cut away using a clip plane through the small circle. However, there might be an inner remains which also has to be removed (shown as red patch in Figure 4.22). Using the RS for the SES computation, these problematic remains lie inside the RS. The RS is a closed triangle surface. The RS triangle edges connect atom centers that share a torus patch. The RS triangle faces span atoms which share a spherical triangle. Please refer to [Sanner et al., 1996; Krone et al., 2009] for more details.

PPLLs with CSG operations are used to remove the spherical remains. Please note that this application could also use α-shapes [Edelsbrunner and Mücke, 1994] or Power Cells [Varshney et al., 1994] to compute the SES and remove the inner remains of the spheres since they are essentially dual to the RS.

For the VdW and SAS, CSG is performed on visible objects only. However, it is also possible to render invisible objects (alpha equals zero) which will not contribute to the final image, but can be considered for the CSG operations. Hence, the triangles of the RS are rendered with alpha set to zero, as under certain conditions for concave structures, the RS might peek out of the atoms. Consequently, the fragments of the atom spheres inside the RS will be removed



◀ **Figure 4.22 —** Schematic representation of the SES and Reduced Surface of a molecule depicting the inner remains of a VdW sphere which have to be removed.

in the compositing stage without changing the algorithm, resulting in an artifact-free rendering of the protein surface.

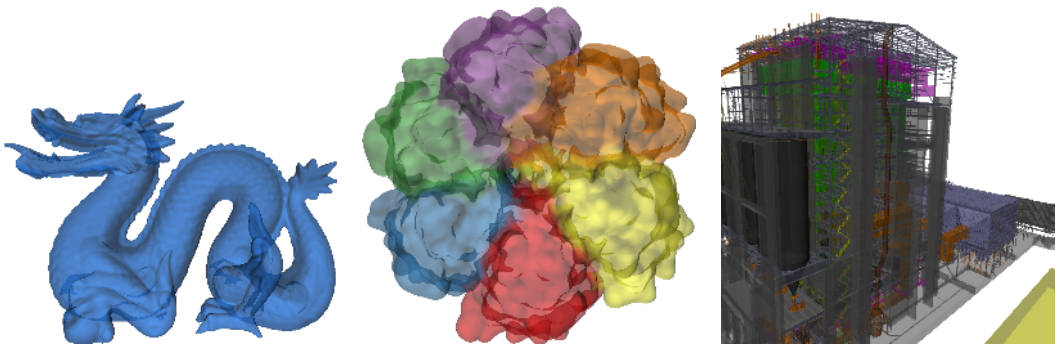
However, not all parts of the sphere which are located within the inner torus ring are also inside the RS. These fragments have to be removed as well. For that purpose, the torus is closed. That is, two planes are drawn to delimit the torus patch and the interior parts of the torus ring (green dotted lines in Figure 4.22) as invisible objects as well. Therefore, the CSG operation additionally will remove all fragments inside the torus object. The final rendering will only show the outer parts of the VdW spheres, the spherical triangles, and the outer torus patches. Thus, the transparent SES is displayed correctly with the PPLLs rendering algorithm.

Alternate SES Rendering

In addition to the transparent SES rendering explained above, there is another way as follows. As the problem is to determine whether a fragment of a VdW sphere may be drawn or not, one has to determine if this fragment is inside the surface or not. This can be achieved by rendering the SAS and projecting these fragments back to the original VdW spheres. A fragment from an atom is only visible if the corresponding SAS sphere fragment is visible. Although this method can be implemented straightforwardly, the problem here is the massive overhead of SAS rendering due to the much bigger spheres which produce many more fragments compared to the SES approach. As the evaluation shows, the SES is faster than the SAS rendering and, thus, the alternative SES generation was not inspected further.

Evaluation

As mentioned in beforehand, Parulek and Viola [2012] defined an implicit function for the SES. They render the SES using a sphere tracing of this function.



(a) Stanford Dragon (b) 1GKI protein (c) UNC power plant

▲ **Figure 4.23 —** PPLLs renderings used for evaluation (cf. Table 4.4).

Their performance evaluation for opaque surfaces shows similar frame rates on a NVIDIA Geforce GTX 480 as this approach achieves on a GTX 680 for transparent renderings. Since the sphere tracing is only executed until the first hit is found, more steps would be necessary for transparency rendering. This would drastically lower the performance of their method for large numbers of surface layers. Additionally, their visualization approach might have visible artifacts if the number of neighboring atoms is set too low [Parulek and Viola, 2012]. While this is only a minor distraction in opaque surfaces, the artifacts might become prominent in transparent renderings.

The performance of the PPLLs method is compared to DDP and, for reference, with pure (unsorted) OpenGL blending. DDP was chosen since it guarantees full visual quality whereas other accelerated DP methods, e.g. bucket depth peeling, may introduce visual artifacts. Here, the freely available implementation of DDP from the NVIDIA SDK is used. For benchmarking, an Intel i7 920 CPU with a NVIDIA Geforce GTX 580 (Fermi) and NVIDIA Geforce GTX 680 (Kepler), driver version 310.90 was used. The tests utilize freely available meshes and protein data sets from the Protein Data Bank [Berman et al., 2000].

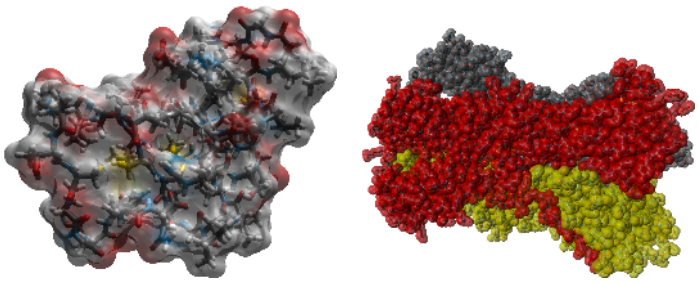
Table 4.4 shows the performance in frames per second of the PPLLs and per-pixel arrays compared to the DDP method for polygon meshes: a single Stanford Dragon (cf. Figure 4.23a), a protein surface mesh generated by VMD [Humphrey et al., 1996] (cf. Figure 4.23b) and the UNC power plant (cf. Figure 4.23c).

As DDP can only peel two depth layers per rendering pass, the framerate is very low compared to this implementation and to the OpenGL reference due to the number of render passes.

The difference between the list and the array approach is due to the different data structure and the additional operations necessary. Per-pixel arrays require

	Stanford Dragon	1GKI Protein	UNC power plant
Vertices	435 k	1,046 k	10,975 k
Faces	871 k	348 k	12,748 k
Fragments	1,157 k	2,227 k	14,765 k
Depth Layers	10	16	186
Screen Coverage	48.4 %	70.2 %	76.1 %
OpenGL	599.5	693.0	110.4
DDP	89.8	62.9	0.8
PPLLs	135.2	97.0	3.9
Per-Pixel Arrays	102.4	82.8	4.2

▲ **Table 4.4 —** Comparison of the PPLLs algorithm with other approaches for different polygon models using an NVIDIA GTX 680 on a 1024×1024 viewport. The numbers for the rendering methods are frames per second.



(a) SES of 1YV8 (b) VdW of 4ADJ

◀ **Figure 4.24 —** Molecule surfaces with internal ball-and-stick representation as used for PPLLs rendering evaluation detailed in Table 4.5.

two render passes and the calculation of the prefix sum. The benefit is a more structured memory layout as each pixels' fragments are stored consecutively. In the tests, this proofed to be beneficial for large models like the UNC power plant. By contrast, the list approach has a scattered memory layout.

Table 4.5 details the performance for the ray casted molecules (see Figure 4.24). The surface representation directly influences the fragment count, especially due to inner fragments and hull fragments needed for CSG. The SES generates about twice as many fragments as the VdW representation. The fragment count of the SAS is three to four times the count of VdW. This is also the approximate loss in the frame rate for the surface representations. A higher number of fragments slows down the performance when sorting and blending. The same effect as seen in Table 4.4 occurs here, too. Per-pixel arrays outperform PPLLs only for very high numbers of fragments. For small and medium fragment counts, the PPLLs are preferable performance-wise.

Another interesting fact is the performance difference between the GTX 680 and

Surface		VdW		SES		SAS	
Molecule		1YV8	4ADJ	1YV8	4ADJ	1YV8	4ADJ
Atoms		641	9720	641	9720	641	9720
Fragments		2.30 M	9.34 M	4.64 M	18.00 M	8.87 M	32.16 M
Screen Coverage		18.0 %	42.6 %	18.0 %	42.8 %	23.8 %	47.1 %
Depth Layers		44	92	117	188	128	234
GTX 580	Array	69.5	14.7	21.3	3.8	14.0	2.5
	Lists	85.4	16.6	31.0	6.2	14.7	1.8
GTX 680	Array	58.2	12.9	16.0	2.1	8.0	1.4
	Lists	76.6	13.9	22.1	4.2	10.3	1.3

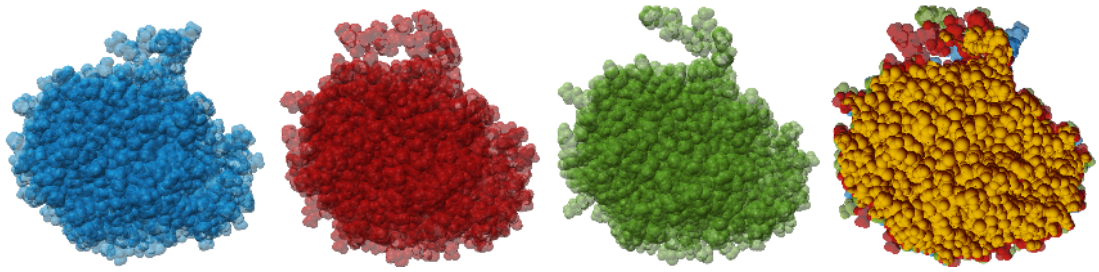
▲ **Table 4.5 —** Performance evaluation for different molecules and surface models rendered with the PPLLs framework. The numbers for the rendering methods are frames per second.

GTX 580 which might be due to changed caching or memory access strategies within the hardware design. On average for the selected benchmarks, the GTX 680 based on the newer Kepler architecture is about 40% slower than the older Fermi architecture of the GTX 580. This is not a PPLLs-related effect but, according to the experience gained throughout the tests, holds for other applications as well.

4.6.2 Visual Multi-Variant Analysis

This section details a multi-variant analysis prototype tool for molecular surface visualizations and multi-variant data analysis to investigate structural changes in molecular data. Its core component is the distributed rendering using PPLLs rendering. Each render node has the same data set but renders a different user-defined time step, effectively returning different states to the display node. The application is a visual analysis application [Kehrer and Hauser, 2013], delivering a tight integration of the visual mapping, visual interaction concepts, and computational analysis. It offers the possibility for the user to visually compare different variants of a molecular surface visualization and gives rudimentary tools to the user to explore the comparative visualization.

In many fields, a visual analysis of different variants of data is of interest for researchers. This can, for example, provide a succinct representation of interesting parts of simulated data that change over time. Therefore, multi-variant data analysis is proposed to investigate structural changes in molecular data. The data originates from molecular dynamics simulations and from the publicly accessible Protein Data Bank [Berman et al., 2000]. Either the data sets



(a) Variant 1 (b) Variant 2 (c) Variant 3 (d) Analysis View

▲ **Figure 4.25 —** (a) – (c) Three different variants taken from the *Lipase-2* simulation (see Table 4.6) of the 2VEO molecule rendered in the VdW representation. (d) The analysis view. The main body all three variants have in common is colored yellow, the moving arm is colored blue, red, and green, according to the variant. The presented method also highlights small changes clearly like the side chain (the little 'tail') in the lower left, which is very prominent in the green variant only.

can contain different conformations of a molecule or the structural changes might arise due to the simulated boundary conditions. The benefit of using a PPLLs approach is that every generated fragment is available for displaying on the display node. This means the transparency of the fragments can be modified after the scene has been generated on the render nodes and still deliver a correct visual impression on the display node.

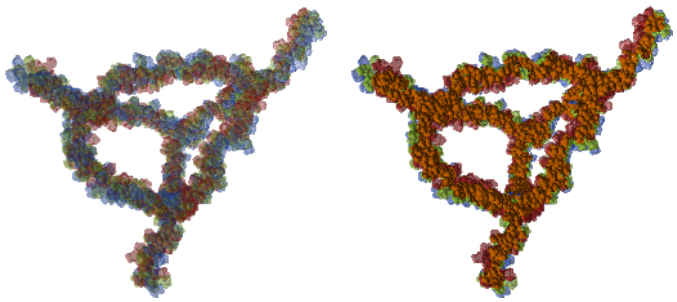
For visualizing molecule data sets, there exist a plethora of methods (cf. Section 4.6.1). This application uses the VdW representation but the described methods can be implemented with any other surface representation as well.

To render the proteins, GPU raycasting is used. In detail, the rendering using PPLLs works as follows: In the fragment shader, each sphere is raycasted and generates two fragments, one per intersection with the viewing ray. For both fragments, the depth as it is stored in the z-buffer, and the normal converted to θ and ϕ values from the spherical coordinate representation, is stored. When the fragments are generated, they are ordered by depth for the CSG (cf. Section 4.5.2) operation. During the CSG step, the union operation is performed to discard fragments that lie inside the molecule surface.

After the operation, only the surface fragments are left and are used for blending when displaying the molecule. This ensures minimal network usage and the display node only receives the fragments it actually needs for the visualization. Additionally, this implicitly sorts the resulting list on each node, reducing the work load for the merge step on the display node. Table 4.6 details the data

	Atom Count	Number of Fragments Rendered	Transferred	CSG Ratio
2JQW	260	1,222.0k	344.7k	28.2%
1YV8	641	2,055.4k	553.0k	26.9%
Lipase-1	4,622	5,479.3k	1,546.9k	28.2%
Hydrogel	4,822	1,465.3k	431.6k	29.5%
Lipase-2	6,626	7,147.7k	2,053.3k	28.7%
Pore	17,432	6,073.0k	1,751.5k	28.8%

▲ **Table 4.6 —** Number of fragments rendered, transferred, and the filter quote when using CSG on the render node to filter invisible/unused fragments.



(a) Standard Blending (b) Proposed Visualization

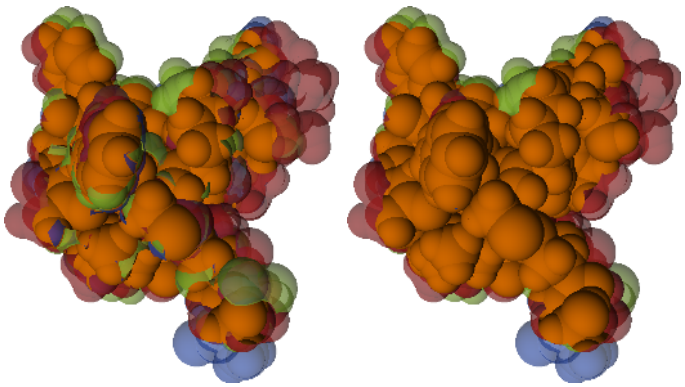
▶ **Figure 4.26 —** VdW representation of a hydrogel rendered in three different variants with PPLLs on three nodes, displayed using standard blending (a) and the multi-variant analysis application (b).

sets used for the benchmarks, and the number of rendered and transferred fragments. The filtering of the CSG operation reduces the data to be transferred down to about 28%.

As an example, Figure 4.25 depicts the view of the *Lipase-2* simulation on three render nodes (4.25a to 4.25c), each showing a different variant. The merged final view on the display node can be seen in Figure 4.25d. On the display node, each variant is assigned a separate color and a low alpha value is used where the variants differ from each other. Segments which are similar beyond a user-definable object space threshold in the variants are colorized in a special color and are displayed opaquely to reduce the visual clutter. For molecular dynamics, these parameters can be adjusted in Ångstrom to be meaningful to the domain experts.

Implementation Details

For each data buffer sent by the render nodes, the merging compares the depth and normal for each pixel to the data in the display node buffer. If normal and depth match by a certain user-defined threshold, a per-fragment counter



(a) Low Tolerance (b) High Tolerance

◀ **Figure 4.27 —** 2JQW protein rendered by three nodes with different merge settings. Each variant is rendered in a different color and the common parts are rendered in orange. (a): low tolerance visualization (neighbor area of 5×5 and epsilon of 0.03), and (b): high tolerance (neighbor area of 15×15 and epsilon of 0.08).

is incremented to determine the number of variants which comply with each other. As matching fragments are similar with respect to the thresholds, only one is used in the result buffer and the others are discarded. This reduces the work load for later merging operations and blending. When one merge step is completed, the input from the render node is taken over into the display node buffer and the buffer of the next render node is merged.

As the different versions of the data might have a small offset in image-space, a neighbor search is included which can be defined by the user. Here, a fragment is not only compared to the fragments at the exact same position but a small area in image-space is also searched for a fitting fragment to compensate position jittering of the different variants. Note that this is only executed for the first layer, that is, the front most fragments of the respective per-pixel lists are compared as the shared surface is rendered opaquely. Thus, it does not impact the overall performance much. Comparing the more layers and a rendering transparent shared surface would cost reasonable performance and result in much higher visual clutter. Figure 4.27 shows how the user defined tolerance settings impact the visualization.

After this step, the buffer can be displayed. Each linked list is iterated as described before. For fragments with a counter value higher than a certain user-defined threshold, the fragment is rendered opaque. This effectively renders the common parts of different variants opaque and the differences semi-transparently with a lower alpha value.

	Render Node[1]	Network Mem.[2]	Network Wait[1]	Display Node[1]	System Overall[1]	FPS
2JQW	32	45.0	539	64	635	1.6
Hydrogel	60	50.3	674	78	812	1.2
Lipase-2	287	149.3	1,537	171	1,995	0.5
Pore	289	130.9	1,576	136	2,001	0.5
2JQW	65	20.0	187	115	367	2.7
Hydrogel	94	25.5	337	126	557	1.8
Lipase-2	399	124.4	1,760	395	2,554	0.4
Pore	388	105.9	1,378	270	2,036	0.5

(a) GigaBit Ethernet

	Render Node[1]	Network Mem.[2]	Network Wait[1]	Display Node[1]	System Overall[1]	FPS
2JQW	31	45.0	162	71	233	4.3
Hydrogel	58	50.3	161	85	246	4.1
Lipase-2	334	149.3	700	210	910	1.1
Pore	295	130.9	494	141	635	1.6
2JQW	53	20.0	158	108	266	3.8
Hydrogel	100	25.5	252	134	386	2.6
Lipase-2	435	124.4	693	404	1,097	0.9
Pore	365	105.9	564	314	878	1.1

(b) Infiniband

▲ **Table 4.7 —** Performance evaluation of the multi-variant analysis application using four render clients with a GigaBit Ethernet (a) and IP over InfiniBand (b). The upper half shows the standard approach whereas the lower half uses LZO compression. Values shown are milliseconds for render and transfer times[1], and the transferred data[2] in MB.

Discussion

In the resulting images, the common parts of the molecules are clearly highlighted by the algorithm in contrast to the simple combined rendering (cf. Figure 4.26). The differing parts like the arm of the *Lipase-2* molecule in Figure 4.25 are still visible. Using different color for each version, they can clearly be distinguished.

To evaluate this approach, different data sets are used on four render clients, all

connected via a fully switched GigaBit Ethernet and InfiniBand. The display node and render clients were equipped with Intel Xeon X5660 CPUs and NVIDIA Quadro 6000, rendering a view port of 1024×786 pixels.

Table 4.7 shows the performance evaluation: Ethernet variant (cf. Table 4.7a) and InfiniBand variant (cf. Table 4.7b) with no compression (upper half) and with LZO compression (lower half). GFC was not used for this application, since the compression rate is very low. Although compression using LZO has a negative impact on the performance as seen in Table 4.8, it did compress the data significantly. Slow network architectures benefit from the LZO compression for small and mid-range fragment counts whereas InfiniBand is fast enough to transfer the plain uncompressed data for every case. For large fragment counts, the faster data transfer does not compensate for the effort of compressing and decompressing the amount of data. Although LZO was able to compress the data to about 44% to 83% of the original size, the resulting buffers still range from about 45 to 150 MB. For the overall performance, the compression gains about factor 1.5 for the small and medium data sets. The large data sets do not benefit from compression as it needs more time than the resulting smaller transfer can save. As the times show, the network waiting time is significant for the overall performance.

The *Overall* timings and FPS are measured on the display node. They include sending the request to the render nodes until the final image is blended and displayed on the screen.

Of course, this method also works on a single node. On the one hand, multiple instances of the same application can be run on one machine. Here, the GPU memory is the limit as each variant needs the same amount of memory for its buffers. On the other hand, the rendering tasks can be integrated into the display node. The required GPU memory for the PPLLs is about the same as one render node and the display nodes consume, regardless of how many variants are processed. However, each variant has to be rendered consecutively, removing the parallelism from the distributed approach. When a rendering is finished, it has to be merged into the display buffers to free the render buffers for the next variant.

The multi-variant analysis demonstrates a prototypical application for OIT in distributed rendering environments. The performance when requesting the data from the display node is about 1 to 4 FPS, depending on the data size and compression scheme. As the main performance loss is due to the network architecture, faster network architecture could drastically improve the performance. However, as the user inspects a static image displaying multiple variants at once, the performance is still reasonable.

As a result, when using PPLLs for distributed OIT rendering, the compression with LZO for standard GigaBit Ethernet networks is feasible, especially when a low but interactive frame rate caused by the network transfers is acceptable. For faster network architectures, e.g. Infiniband, the overall performance benefits more from the network than it gains by the compression step.

4.6.3 Distributed Mesh Rendering

A use case for distributed rendering of semi-transparent data is a simulation which runs in a distributed environment. For proving the concept, large static meshes like the power plant model and the MPI Informatics Building Model [Havran et al., 2009] are used. The full UNC power plant model, the index buffer and vertex buffer (position, normal and color) are about 564MB in size, resulting in about 12.7M triangles. For the MPI model, the data sums up to about 8.53GB and about 73.2M triangles, which is due to a different memory structure than the power plant uses.

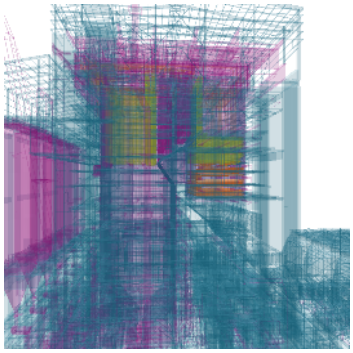
For a simulation or a dynamic scene it is important that every update generated by PPLLs is significantly smaller (180MB as seen in Table 4.1 for the power plant) than the raw data. The amount of transferred data mostly depends on the fragment count which is strongly related to the view port size, camera settings, or zoom level, and only indirectly depends on the model size and depth complexity.

Instead, the result of the simulation can be rendered locally on the respective node. Using PPLLs ensures that the final result is the correct and artifact-free visualization of the simulation output. Figure 4.28a shows the UNC power plant created by four render nodes and Figure 4.28b shows the MPI model created by eight render nodes. Each node renders a sub model, decomposed in object space with overlapping sections. The fragments are transferred to the display node. It composites the final image and shows it for user interaction.

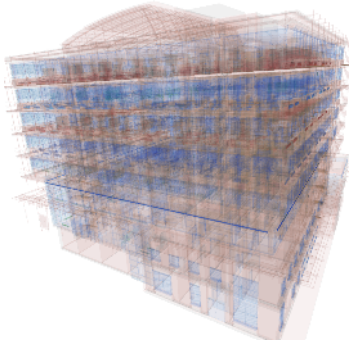
For displaying, a simple application which connects to the render nodes and sends the camera updates is used. Each render node uses the this information to render its part of the scene into its PPLLs buffers. The buffers are transferred back to the display application. Here, all received buffers are merged into one buffer. Each PPLL is then sorted by depth, the fragments are composited, and finally blended onto the screen.

Discussion

Four render nodes and one display node are used for evaluation of how compression impacts the performance. The machines were using an Intel Xeon



(a) UNC power plant



(b) MPI building

◀ **Figure 4.28 —** The UNC power plant and MPI Informatics Building rendered using multiple render nodes. The different colors indicate the parts rendered by each render node.

	Render[1]	Network[1]	Display[1]	Mem.[2]	FPS
FBO	76.8	243.2	329.3	32.0	3.1
PPLL	344.3	2,071.0	375.1	241.3	0.4
GFC	270.5	2,042.6	424.7	238.2	0.4
LZO	962.2	1,868.9	483.8	138.9	0.3

▲ **Table 4.8 —** Elapsed average time[1] in milliseconds on the render node, for network transfer, and compositing on the display node. The scene is a completely opaque view (FBO) for reference and a fully semi-transparent view with the PPLLs variant for the UNC power plant model. Additionally, the GFC and LZO compression algorithms were tested on the PPLLs data. Transferred memory[2] is shown in MB.

E5620 processor, connected via GigaBit Ethernet, and using an NVIDIA Geforce GTX 480 graphics card to render a viewport of 1024×1024 pixels.

The 12.7M triangles of the power plant are split into four different sub-models of equal input size. This emulates a simulation where each node has about the same work load to simulate and to render.

Table 4.8 shows the timings for the render nodes, the transfer, and the display nodes. Note that the communication between host and GPU is included here, as well as the time spent compressing the data. The frame rate is low for the transparent case due to bandwidth limitations, that is, it is barely possible to render, transfer, and display one frame in approximately 2.5 seconds. Using the GFC compression hardly improves the network performance, gaining no overall speed-up. For the LZO compression, a significant improvement in the network transfer times is achieved but the computational overhead negates this gain completely.

Nodes	Render[1]	Network[1]	Display[1]	Mem.[2]	FPS
2	120.9	708.8	98.6	213.5	1.3
4	59.6	758.5	114.3	221.5	1.2
6	40.7	644.0	116.7	229.5	1.4
8	31.1	582.8	110.0	237.5	1.4

▲ **Table 4.9 —** Evaluation of the MPI model using a different number of render nodes. Average time[1] in milliseconds on a render node, for the network transfer and displaying operation. The amount of memory[2] transferred is shown in MB.

Table 4.9 shows the evaluation of the MPI model rendering for four setups with different number of render nodes. The benchmarking computers were equipped with Intel Xeon X5650 CPUs and NVIDIA Quadro6000 GPUs, and connected via Infiniband. The view port is 1024×1024 pixels. As the timings of the render nodes show, the computational effort for rendering decreases linearly with the number of render nodes. But as the total amount of fragment data is the same for all cases – the complete model has to be transferred and merged – the performance gain is very low.

Comparing the distributed rendering approach from Table 4.8 to the local approach from Table 4.1 and Table 4.4, the enormous data transfer drastically reduces the performance. Although the LZO compression significantly reduces the network load, the computational overhead is a problem. Due to the large data that needs to be transferred for the complete scene, distributed OIT is very costly. Comparison against local rendering is, however, beside the point in context of a distributed simulation, since one cannot afford the gather operation for every completed simulation step (see above).

For a large model like the MPI building which does not fit in the memory of one graphics card, the evaluation has shown that the performance does not benefit from a larger number of rendering nodes. Here, the network performance is the main bottleneck of the system.

4.7 Summary

This chapter has demonstrated advanced PPLLs techniques for applications. Implementing CSG made it possible to adapt existing molecular surface visualizations to semi-transparent surfaces. The general approach for storing the fragments and communicating them between the steps made it easy to extend the approach to be used in distributed rendering environments.

The rendering approach delivers a reasonable performance for the presented applications. For the single node renderings, the bottle neck is clearly the graphics card. Although there was a noticeable negative performance difference between the NVIDIA GeForce GTX 580 and the newer NVIDIA Geforce GTX 680, it can be assumed that the frame rates will increase again with new and upcoming hardware architectures.

However, the bottleneck for distributed rendering is the GPU-to-CPU communication and network. The benchmarks have shown that even single-threaded CPU compression can make a serious improvement on the overall performance for slow networks. A alternative approach has been presented as well, reducing the amount of memory transferred for each frame by exploiting the fact that the display node can transform the fragments, effectively mirroring the operations on the render node. The evaluation has shown that the computational effort for the operations are higher but the network load could be reduced without negative influence on the image quality.

4.7.1 Outlook

For future work, the PPLLs approach can be combined with post-processing effects like ambient occlusion or cel shading with transparency which might help to understand complex visualizations. As all fragments are available, including additional information like the normal vector, a mixed realization of screen-space ambient occlusion and object-space ambient occlusion could be implemented. The PPLLs buffer does not hold all fragments in the scene, but from a given view. Thus, the approach could include more information than the standard screen-space ambient occlusion that only has the first layer of fragments available.

Furthermore, to improve the strategy for updating the current frame on the display node, for example by sub-sampling or interpolation, could be worth looking into. More and upcoming real-time compression algorithms are to be evaluated for using them with per-pixel linked-lists, reducing the size of the buffers, thereby further optimizing the GPU-to-host communication and network transfers.

A third topic is to extend the PPLLs into the time domain. As the approach is able to deliver interactive frame rates, using animated scenes is trivial. More interesting, however, is to use it with large data sets, e.g. from volume visualization. This requires a far more integrated approach, combining the volume data with the PPLLs. From a given flow field or gradient in the time-domain, the direction of a given sample could be extracted. To prepare the next frame on the display node, the PPLL element representing that sample could be moved to the

new position – similar to the transformation in the memory saving two-sided distributed rendering approach (cf. Section 4.4.3). Annotating the volume voxels with a direction and velocity, the voxels could be stored into the correct neighboring PPLL. Thus, the PPLLs algorithm would provide interpolation for data sets with multiple time steps. This would allow for fluent updates and a more interactive experience. In the background, the render nodes could produce the correct buffers and send them to the display node. Once the correct buffers are available, they can be combined with the predicted buffer currently shown. Therefore, the frame is completed and can be displayed to the user before the next frame is in preparation.

The distributed rendering approach can be applied to volume visualization as well. Very large volumes could be distributed among the participating render servers and the output could be sent to the display node(s), just as described for the normal rendering. To extend the two-sided approach, an image analysis algorithm can calculate the quality of the image on the display node and on the render node and decide whether to send a differential update or a full update. This would keep the traffic as low as possible while still maintaining a high image quality even for complex movements, animations, or a long series of view transformations.

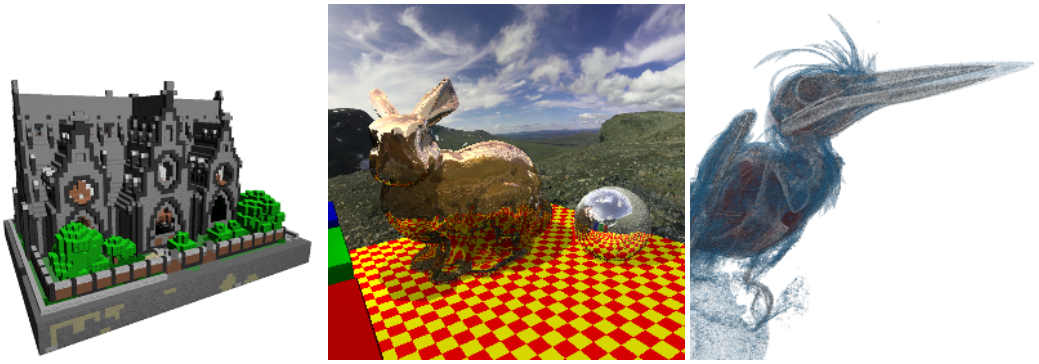
4.7.2 Conclusion

PPLLs are a general and flexible rendering algorithm. The strength is its use for artifact-free transparency rendering. It can be used with all sorts of fragment generation, let it be rasterized vertex data, implicit surfaces, or volumetric data. The algorithm delivers interactive performance and can easily be integrated for use with existing code. It also enables complex operations like CSG without large effort.

Due to the flexibility and the buffer interface, the PPLLs approach can be used for local rendering, distributed rendering, or remote rendering. Three different prototypical applications have demonstrated that the effectiveness of the algorithm. As a result, PPLLs enable OIT effects and do perform an interactively, thus delivering a great visual quality suited for large data visualization, even on distributed rendering systems.

CHAPTER 5

Generalized Rendering using Per-Voxel Linked Lists



(a) Polygon brick house (b) Raytracing scene (c) CT-scanned Kingfisher

▲ **Figure 5.1 —** PVLLs renderings of a geometry model (a), a raytraced scene containing reflection, refraction, and shadows (b), and a sparse volume visualization of a kingfisher CT scan (c).

The previous chapter focused on transparency rendering of distributed, decomposed objects and mesh objects too large for the graphics card memory. This chapter presents research on rendering and storing sparse volume data resulting in the Per-Voxel Linked Lists (PVLLs) approach – a spatial data structure based on linked voxels for volume data sets which might not fit onto current graphics cards at once. It extends the concept of PPLLs by Yang et al. [2010] for using it

in combination with voxelization, voxel-based rendering, and the visualization of sparse volume data.

Nowadays, data sets obtained from measurements, modeling, simulations, or other sources grow larger and larger in size. Data sources include, among others, image acquisition modalities, like CT or laser scanners, and large-scale models, e.g., in the field of geostatistics, being both of particular interest in the context of this paper. Regardless of their origin, these large data sets have to be processed and visualized pushing the limits of available hardware. In most cases, however, the original raw data contains lots of information which is of no interest in the subsequent processing or visualization steps. This uninteresting data can be filtered out in a pre-processing step for example by applying a transfer function to volumetric data sets or applying simple threshold filters. Then again, the data might already be sparse.

Voxelization is a very generic approach of sampling arbitrary three-dimensional objects, like meshes, unstructured data, or implicit object representations, into a discrete volumetric representation based on a regular grid. A voxel cell of this grid acts like a bin where all samples within that location are put into. Mesh objects can be voxelized by intersecting the voxel position with the triangles and implicit representations like spheres are voxelized by evaluating samples at the voxel position [Falk et al., 2013]. In addition, continuous volumetric data can be sampled as well. Figure 5.1 shows three exemplary scenarios where the presented method can be utilized. From left to right these include voxelization of mesh-based objects, global effects rendering with shadows, reflection as well as refraction, and sparse volume representations for data from CT scans.

When using the linked lists with the standard A-buffer approach [Carpenter, 1984], the data structures are view-dependent and, thus, have to be updated each frame. With the presented approach this is no longer necessary as any arbitrary view can be reconstructed. Therefore, the buffers do not need to be updated for each frame on static scenes and, as all fragments are available, global rendering effects like refraction and reflection can be realized easily. In this approach, the fragments of a scene or data set are stored in a single buffer in a volume like fashion, however avoiding the storage of empty spaces. This makes the approach suitable for sparse volume representation and for volume rendering as empty parts are omitted and do not waste graphic memory.

The presented method is able to render voxelized scenes – including global illumination effects – with interactive frame rates. For the sparse volume data, the proposed algorithm is able to reduce the required memory footprint in comparison to standard volume rendering methods, allowing the inspection of high resolution sparse volumes even on low-end graphics devices.

5.1 Related Work

The presented approach extends the PPLLs approach so it does not only contain the information of a particular view but comprises the entire scene.

5.1.1 Voxelization

The method of voxelization has long been used for voxel-based graphics systems [Kaufman and Shimony, 1987] and to speed up ray tracing [Yagel et al., 1992]. Karabassi et al. [1999] utilize the depth buffer of the GPU to voxelize certain non-convex objects and their surfaces. In contrast to the PVLLs approach, features like cavities might be missed since only six depth images are used, which are generated along the coordinate axes. In 2000, Fang and Liao [2000] presented an voxelization approach for CSG models where the models are evaluated for multiple slices along the view direction. Recent work by Eisemann and Décoret [2008] computes the voxelization of solid polygonal objects using the GPU in a single rendering pass. GigaVoxels [Crassin, 2011] is a voxel-based framework real-time rendering of large and highly detailed volumetric scenes. These works are specialized to voxelize volumes or geometry objects while the approach presented here can voxelize and visualize both data types. Kämpe et al. [2013] evaluated directed acyclic graphs instead of octrees to store the voxels in hierarchical levels, but only for polygonal models.

5.1.2 Sparse Volume Rendering

While there exist various methods for volume rendering, this chapter refers only to volume ray casting without loss of generality. Volume ray casting was first presented in 1988 [Drebin et al., 1988; Levoy, 1988; Sabella, 1988] and bears a high resemblance to ray tracing without secondary rays. In recent GPU-based approaches, the volumetric data is stored in a 3D texture and the volume rendering is performed within a single pass [Stegmaier et al., 2005b]. For large, sparse data sets the memory requirements might exceed the amount of available graphics memory and more advanced storage schemes have to be used. Museth [2013] presents VDB, a framework for sparse, time-varying volumetric data that is discretized on a grid. This kind of data is used for animations and special effects in movies. Teitzel et al. [1999] presented an approach to render sparse grid data by interpolating the sparse grid and in fact turning it into a volume. Kähler et al. [2003] use adaptive hierarchical methods to render sparse volumes, effectively partitioning the whole volume into smaller volumes which are then used for volume ray casting, similar to the two-stage storage of the PVLLs approach but using volumes instead of voxel lists. Gobbetti et al. [2008]

present an alternative single-pass approach for rendering out-of-core scalar volumes. For additional state of the art, refer to Balsa Rodríguez et al. [2014] and Beyer et al. [2014]. Other methods for storing sparse data are the *compressed row storage* (CRS) or *compressed sparse row* (CSR) patterns [Koza et al., 2014]. Here, sparse matrices are compressed by removing the entries which contain zeros. The PVLLs algorithm adopts these notions of sparse matrices and extends it to 3D by relying on the linked lists. Instead of storing all volume data, the volume data is pre-classified with a given transfer function and only non-zero voxels are stored. This allows for a smaller memory footprint of the volume, in particular when representing sparse volumes.

Nießner et al. [2010] use *Layered Depth Images* (LDIs introduced by Shade et al. [1998]), a concept similar to the PVLLs approach. Instead of using three orthographic projections and linked lists of voxels, they use n orthographic LDIs to store the scene of geometric data. Frey et al. [2013] extended the LDI approach to *Volumetric Depth Images* generating proxy data which displays volume data sets. Bürger et al. [2010] propose *Orthogonal Frame Buffers* as extension to LDIs and perform surface operations like recoloring or particle flow on rendered meshes. Reichl et al. [2012] use a rasterization and ray tracing hybrid technology to render meshes. In contrast and as extensions to these approaches, PVLLs can render meshes, volumes, and implicit surfaces and internally stores the voxels in linked lists. It displays arbitrary rasterized data and uses ray tracing for rendering and visualizing global lighting effects.

5.2 From Pixel to Voxel: Per-Voxel Linked Lists

In the standard rendering approach with a depth buffer, only the front-most fragments are visible. For rendering semitransparent scenes, the fragments have to be drawn either in a specific order to allow for appropriate compositing or by approximating the blending function [Salvi et al., 2011]. Rendering such scenes in a single rendering pass is possible when e.g. utilizing the PPLLs approach. After all fragments have been gathered in the PPLLs, the lists are sorted according to the fragment depths. The final image is obtained by blending the sorted fragments based on their opacity. Since the A-buffer is created for a particular camera setting, its content has to be updated as soon as the view point is changed. In addition, the A-buffer also reflects the projection mode of the camera, i.e. perspective or orthographic projection. Thus, the contents of the A-buffer describe only the parts of the scene that are inside the view frustum.

In contrast to the classical A-buffer approach, orthographic projections are used along the three coordinate axes to capture the scene in an entirely view-

independent manner. The concept of the PPLLs is extended to PVLLs, allowing to combine the results of the three individual views in a single storage buffer – the main PVLLs buffer. This buffer makes it possible to store the voxelized scene in a memory-efficient way and to reconstruct it in the subsequent rendering step. Since the buffer is not view-dependent it is created once in the beginning and has to be updated only when the scene geometry changes.

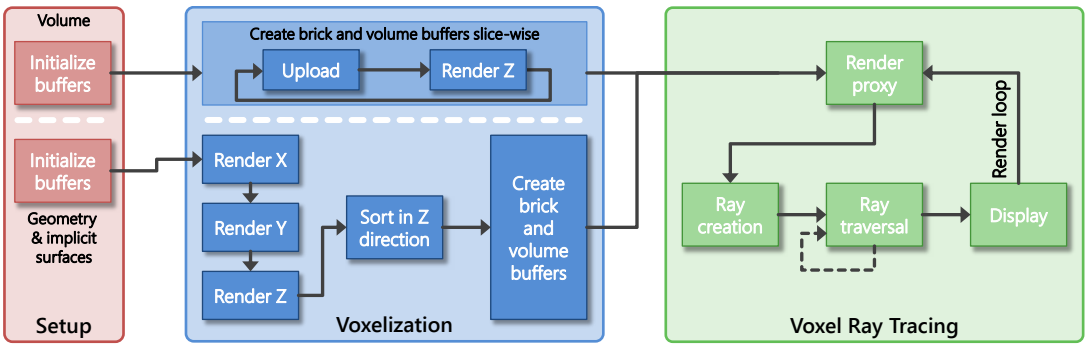
From a conceptual point of view, the *bounding volume* is defined with a given volume resolution to comprise the scene similar to an axis-aligned bounding box. One axis of the bounding volume is chosen to be the predominant axis for the traversal of the voxel-linked lists – without loss of generality let it be the z-axis. Each fragment inserted into the bounding volume is transferred by a simple coordinate transformation, using the swizzle operators. As for each of the three views from the main axis, xy is always the screen and z is always the depth component, the transformation ensures the correct voxel position in the volume. The bounding volume itself is subdivided into multiple *bricks*. Each brick contains a number of voxels depending on the brick resolution. For reasons of simplicity, let the brick resolution be $1 \times 1 \times 1$ unless otherwise noted. The voxels represent spatially extended pixels which can either be filled or empty. By applying the linked-list approach, only filled voxels are stored.

In the following, PVLLs algorithm is described. The underlying data structure for the individual bricks, implementation details, and proposed optimizations are presented in Section 5.3. Section 5.4 focuses on the application of the PVLLs in the context of sparse volume ray casting and the utilization for global ray tracing effects.

5.2.1 Voxelization Algorithm

The PVLL algorithm consists of three stages: the setup stage, the voxelization, and the rendering stage. Figure 5.2 depicts the algorithm and shows the connections between the three stages. During the setup, the necessary buffers are created and initialized. For both geometric and volumetric data a single buffer, i.e. one A-buffer based on PVLLs, is sufficient.

The voxelization of geometric objects, i.e. mesh-based models, takes place in three steps. First, an orthographic projection of the entire scene is rendered for each of the coordinate axes. During the rendering of one view, all fragments are transformed into a common coordinate system to match the predominant axis of the buffer – the growth direction of the buffer. Each fragment is then inserted into the respective per-voxel list. If a voxel entry does not exist for that position, the voxel is created and added to the A-buffer and a pointer to the fragment data is added. In case the voxel entry already exists, the fragment data



▲ **Figure 5.2 —** The PVLLs algorithm. Geometry and implicit surfaces are voxelized for each of the three coordinate axes whereas the per-voxel buffer is set up in one pass for sparse volumes. The final rendering stage is identical.

is appended to the list of fragments of the voxel. Upon insertion, the attributes of the fragment, like position and normal, are transformed to object space to maintain consistency between the three coordinate views.

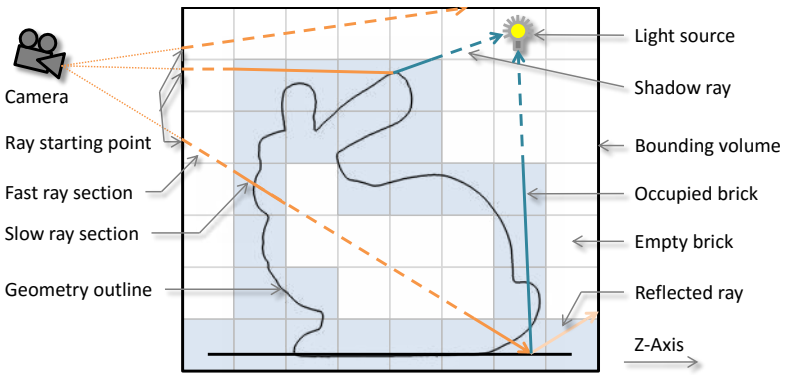
By using three orthogonal cameras/views, the algorithm is able to rasterize the entire scene. This is due to the fact that if an object is not visible in one view, e.g. a plane parallel to the viewing direction yielding no fragments for this particular direction, will be visible in one of the other views. Thus, the scene is fully rasterized and the final voxel volume will include each fragment.

After rendering the scene into the main buffer, the buffer contents have to be sorted. Similar to the PPLLs approach each list is then sorted according to depth independently. In the last step of geometry voxelization, the brick states are updated. A brick is marked as occupied if it contains at least one filled voxel. In case the brick resolution is equal to $1 \times 1 \times 1$, i.e. a brick consists only of a single voxel, this step is skipped. The brick occupancy is used during the rendering stage to support empty-space skipping. Please note also that the buffer holds only individual non-empty voxels and information on empty space is encoded indirectly. Figure 5.3 shows the Stanford Lucy model in different voxelization domains, clearly showing the voxel structure for the smaller domain sizes.

Besides rasterized polygons, the voxelization can also process implicit object representations that are typically used for ray-casting, e.g. to visualize particle data sets. In this prototype, the ray casting of spheres is used as an example, but the approach can be extended to other shapes as well. The spheres are raycasted and for each generated fragment, the sphere data (center and radius) is embedded into the voxel data. This allows the sphere to be raycasted again when the traversal ray enters the sphere voxel, resulting in a pixel-accurate image. Falk et al. [2013] presented this approach in a similar way before.

(a) 64^3 (b) 128^3 (c) 256^3 (d) 512^3 (e) 1024^3 (f) 2048^3

▲ **Figure 5.3 —** The Stanford Lucy rendered with different voxelization parameters. While the voxelization is clearly visible for the low values, the 2048^3 volume exceeds the framebuffer resolution of 1024×1024.



▲ **Figure 5.4 —** Schematic two-dimensional view of the bounding volume and the ray tracing process of a translucent bunny on a mirroring surface. Exemplary rays (orange) are traced from the camera through the voxelized geometry (black outline) including secondary shadow rays (teal) and a reflection on the ground surface (light orange). Dashed lines denote fast grid traversal for skipping empty bricks.

Volume data, in contrast to meshes, can be transformed into a buffer without explicit voxelization. The volume is processed along the predominant axis in slices that are one voxel thick to keep the memory requirements down. The slices are uploaded into GPU memory and evaluated one after another. On the GPU, a pre-classification of the volume slice is performed by applying the transfer function for alpha thresholding to discard transparent voxels. The remaining voxels are appended to the respective voxel lists. When the user changes the transfer function, the pre-classification is computed within 3-5 seconds, not hindering the interactive experience too much. Since the volume is

sliced along the same axis as used for the PVLLs buffer, the contents of the linked lists are automatically sorted according to depth by definition. Again, only non-empty regions are preserved and stored in the buffer, thus enabling support for volumes which would otherwise not fit into GPU memory. In addition to the original data value, local gradient is also stored to be used for illumination computations. In the last step, the brick states are updated.

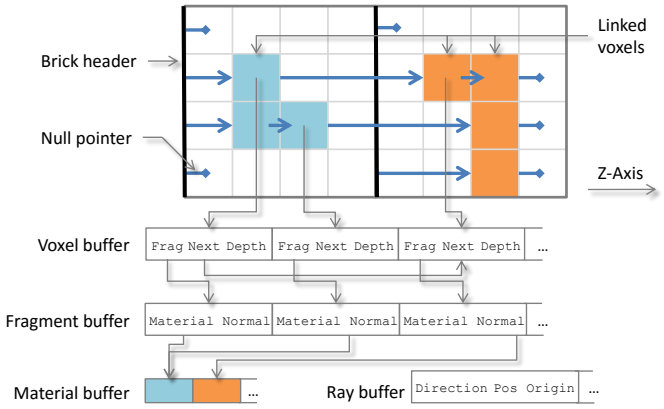
5.2.2 Final Image Rendering

The final rendering is obtained by voxel ray tracing within the bounding volume. First, initial rays are created for each pixel on the screen with a starting point on the bounding volume and their direction determined by on the camera position. The camera can be positioned arbitrarily and even be situated inside the bounding volume since the entire scene was captured with the buffer. Rays are traversed by projecting the direction vector onto the boundary of the nearest next voxel.

Similarly, empty-space skipping is performed by projecting the direction vector onto the next non-empty brick. Within a brick, generic ray-voxel traversal with a slight modification is employed. Finding the next voxel along the principal direction of the VoxLink space is straight forward, i.e. the next list element. To find a voxel in a neighboring list, at first, the the respective list has to be identified and then it is traversed from its beginning until the algorithm arrives at the correct depth. In case there is no stored information available, the voxel is empty and the traversal is continued. Otherwise, the color value is computed for contributing fragments using the stored normal.

During ray traversal, the voxels of non-empty bricks are inspected and a color value is computed for contributing fragments using the stored normal vector and standard per-pixel shading algorithms. For volume rendering, the normal is derived from the local gradient of the actual volume and also stored. The rays traverse the voxel volume cell by cell, following the exact direction vector. For each step, the current position is forwarded to the next cell boundary (if the brick contains voxels) or to the next brick (if the current brick is empty). The color value is then blended with the respective screen pixel utilizing front-to-back blending. Secondary rays can be generated and traversed as well to achieve shadowing, reflection, or refraction. Early-ray termination is carried out once a ray leaves the bounding volume or the accumulated opacity exceeds a user-specified threshold.

Figure 5.4 shows a scene containing the outline of a semi-transparent bunny sitting on top of a reflective surface with exemplary primary rays (orange) and secondary rays (teal). A lookup in the brick volume determines whether



▶ **Figure 5.5 —** Data structure and auxiliary buffers of two bricks, each holding 4×4 voxels and having a brick header pointing to the first non-empty voxel in that row or nothing. Colored voxels contain only data needed for the ray traversal and are linked similar to the PPLLs approach.

the local area is completely empty (white cells) or whether there are some populated voxels (blue cells). In the first case, the ray can skip all voxels within that brick and fast-forward to the next brick (indicated by dashed lines). Otherwise, each voxel within this brick has to be traversed and its contribution is computed.

5.3 Implementation Details

The OpenGL rendering pipeline and GLSL shaders are used for the voxelization and the subsequent ray tracing. The buffer is created in a similar way as the A-buffer with the linked-list implementation except for the transformation of fragments to the predominant axis before inserting them into the respective lists and the fact that the scene is rendered three times to capture all features. The data layout of the lists are adjusted to benefit from cache coherency during ray tracing. The general layout of the data structures is detailed below and followed by some considerations regarding optimizations.

5.3.1 Data Structures

The proposed data structure is quite similar to those used in PPLLs A-buffer implementations and is depicted in Figure 5.5. Blue arrows denote the pointers and gray connectors show the layout of the data and how the different buffers are connected via indices. The *global header*, a 2D buffer, covers the front face of the bounding volume, which is perpendicular to the predominant axis. In the original PPLLs concept, this corresponds to the A-buffer covering the viewport. The indices stored in the global header point to the list of voxels for that particular position. The voxel lists are stored in the *voxel buffer* as linked lists

and each list element consists of a fragment index, the depth of the fragment in terms of the predominant axis, and a pointer to the next list element.

The fragment index in turn points to an element in the *fragment buffer*, which holds a material ID and the normal to be used for deferred shading. During the rendering phase, multiple fragments might be created for a specific voxel. These fragments get merged by averaging their values during the sorting step. The material ID is used for a lookup in the *material buffer* that holds all the different materials. It stores the color, reflection, and refraction coefficients.

The global header is mainly used during the voxelization stage. To support empty-space skipping, the bounding volume of the scene is subdivided into multiple bricks of size n (cf. Section 5.2). Each brick has its unique *brick header* representing n^2 entry indices. In principle, brick headers are identical to the global header but they posses additional information on whether the entire brick is empty. As can easily be seen, the combined brick headers of the first layer represent exactly the same information as the global header. Thus, the global header is not necessary for rendering.

For sparse volumes a bounding volume resolution identical to the original volume is used for simplicity. With this restriction, a voxel of a brick can only contain a single data value and, thus, the data is stored directly in the voxel buffer without the need of an additional fragment buffer. Additionally, the normal derived from the volume density is stored in 16 bit polar coordinate form. The actual color contribution of each voxel is calculated during rendering by applying the transfer function to the data value as it is done in regular volume rendering.

5.3.2 Optimizations

There exist a couple of major starting points for optimizing the approach outlined above. Although the optimizations proved to be beneficial in improving the performance, some some of the ideas and their identified problems are discussed in more detail in the following.

Data Layout

The PVLLs data layout separates the voxel cells and the actual content so the stored fragment information like color or normal is only accessed if it is needed and not every time the linked list is traversed. Additionally, shared information like the material is only stored once and each fragment has a link to its material where the information can be retrieved from in the appropriate stage.

Empty-Space Skipping

The introduction of the bricks drastically improved the rendering performance by enabling empty-space skipping. An evaluation showed that the optimal brick size n varies with the bounding volume resolution N, but a ratio of $N/n = 64$ gave the best results in the tests. If the region is empty, it can be traversed in one single step, skipping the look-up for 4^3, 8^3, or 16^3 voxels at a domain size of 256, 512, or 1024, respectively (see Figure 5.10).

GLSL Shader Optimizations

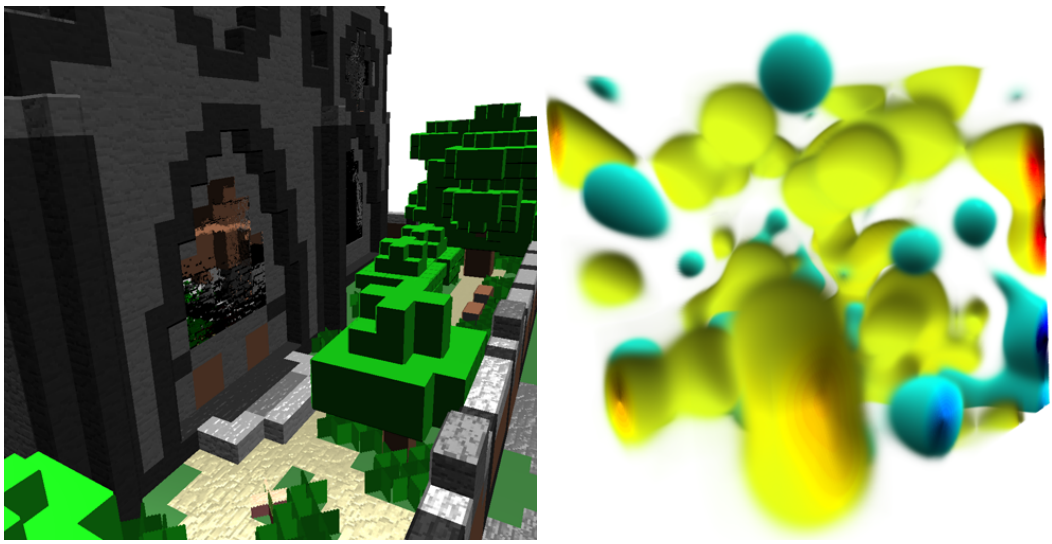
Although GPU architectures are becoming more and more generalized nowadays, they are still sensitive to branching which cannot be entirely avoided for ray traversals. Rewriting `while`-loops into `for`-loops with constant upper bounds, reducing the branching to a minimum, and converting boolean expressions into formulas helped to improve the overall performance. However, care has to be taken when using upper bounds for loops as computations might stop prematurely, potentially resulting in artifacts or unfinished images.

5.4 Voxel Ray Tracing

After the buffer has been created by either voxelizing geometric objects or embedding sparse volume data, the final image is obtained by means of ray tracing. Two exemplary renderings are depicted in Figure 5.6. The house (Figure 5.6a) is built from geometry blocks, voxelized, and rendered with refraction and reflection on the windows, and the other data set (Figure 5.6b) is a sparse volume from the field of geostatistics. Although the traversal and evaluation of the buffer, i.e. the rendering, is basically the same for both sparse volume data and voxelized geometry, the following sections point out some data-dependent peculiarities.

5.4.1 Sparse Volume Rendering

One potential application for the proposed PVLLs is the rendering of sparse volume data. Depending on the volume data and the associated transfer function, typically only a low percentage of voxels is of interest (cf. Table 5.3). All other density values are not of specific interest and can, thus, be omitted safely. More specifically, they consume precious memory although their content is never visualized. With the PVLLs, it is possible to store the remaining and interesting data in a memory-efficient and sparse data structure.



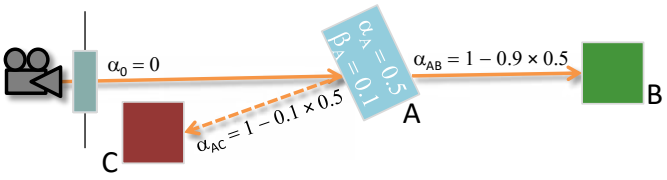
(a) (b)

▲ **Figure 5.6 —** Example scenes for voxel-based ray tracing. (a) detailed view of a house built from blocks showing refractions in the window. (b) volume rendering of the Kronecker 2 data set from geostatistics.

The rendering of the sparse volume data itself suits fine with the rendering algorithm proposed in Section 5.2.2. During voxel ray tracing the density values stored for the occupied voxels are mapped to colors by applying a transfer function identical to regular volume ray casting [Stegmaier et al., 2005b]. The main difference is, however, that changes of the transfer function are somewhat limited in this approach with respect to interactive rendering. Since the approach relies on a pre-segmentation of the volume to generate a sparse volume, the transfer function can be adjusted interactively for all densities occurring in the sparse volume. But voxels which were initially discarded during voxelization cannot be made visible without a re-computation of the sparse volume. This allows the user to alter the color and opacity of existing voxels only. However, redoing the pre-classification only takes 3-5 seconds, thus it is not hindering the interactive experience.

5.4.2 Global Effects Rendering

With the availability of an in-memory representation of the whole scene additional global effects are feasible. Therefore, support for secondary rays allowing for shadows, refraction, and reflection was added. The original algorithm described in Section 5.2.2 was extended by performing the ray traversal in a loop (cf. Figure 5.2, dashed line) and adding an additional *ray buffer* for dynamically



▲ **Figure 5.7 —** Correct color blending for split rays. The ray is split in voxel A and directly traversed to voxel B while the ray to voxel C is stored for the next iteration. Voxel A is a refractive material with transparency $\alpha = 0.5$ and also reflects light with coefficient $\beta = 0.1$.

adding and storing rays generated during the rendering phase. Besides the ray origin and direction, the ray buffer holds the opacity accumulated so far and the pixel position of the initial camera ray. The explicitly stored pixel position allows us to process primary and secondary rays independent of the final pixel position while still being able to contribute to the correct pixel. To account for correct blending of the ray results, the accumulated opacity is propagated with each split as well.

Figure 5.7 depicts an opacity-value transportation scenario where voxel A is refractive as well as reflective. The refraction ray hits voxel B and the reflection ray hits C. The standard color blending equation for front-to-back compositing $C'_i = C'_{i-1} + (1 - \alpha'_{i-1})\alpha_{\text{mat}} C_{\text{mat}}$ is not applicable when using the iterative ray traversal as the rays can split on reflective and refractive voxels. Adding the color of the delayed ray to the final image with said equation does not yield correct results. However, if the contribution of the rays is split and the alpha value propagated with them, one can rewrite the equation yielding the correct color for the example as follows

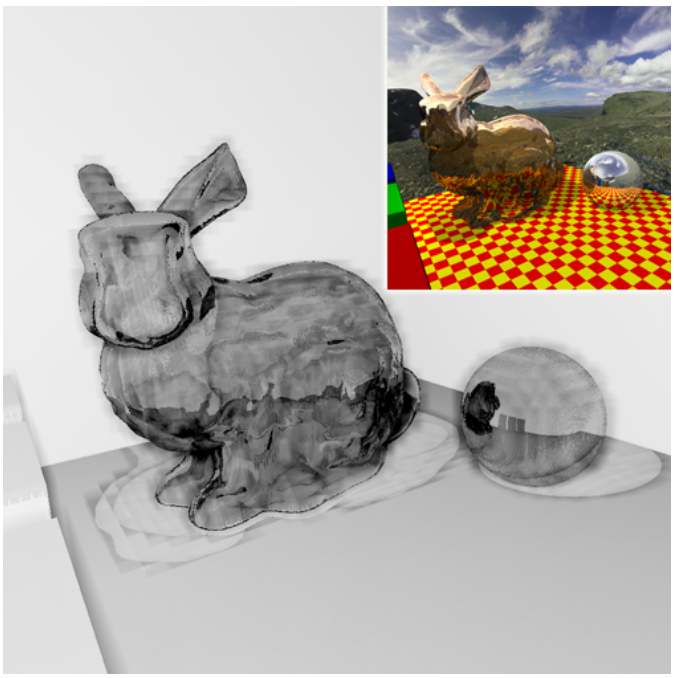
$$C = (1 - \alpha_0)\alpha_A C_A + C_B(1 - \alpha_{AB}) + C_C(1 - \alpha_{AC}),$$

with

$$\begin{aligned} \alpha_{AB} &= 1 - (1 - \beta_A)(1 - \alpha'_A), \\ \alpha_{AC} &= 1 - \beta_A(1 - \alpha'_A), \\ \alpha'_A &= \alpha_0 + (1 - \alpha_0)\alpha_A = 0.5, \end{aligned}$$

and β_A denoting the reflection coefficient of A. This principle can be applied at each ray split by simply propagating the computed opacity at that point and, thus, achieve proper compositing of the final image.

The ray traversal step of the algorithm is adjusted for the iterative ray storage to take care of potential ray splits that are depending on the material. Storing the



◀ **Figure 5.8 —** Linear white to black visualization of the number of ray steps per pixel. Darker pixels correspond to more ray steps. Atop of a plane, a refractive Stanford bunny models sits together with a reflective sphere, both casting shadow onto the plane. At the side there are three little boxes, also visible in the reflection. The scene is lit by a single point light source.

rays causes memory writes using atomic counter operations and after each pass the CPU starts the next ray batch with the respective number of shaders derived from the atomic counter holding the number of active rays. If the material at the intersection point has either a reflective component or a refractive component, but not both, the current ray direction is adjusted accordingly and no split occurs. In case of reflection and refraction, one ray is continued directly and the other is stored in the ray buffer for the next iteration. Which ray is traversed directly and which one is stored for the next iteration is subject to the implementation. The ability of splitting rays also enables shadows generation. Whenever a ray hits a fragment, a new shadow ray is created which targets a light source (cf. Figure 5.4). When the shadow ray hits another fragment, this causes shadow with respect to the translucency of the hit object. Otherwise, if the light source is reached without hindrance, the respective fragment is lit. Again, the ray is traced until the alpha value is saturated, it leaves the bounding proxy volume, or the shadow ray hits a light source.

To keep the shader executions from diverging too much, a ray is only iterated for a fixed number of steps $\max_{\text{steps}}$ and kept active in the next shader pass if not terminated. This prevents stalling of already finished shaders by long and complex ray traversals in the same ray batch. After all rays have either finished or reached their iteration count in the current pass, only active and newly created rays are traced in the subsequent pass.

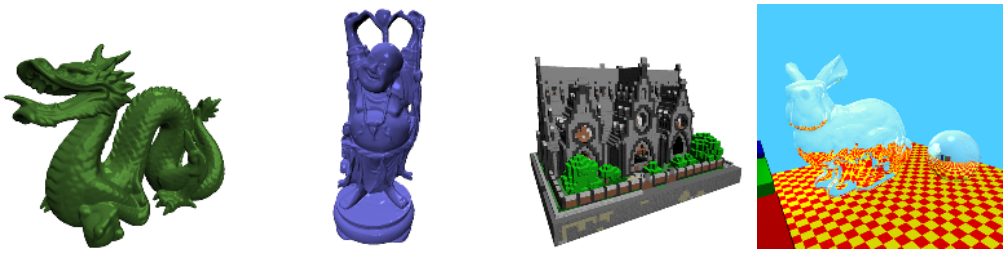
Figure 5.8 shows a visualization of the number of ray step iterations for a refractive Stanford bunny and the respective scene. Here, the structure of the bricks are clearly visible as they surround the geometry of the bunny and the sphere. In this example, the bounding volume resolution is 1024 and the brick resolution is 16. A total of 1.03M rays are generated in the scene for a viewport of 1000×1000. After the first pass which generated rays for each pixel in the viewport, the number of active rays drops rapidly since only refractive and refractive rays or rays exceeding the maximum number of iterations are continued. When enabling shadow rays with one light source the total ray count is about 1.88M.

5.5 Results & Discussion

The performance evaluation was carried out on an Intel Xeon E5-2637 machine with 128 GiB of RAM and an NVIDIA Quadro K6000 graphics card, the viewport size used was 1000×1000 for all measurements. The framerates were determined by averaging the frame rates of a 360° rotation around the y-axis followed by a 360° rotation around the x-axis. With these two rotations the entire scene is covered. Since the PVLLs are generated along the z-axis, a rotation around this axis will have only minor effects on the performance due to rather similar bounding volume traversals for this direction.

Figure 5.9 depicts the four scenes used in the evaluation. Both the dragon model and the Buddha statue feature a non-reflective, non-refractive material. The window panes in the house scene are refractive and the Bunny scene contains a refractive Stanford bunny and a reflective implicit sphere. The performance was measured for a *plain* rendering of the voxelized dragon and the Buddha statue and with global effects enabled for all scenes (denoted by *ray tracing*). In Table 5.1, the results are shown. Please note that measurements in Table 5.1 include only the rendering; the one-time costs of the voxelization are omitted. Information on the voxelization can be found in Section 5.5.1. For all scenes the rendering is possible at highly interactive frame rates.

To investigate the influence the view direction has on the voxel traversal, a scene containing a single sphere was set up. Again, the frame rate is recorded while the scene is rotated around the y-axis and the x-axis. The results for different bounding volume resolutions as well as varying brick sizes are shown in Figure 5.10. Since the sphere always has the same appearance for every view direction, one can conclude that the variations are due to the voxel lookup. The evaluation shows that while a brick size of 4 works best for a bounding volume of 256. For a volume resolution of 512 and 1024, brick sizes of 8 and 16, respectively, yield the best performance. All lines share the same features

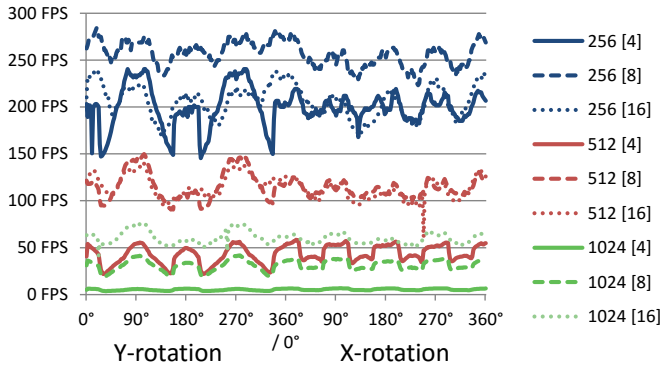


(a) Dragon (b) Buddha (c) House (d) Bunny

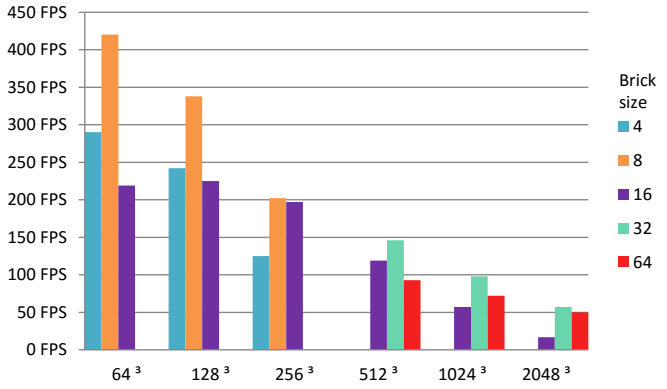
▲ **Figure 5.9 —** Scenes used in the evaluation. Results of the performance measurements are shown in Table 5.1.

Scene (Render Mode)	Resolution	# Voxel (10^3)	occupied Bricks	Memory [MiB]	FPS
Dragon (plain)	256, [8]	81.6	5.0 %	9.4	307.8
	512, [8]	326.2	2.6 %	70.0	134.5
	1024, [16]	1303.2	2.6 %	277.0	87.3
Buddha (plain)	256, [8]	67.1	3.7 %	9.2	310.0
	512, [8]	268.5	2.0 %	69.1	147.9
	1024, [16]	1073.6	2.0 %	273.0	97.4
Dragon (ray tracing)	256, [8]	81.6	5.0 %	9.4	47.6
	512, [8]	326.2	2.6 %	70.0	39.6
	1024, [16]	1303.2	2.6 %	277.0	33.0
Buddha (ray tracing)	256, [8]	67.1	3.7 %	9.2	48.1
	512, [8]	268.5	2.0 %	69.1	40.3
	1024, [16]	1073.6	2.0 %	273.0	34.1
House (ray tracing)	256, [8]	560.0	18.8 %	16.7	45.2
	512, [8]	2319.3	12.2 %	100.0	38.8
	1024, [16]	9444.9	12.2 %	401.0	31.1
Bunny (ray tracing)	256, [8]	139.2	6.9 %	10.2	37.3
	512, [8]	557.3	3.6 %	73.5	26.4
	1024, [16]	2233.8	3.6 %	291.0	20.8

▲ **Table 5.1 —** Results of the performance measurements for the test scenes. Columns denote the resolution of the bounding volume with brick size in brackets, number of non-empty voxels, percentage of non-empty bricks, memory footprint, and frame rate.



▶ **Figure 5.10 —** Ray-casted sphere rotated around the y-axis and x-axis for 360° at different bounding volume resolutions and brick sizes (denoted in brackets).



▶ **Figure 5.11 —** Evaluation of different voxelization and brick dimensions for the Stanford Lucy model (cf. Figure 5.3.

with slight variations. At the y rotation of 90° and 270° for example, the spikes indicate the highest performance for that particular view of the scene.

Figure 5.11 shows the evaluation of different brick sizes and voxelization dimensions, measured on a NVIDIA GTX 680. The evaluated scenes are depicted in Figure 5.3. The best brick size (middle bars) depends on the scene, especially on the number and arrangement of the empty sections.

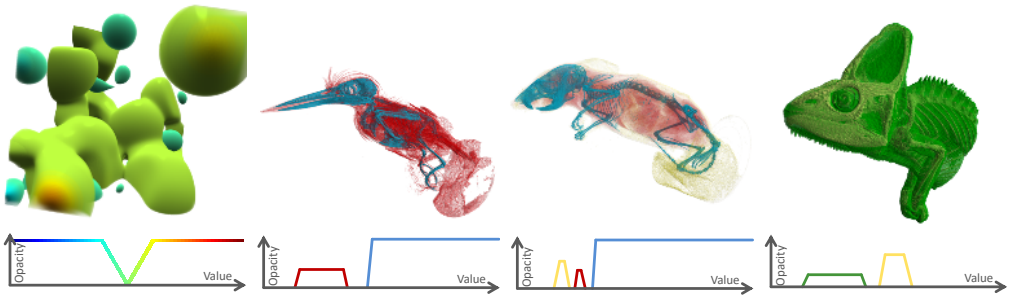
5.5.1 Voxelization

As mentioned before, the voxeliation of static scenes has to be done only once in the beginning. If, however, dynamic scenes are considered, the PVLLs have to be recomputed all the time. Table 5.2 shows the combined results of voxelization and rendering for the four test scenes where the voxelization is performed every frame. This demonstrates that the approach is capable of voxelizing and rendering a scene at interactive frame rates except for high volume resolutions and global effects rendering enabled.

While the approach performs well with the Dragon model and the Buddha

Resolution	256			512			1024		
Brick Size	4	8	16	4	8	16	4	8	16
Dragon	132.1	157.1	159.1	62.1	76.5	81.1	21.4	28.4	31.0
Buddha	84.6	94.8	94.1	44.0	52.2	53.6	17.7	22.4	23.8
House	29.4	29.6	29.5	6.1	6.2	6.3	1.4	1.4	1.4
Bunny	20.5	21.0	20.7	4.8	4.9	4.9	1.2	1.2	1.2

▲ **Table 5.2 —** Voxelization in combination with rendering for different volume resolutions and brick sizes. Measurements are given in frames per second.



(a) Kronecker (b) Kingfisher (c) Mouse (d) Chameleon

▲ **Figure 5.12 —** Data sets used in the evaluation of sparse volume rendering together with an illustration of their respective transfer functions. (a) Kronecker data set from geostatistics. (b)-(d) CT scans of animals: malachite kingfisher, old field mouse, and veiled chameleon.

statue, even for large sizes, the performance for the House and Bunny scene drops dramatically. This is due to the high depth complexity of the scenes, particularly in the z-direction, and the related costs for voxelization. In all cases, similar frame rates were obtained for brick sizes of 8 and 16. Although this seems contradictory to the findings in Figure 5.10 at first, the explanation is rather straight forward. The rendering process does not have a too big impact on the overall performance of one frame and the benefits of using the optimal brick size are canceled out by the additional costs of the voxelization.

5.5.2 Sparse Volumes

To illustrate the applicability of the approach in combination with volume rendering, three large-scale CT data sets of animals and two large data sets from the field of geostatistics are used. In Figure 5.12, the volume rendering results are shown for these data sets. The data sets are not inherently sparse

by themselves, but they share the common property that they contain large regions which are not of particular interest. For the CT scans of the animals (Figure 5.12b-d) this applies to the air surrounding the specimen. Other volume densities representing, e.g., tissue and bone structures are conserved and separated by applying a simple transfer function. The resolution for all three animal data sets is 1024^3 at 8 bit per data voxel.

The geostatistics data comprises two volume data sets which represent scattered measuring points which are interpolated to form a volume data set through a geostatistical method called *Kriging* [Kitanidis, 1997]. They are stored in a data-sparse format through low-rank *Kronecker* representations [Nowak and Litvinenko, 2013]. A domain size of 1024^3 and double precision was used in the simulation for computing the Kronecker volumes. The data is converted to 32 bit floating point values before voxelization and uploading into GPU memory to match the data structures. Despite the conversion no significant loss in the dynamic range could be detected when compared with the original results. Although the entire domain of the data set actually contains data, domain experts are only interested in the data outside the 95 % confidence interval of the standard deviation $\mathcal{N}(\mu = 0, \sigma^2 = 1)$. This turns the volume data sets into sparsely occupied volumes. Figure 5.12a shows the data outside the confidence interval thereby illustrating the sparsity of the data itself. The depiction in Figure 5.6b (center left), in contrast, shows the rendering of one Kronecker data sets with low opacities, generating the fluffy surface.

The generation of the PVLLs is carried out on the GPU for all volume data sets as described in Section 5.2.2. The computation is non-interactive but takes only 2.6 seconds for a 1024^3 volume data set. Table 5.3 shows the results of the volume benchmarks. The interesting parts in the Kronecker data sets still occupy large parts of the volume (34 % and 50 %) resulting in a comparatively large memory footprint. The volume occupancy for the CT scans are in a range of 0.5 % to 1.8 %, excluding the air surrounding the specimen (60 to 80 %) and some of the internal tissue but keeping the main features visible.

The approach delivers interactive framerates for most of the tested cases. Naturally, higher numbers of voxels result in lower average frame rates. This also directly impacts the memory footprint but still delivers low memory footprints compared to the original data set.

5.5.3 Global Effects Rendering

In Table 5.1, the rendering performance is shown for scenes with global effects enabled. Since the inclusion of refraction and reflection requires the utilization of the additional ray buffer, the frame rate for the Buddha and the dragon

Volume	Resolution	occupied Voxels	occupied Bricks	Memory [MiB]	Voxel per Byte	FPS
Kronecker 1	512 [8]	34.7%	39.7%	738.0	16.6	11.6
	1024 [8]	34.6%	37.1%	5874.3	16.6	1.9
Kronecker 2	512 [8]	49.6%	57.3%	1054.5	16.6	9.6
	1024 [8]	49.6%	53.4%	8408.9	16.6	1.4
Kingfisher	1024 [8]	0.5 %	2.3 %	107.6	19.6	35.4
	[16]		3.5 %	97.9	17.8	21.0
Mouse	1024 [8]	1.7 %	6.2 %	322.3	18.2	15.9
	[16]		7.5 %	302.8	17.1	8.4
Chameleon	1024 [8]	1.8 %	5.2 %	338.1	17.8	24.2
	[16]		7.2 %	322.8	17.0	13.3

▲ **Table 5.3 —** Results for the volume data sets regarding memory usage and rendering performance. The columns denote the resolution of the bounding volume with brick size in brackets, percentage of non-empty voxels, percentage of non-empty bricks, total amount of memory, number of voxels stored per byte, and the frame rate.

scenes drops to a mere 15 % and 35 % for bounding volume resolutions of 256 and 1024, respectively. At higher volume resolutions the actual costs of the ray traversal outweigh the impact of the ray buffer.

With shadow rays enabled, the performance drops to about a third due to the increase in the number in rays as well as storing the shadow rays in the additional ray buffer. Different upper limits of ray steps $\max_{\text{steps}}$ per rendering iteration were also tested without noticing a significant difference between 256, 512, and 1024 steps.

5.5.4 Limitations

In its current state, the implementation has several shortcomings which are partly due to the inherent nature of voxels and partly due to unoptimized code. For the voxelization, the biggest performance issue is the slow speed for updates in every frame for some scenes (cf. Table 5.2). This, however, could be overcome by inserting the fragments immediately into distinct bricks instead of the global buffer. Thus, only fragments in the individual bricks have to be sorted which should improve overall performance.

Another limitation is that reflection, refraction, and shadow generation for geometries relies on the voxel representation since rays only hit cube-shaped

voxels. Thus, the PVLLs approach cannot keep up with other ray-tracing programs, e.g. NVIDIA OptiX, neither in terms of computation time nor in terms of image quality.

5.6 Summary and Outlook

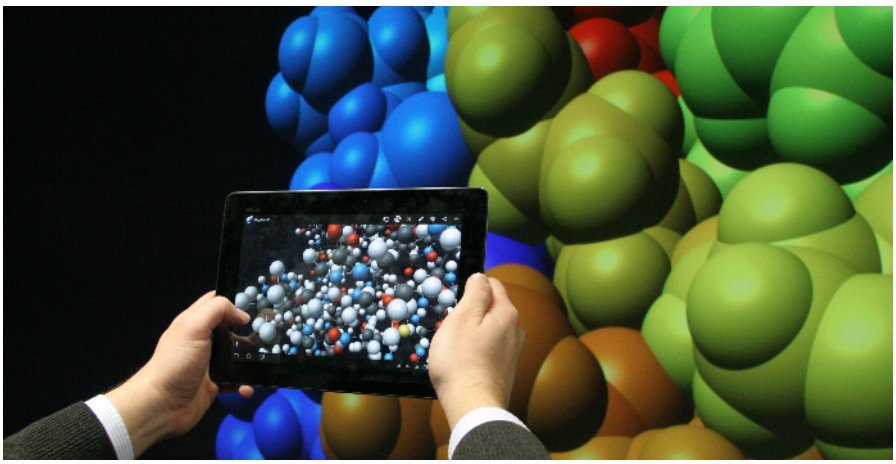
For future research, the ray traversal can be optimized further by employing adaptive space subdivision methods like BSP trees or kd-trees instead of uniform bricks. This might also lead to a reduced memory footprint, thus making the approach more profitable for even larger volumes. Additionally, the implementation of level-of-detail mechanisms could improve the performance for large proxy geometries. Here, the idea is that voxels in the back of the cube which are not traversed by the rays could be omitted, potentially speeding up the traversal step. By achieving a higher performance in the rendering phase, the number of secondary rays can be increased and, thus, enable effects like caustics, sub-surface scattering, or diffuse mirroring. The memory footprint could be reduced further by using a tighter bounding box scheme which is aligned and fitted to the sparse data, thereby eliminating the empty space within the bricks.

Section 4.4 details how to use PPLLs for distributed and remote rendering. With the similar data structure of the PVLLs, it might be possible to also extend it to make use of multiple distributed rendering engines. Similar to multi-variant rendering (cf. Section 4.6.2) for molecular surfaces, distributed PVLLs could be used for a comparison of different volumes.

The PVLLs approach presented in this section can be used for rendering sparse volume data as well as voxelization and rendering of geometry including reflection and refraction at interactive frame rates. PVLLs are the only approach that can handle arbitrary representations using voxelization. It adapts the original A-buffer concept of PPLLs and extends it to store the entire scene in three dimensions. The performance results show that PVLLs reduce the memory footprint of sparse volume data while still being interactive.

CHAPTER 6

Using Mobile Devices for Interaction with Visualizations



▲ **Figure 6.1 —** X-ray scenario with the VdW representation displayed on the VISUS power wall and the inner ball-and-stick representation rendered on the mobile device.

With the larger data sets, the visualizations also get more complex and more detailed. The last component of the visualization pipeline is the user, viewing and interacting with the presented data.

There are established interaction methods like for example mouse and keyboard for workstations or tracking systems for large display walls. This chapter will

describe mobile devices as an alternative to these methods. Mobile devices feature a touch-enabled display which can be used as an input device as well as an output device.

The input is similar to a touch pad and in combination with the display it can be used as different user controls: buttons, sliders, menu lists, and so on. This allows the user to perform simple tasks like navigation and also complex tasks like direct selection or manipulation of data. Moreover, the controls allow the user to operate the system and modify all parameters in a user-friendly manner. For the output device, it can be used as an additional screen, displaying contextual information or other visualizations. The built-in sensors allow new interactions, for example an x-ray-style looking-glass for complex data sets (see Figure 6.1).

Parts of this chapter are based on the paper "Remote Rendering and User Interaction on Mobile Devices for Scientific Visualization" [Krone et al., 2015].

6.1 Motivation

Interacting with a visualization system can be challenging, especially if the system uses more than the two planar axes of the screen. When using a standard mouse, the screen area is mapped to the surface the mouse is moving on. The third dimension is then controlled e.g. by the mouse wheel. Alternatively, a combination of mouse and keyboard can be used, mapping the depth axis onto two buttons.

A multi-touch enabled smartphone or tablet, however, offers a wide range of interaction possibilities by using gestures. Using this type of device allows the user to perform similar actions on the large visualization the same way he is used to for mobile devices.

Additionally, the touch pad of the mobile device is also a display which enables the user to view further information or to view a completely different visualization. Feedback can be given by sound or vibrations, so the user is not necessarily distracted from the visualization he is looking at.

For large displays like the VISUS power wall, there is also the possibility for physical navigation by the user. Standing at a certain distance allows the user gain an overview of the scene and then he can walk directly to the interesting part. Studies by Ball et al. [2007] suggest that users tend to navigate physically, e.g. walking or head moving, even if they have the ability to directly interact with the visualization, e.g. by zooming. Still, the mobile display can be useful for the viewer, e.g. for parameter control. For some types of power wall setups, the diffuser must not be touched, though it cannot be used as a touch screen for

interaction. Using mobile devices for these types of power wall retrofits these capabilities, then allowing direct user input.

Additionally, a mobile device can either be used for the direct interaction or it can be used to keep an overview over the scene even when the user stands directly in front of the display. It could display the total scene, any specific part, or additional information so that the user does not have to wander around to inspect different parts, e.g. for a visual comparison.

Mobile devices are usually powered by their own battery and communicate with wireless technology, allowing the user to move freely in front of the display device. Additionally, the device can be passed to other users easily and it does not require a flat surface like a mouse to be operated on.

The graphical capabilities of the devices allow them to be used as a second screen, also displaying advanced visualizations like on the large display. These visualizations can either be streamed or rendered on the device itself, depending on the hardware and complexity of the visualization. This implicitly enables remote visualization on the device and the option to *take the visualization home* by storing all necessary data on the device for later use.

6.2 Related Work

Interaction with visualization systems is a wide topic and there exist a plethora of methods [Badillo et al., 2006; Khan, 2011]. Ni et al. [2006] survey interaction techniques and devices for large displays, with the focus on tracking and gesturing. Tracking devices use camera-based systems that can detect special markers or track the user(s). For gesture input, the users may use devices fitted with accelerometers, effectively recording the movement and using this information for the input. Cockburn et al. [2009] give an survey of "overview+detail, zooming, and focus+context interfaces", aiming to review and categorize state-of-the-art of general interface concepts that also apply to large displays. Hardy and Rukzio [2008] and McCallum and Irani [2009] use mobile devices for cursor positioning on large displays. Their approach uses the touch system for limited mouse input only while the approach presented here is more flexible and can also be used as a second screen. Cheng and Zhu [2014] demonstrate a tablet-based system. The user can select the view port then this part of the screen is displayed on the tablet device, being an output device only.

For remote visualization on smartphones, the standard desktop applications like VNC, Microsoft Remote Desktop, or TeamViewer do also have versions for mobile devices. Eissele et al. [2009] present a system for mobile devices

which can display additional information and blend it into the camera view, creating a type of augmented reality vision. The system is capable of streaming or local rendering of context-aware visualizations. Lamberti and Sanna [2007] demonstrate a streaming-based visualization framework for mobile devices, using the MPEG codec for streaming complex 3D models in motion. Wessels et al. [2011] describe how to use WebSockets for remote visualization on smart phone hardware. Hoffmann and Kohlhammer [2009] detail a portable remote visualization framework that can easily be integrated into existing visualization applications but focus on portability and visualization instead of the interactive focus of the system presented here.

Commercial and industry-grade frameworks also support remote visualization on mobile devices. The FEI Visualization Sciences Group developed RemoteViz[1] for the Open Inventor framework. Dragon HPC offers the remote Visualization Platform[2] for their services that also supports mobile device clients.

In contrast to the previous works and available systems described above, the system and concepts described here aim to include the interaction and the visualization on the mobile device. Additionally, the collaborative part is an important aspect, broadening the scope of the presented system. Of course, the system presented here builds upon and extends the same concepts explained in the previous work.

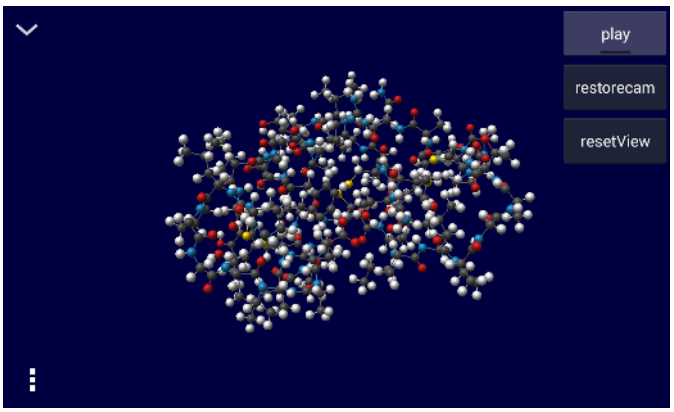
6.3 Technical Foundation

In the context of this thesis, the system consists of a large visualization wall driven by a number of rendering nodes. Additionally, an operator node is used to steer the visualization on the power wall. For the mobile devices, a number of wirelessly connected mobile phones or tables are assumed. Any machine or mobile device connected to and synchronized with the system can receive the visualization or be used for interaction.

6.3.1 Data Flow

Although the visualization might have a plethora of parameters to modify, many of them might not be of interest for a presentation and are fixed before hand when preparing the presentation or in a collaboration scenario. Therefore, the few remaining parameters can be put into special input categories displayed aside with the visualization so the user can access them without searching

[1] http://www.vsg3d.com/open-inventor/remoteviz (accessed 29.06.2015)
[2] http://dragon-hpc.com/remote-visualisation.php (accessed 29.06.2015)



▶ **Figure 6.2 —** Mobile device user interface with the model in the center and buttons for quick settings on the right.

through sub-menus. This provides fast and comfortable access for the user. As an example picture of the prototype, Figure 6.2 shows the user interface of the mobile device with a ball-and-stick molecule representation and buttons to access parameters.

To minimize the risk of invalid input values, the available parameters can be restricted so that they cannot be changed by the remote application. This also applies for the data set parameters. The application only presents files to the user which are compatible to the current visualization, avoiding possibly unpleasant situations during the presentation.

Establishing Connection

When the mobile device application is started, there are different ways to connect to the visualization system. For security reasons, a password can be required to prevent unauthorized access.

Automatic Invitation A fixed IP address or a network broadcast containing this information can be used for the device to connect. This allows a fully automatic configuration of the mobile device without input from the user.

Semi-Automatic Invitation Another option is to use machine-readable information like QR codes[3] to provide the connection details. In case of the presentation scenario, the power wall can display the QR code and the user uses the application to scan the code (cf. Figure 6.4a). For the collaboration scenario, the QR code can be shared via file transfer or email and then read using the mobile device. Alternatively, the code can be printed, too.

[3] http://qrcode.com/ (accessed 29.06.2015)

The code provides all information for the application to connect to the visualization system. Of course, any other machine-readable system can be used, too.

Manual Additionally, there is always the option for the user to manually enter the connection details. This can also be used as a fallback if the other methods fail to work.

If the network setup allows this, the most comfortable variant is the automatic invitation for the user. This, however is a public announcement, so every compatible client can connect and access the visualization. When the access needs to be restricted or broadcasting the invitation is not possible, the semi-automatic invitation should be used.

Display to Device

For an interactive viewing of a visualization, the application typically offers a wide range of interaction possibilities for the user. These include changing the data set, the filtering settings, the mapping and the rendering configuration including the viewport and camera settings. In short: the user should be able to interact with all four stages of the visualization pipeline (cf. Chapter 2.4).

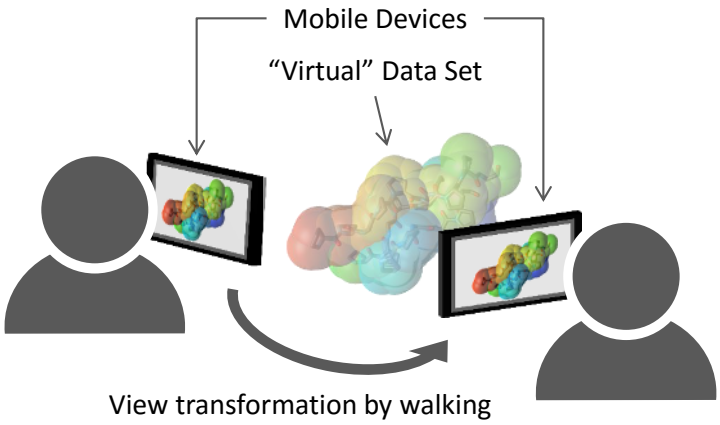
In this context, the application delivers its current state to the mobile device and the user has the ability to change the state. For the interaction itself, the prototype application makes use of two configurable parameter bars on the left side and on the right side of the screen. The user can put the parameters there for quick access and the user interface is not overloaded with unnecessary controls.

Device to Display

When the user modifies the application state, the mobile device communicates the change in the parameters to the host who is in control of the large visualization. However, for a local rendering on the mobile device, the user is able to change its parameters which do not alter the large visualization.

6.3.2 Device Localization

To have an intuitive way of aligning the view of the mobile device with the user's perspective, two different methods are proposed. Alternate device localization methods like for example optical or radio-based tracking systems [Hightower and Borriello, 2001; Gupta et al., 2013] are out of scope of this thesis. Figure 6.3 shows an example of two users and their mobile devices viewing the same data



▶ **Figure 6.3 —** Two devices showing different view angles of the same data set. The view transformation measured by the internal motion sensors or derived from the camera is applied to the internal view matrix.

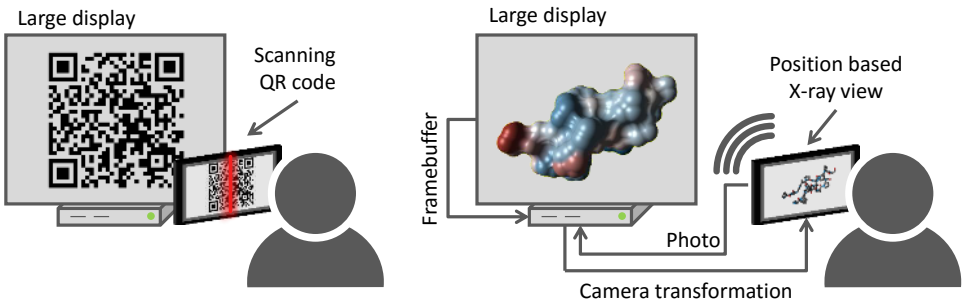
set. The view transformation derived from the sensors or camera ensures a consistent view for the users while wandering around.

Motion Sensor Localization

A simple way of aligning the view of the device with the big display is using the internal sensors of the mobile device, in this case the acceleration sensors and gyroscope. For the initial view, the user holds the device in parallel with the display and the subsequent motion is tracked by the sensors (cf. Figure 6.3). The motion is accumulated and the view on the mobile device display is updated accordingly. To counter sensor drifting issues, the user can again hold the device like for the initial view, recalibrating the sensors. Every parameter change from the external system like camera changes on the large display are still applied to the local visualization, giving the user the same perspective on the data set when looking through the mobile device as when looking on the display directly. For fine-tuning, the user is still able to move, rotate or zoom the camera by hand.

Photo-based Localization

Another possibility for device localization is to use the camera many devices have attached. When holding the device, the camera takes a picture of the current view on the large display and the image is transferred to a node which can access the large visualization, e.g. the operator node (cf. Figure 6.4b). Then, the image taken is compared to the actual view and a homography matrix is calculated, e.g. by using the functions provided by the OpenCV framework [Bradski, 2000]. From this matrix, a camera transition can be calculated [Klein, 2006] and the transition matrix is sent back to the mobile device. Now, the view



(a) Connection Setup (b) Camera based localization example

▲ **Figure 6.4 —** QR-code based connection setup (a) and schematics of the camera based localization approach for mobile devices in front of a large display setup (b).

from the user is aligned with the view on the device, e.g. allowing x-ray like visualizations (cf. Figure 6.1).

When the user moves the device, the view can be updated using the motion sensor approach explained above. Here again, manual input can be used for fine-tuning. To keep drifting errors caused by noisy sensor information at a minimum, the photo-based approach can be used periodically.

6.3.3 High-Scale Visualization on Mobile Devices

This section briefly describes different possibilities for displaying the content of the large display wall on a mobile device. Basically, the difference is whether the rendering takes place on the local device or on a remote system which streams the final images only.

Local Visualization

As the graphics capabilities of today's mobile devices are very advanced, the devices itself can be used for rendering the data. The graphics chips support operations necessary to render particles or volumes on the mobile device. The limitations here are the power consumption and the performance of the mobile graphics chips. However, the PPLLs (cf. Chapter 4.3) and PVLLs (cf. Chapter 5) algorithms cannot be ported directly to current mobile devices running OpenGL ES, as important atomic functions and buffer storing methods are not supported. The standard approaches for mesh or volume rendering are supported and can be used for visualization.

For the local visualization, the data is delivered by the remote system. It can

either be streamed, e.g. for insitu-simulations, or downloaded at once, e.g. for data already available. Alternatively, files stored on the device can be used for the visualization.

Remote Visualization

The other possibility is to create the images for the mobile devices on a remote system and only send the final images. Without a dedicated viewing application, this reduces the complexity on the mobile side, but also removes the option for the user of the mobile device to change the visualization settings locally. The visualization is created on the rendering system, though it may accept parameter inputs from the mobile devices. However, these parameters are then applied to all viewers of the current visualization.

Video streaming has become a very active field in the recent years. Video on demand, TV streaming and video websites pushed the development of new codecs which can adapt to the current network situation.

The network is the main bottle neck for remote visualization using video streaming. For streaming the visualization, the frame buffer is read to the host memory and then the encoding is done on the CPU before sending it to the mobile device. Using multicast, the network load can be reduced as messages are not duplicated for multiple receivers on the same path.

Using a standard video codec systems, it is possible to use any remote video playback application without the need for a dedicated application on the mobile system. Thus, this approach is most suitable for a broad user spectrum with heterogeneous mobile devices.

When streaming data or video, the available bandwidth per device has to be taken into account. While video codecs like H.264 [Wiegand et al., 2003] can adapt to changes in bandwidth and still provide a live stream, even if the details and quality is reduced. But when streaming data, there is usually no such mechanism and a reduced bandwidth also bogs down the framerate.

Another option would be to stream still images for low end devices. However, this method suffers from a high delay which is caused by the image encoding on the CPU of the rendering node. Additionally, as this is not a common practice, a specialized application is necessary for the mobile devices.

6.3.4 Limitations

The described system has some implicit limitations. First, there has to be a reliable network connection between the device and the host system to provide a smooth user experience. For multiple devices which stream the video

content, this can become a bottle neck, depending on the network infrastructure. When only parameter changes are communicated, less network load has to be transferred, allowing more machines or mobile devices to participate. For transmitting the data set, a fast connection is favorable, but a short startup delay is easier to get over with than a stuttering visualization. Besides the bandwidth required for such a system, the latency also plays a role. The user input has to be transferred to the controlling systems fast enough to provide an interactive or near-interactive experience.

Second, for the local visualization, device hardware must be capable of rendering the data. While every device should be able to play video or display images, an interactive rendering needs sufficient graphics capabilities. When rendering local data, the user needs to have access to this data which might not always be in the interest of the owner of the data. Using remote visualization, e.g. via streamed video, the user can interact with the visualization but cannot access the data directly.

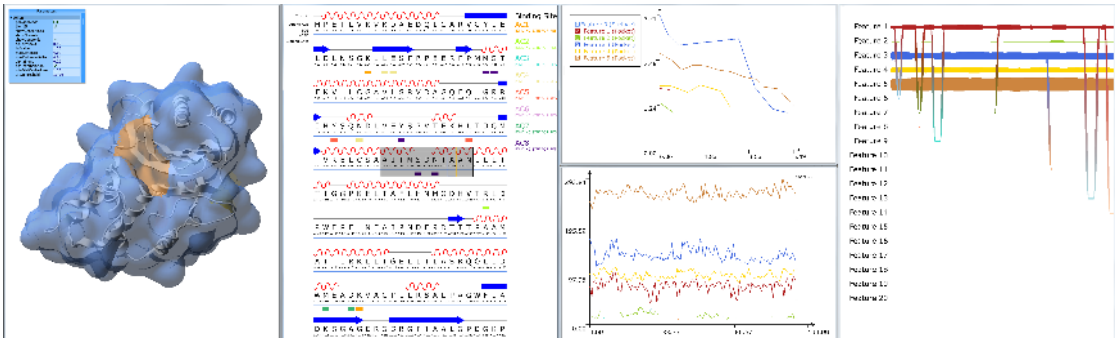
The third aspect is that the user has to hold the device, thus only one hand can be used for interaction on the touch display. This is sufficient for the basic touch or button inputs but more complex interaction methods or additional controllers cannot be used easily.

6.4 Mobile Device Interaction Concepts for Visualization

In the following, an example visual analysis application is presented together with concepts of how to use mobile devices with that application. The application was first presented in Krone et al. [2014b].

6.4.1 Visual Analysis Application for Dynamic Protein Cavities and Binding Sites

The simulation of proteins can provide important insights into molecular mechanisms on an atomistic level. Visualization is a vital method to analyze the simulation data since it illustrates the location and shape of the molecules, or it delivers an abstract representation which focuses on a certain property. Animations of molecular simulations show the dynamics and conformational changes. However, more advanced analysis methods and the visualization of their results can improve understanding significantly and allow researchers to detect certain phenomena that are not obvious from the raw data.



▲ **Figure 6.5 —** Visual analysis application with the views of the protein, sequence diagram, cavity diameter and area, and relational graph showing the evolution of the cavities over time.

Figure 6.5 shows a visual analysis application that uses a set of automatic and semi-automatic feature extraction and analysis methods, visualizes the results in different ways, and supports user controls to parameterize the analysis and filter the results. It adheres to the "visual analytics mantra" by Keim et al. [2008]: "*analyze first – show the important – zoom, filter and analyse further – details on demand*". A similar definition for visual analytics was given by Thomas and Kielman [2009]. For spatial data, this is generally referred to as *visual analysis*. According to Kehrer and Hauser [2013], a tight integration of the visual mapping, visual interaction concepts, and computational analysis are crucial for a visual analysis application.

The goal of the visual analysis process is to gain knowledge or a deeper understanding by visual examination of a data set. In this case, the input is dynamic protein data, mainly obtained by molecular dynamics simulations. Topology files give the number of atoms, their type and spatial relationship, and additional meta information about the atoms or the protein. A trajectory file provides the path of all atoms over the simulated time. Alternatively, the current atom positions can be received from a running simulation. From this information, the application extracts a visual representation of the protein, e.g. a molecular surface. The application is an extension to the method of [Krone et al., 2013] that uses the molecular surface mesh to extract cavities such as pockets or channels as follows: The cavities are classified and centerlines are extracted for all cavities. This allows the user to investigate the inner structure of the protein, e.g. the channel diameter. The cavities provide evidence of possible binding sites of the protein – connections for other proteins to dock on. Chemical information about binding sites can be added and mapped to the visualization. All this extracted information can be explored using brushing and linking in their respective views. For example, when a user selects a binding site

in a diagram, it is also highlighted on the mesh. The user can now determine whether the binding site is located in a pocket. With this system, the domain expert is provided with context to the selected information and features.

6.4.2 Mobile Device Interaction

Visual analysis applications give the user a rich environment where he can view, browse, filter and interact with the data. Brushing and linking [Doleisch, 2004] is one method to interact across different views and windows.

A mobile device can be used to control the visualizations, e.g. as a so called *second screen*. A second screen extends the visual area, allowing the developer to add content for the user. For the user, the information is accessible without the need to switch views or reconfiguring the application. Especially the touch ability of a mobile device, or on a desktop touch monitor, together with a visualization can improve the user experience. The concepts described here can be used for a desktop scenario using standard workstations as well as for a setting which uses large displays such as power walls.

Basically, there are two different types for interaction using mobile devices: Firstly, using the device for input only and, secondly, using the visualization and input capabilities of the device together. For the input-only method, it serves as a touch-pad-like device with contextual controls. In this case, it is used in combination with a computer system which is controlled by the device. The second method adds the opportunity to use the mobile device as a stand-alone system. Both the mobile device and the main visualization system are still connected but also are only loosely coupled, e.g. by using the same data but not necessarily the same visualization or the same parameters.

In the context of the visual analysis application (cf. Section 6.4.1), the mobile device could display the graphs while the main display shows the visualization. Or different visualizations of the protein can be shown on both displays, allowing the user to see different aspects of the data set. Having the device and application connected, input concepts like brushing and linking can also be applied across displays, allowing the user to interact on the mobile device and the workstation.

In the following, two different usage scenarios are described: a single user working with the mobile device and a collaboration scenario where multiple users have access to the same visualization system with their mobile devices.

Single User

In a presentation in front of a power wall, the mobile device can be used for simple tasks like forwarding the slides or displaying text for the speaker. Basically, it works as the control for the presentation slides. As this is a standard feature for presentation applications, the mobile device can also be used for controlling the non-static parts of the presentations, e.g. by switching the applications or by embedding the presentation controls in the application.

Touch controls allow the presenter to steer and modify the presented visualizations. Having the visualization from the large display also available on the mobile device, the presenter knows what the audience is seeing.Visual updates like parameter changes happen on both devices, thus the presenter has no need to turn away from the audience.

The mobile device can be used to display location and view dependent data. For instance, when in front of a powerwall, the mobile device displays the same scene but in a zoomed-in view. When the device and the visualization on the larger display are synchronized (cf. Section 6.3.2), the device can be used as a interactive magnifying glass, displaying other color maps, additional information, or a completely different visualization. For example, when having the protein surface displayed on the power wall, the mobile device can be held up and show the interior ball-and-stick representation, or highlight internal cavities, effectively creating an x-ray metaphor. In this context, the data can either be streamed to the device and rendered there, computed by a dedicated rendering node or be taken from the visualization directly, scaled according to the size of the mobile display. The specific option is implementation dependent and strongly relies on the hardware environment.

Besides the local usage of the device, it can also be used remotely. As long as the connection details are known, the system can connect and receive the data or stream the video. Once a connection is established, the data transferred to the device can be stored there and the visualization can be used even when the connection is terminated. This could be used for conferences, presenting a topic and then sharing the data and visualization with the audience. Or it can be used for public events such as open house. Interested viewers might want to *take the visualization home* after they have seen the demonstration.

Collaboration

Basically, the techniques in for the single user can also be applied when multiple users are participating. Of course, the mobile device can also be passed around when multiple users are in front of a display. For collaboration, this makes the

interaction easier than with a large keyboard or a mouse which always needs a flat surface.

When every user has his own device, all of them can be connected to the visualization system. Here, each device can be used coupled and synchronized, or decoupled from the larger display. The options from the single user case above apply here, too. Each user can work with his own device and visualization, e.g. walking in front of the display and using it as a magnifier glass.

Having multiple users connected to the visualization system, the settings can also be shared among them. Points of interest can be inspected together, and the parameters and views can be shared, thus enabling an in-depth collaboration experience.

Another scenario is remote collaboration. The connection details like the QR code can be made available remotely, e.g. by sending it per email or on a website, and the remote user can connect to the visualization system. For a fully interactive experience, the network has to support the required bandwidth and also provide a low latency.

By using open standard technologies, client applications for all kinds of computers or devices can be implemented. While workstation computers are present in most offices, mobile devices are also powerful enough to be used as remote clients. All the user needs is such a device and Internet connection to participate in a collaboration scenario. The ubiquity and simplicity of mobile devices is a big convenience factor here.

6.5 Summary and Outlook

This chapter discussed concepts of how to use standard mobile devices for interaction with visualization systems. Although the mobile systems are not as powerful as the graphic hardware in a workstation, they can still deliver a good visual impression of the data. Remote rendering allows delivering the same visual effects and quality as provided for a workstation display, assuming that the network connection can provide the required bandwidth and low latency. The touch display of the mobile devices allows users to control the visualization and other sensors like camera and gyroscope allow for an accurate positioning in front of a display. This allows for displaying the same view on both the mobile device and the larger display, allowing for a magnifying-glass like view.

Mobile devices are a relatively new class of devices. The system described in this chapter uses a client-server approach to deliver the visualizations. Another approach is to use participating devices in a cluster-like setting, each one

rendering a specific part of the scene and sharing it with all other devices. Using a broadcast-style network architecture (like a wireless local area network), the data could be sent to all other devices at relatively low cost, using compression mechanisms and detail reduction when necessary.

With respect to the pace of the development in mobile device hardware and software, the mentioned limitations might be obsolete in the near future. This means that what runs today on a workstation might be computable on a mobile device in the near future, but due to new rendering techniques or more detailed data the proposed remote visualization methods will still be relevant. However, as stated in the motivation of this thesis, the data sizes will most likely continue to rise as well as the computation capabilities of the computers, clusters and mobile devices. So there will still be a need for the presented methods and new developments in this field.

The Google Glass project[4] and the Occulus Rift project[5] show possible developments in the area of mobile devices. Whereas the Google Glass projects information into the field of vision of the user, the Occulus Rift project provides immersive virtual reality experiences by using one separate display region per eye, creating a stereo effect. Both devices connect to a host, a workstation computer or mobile device, which delivers the data.

While the mobile devices are not yet powerful enough to drive the Occulus Rift glasses, they might be in the future. This would enable a much deeper immersive experience for the user.

For the Google Glass project, they could be used to display additional information. E.g. a region selected by other users, a different viewing mode, or text displaying statistics with respect to what is in the field of view of the user. The camera in the Google Glass can be used to perform a more steady tracking of where the user is looking at in the picture. However, as for now, the Google Glass project offers only limited input capabilities, e.g. by speech.

As said above in the Google Glass example, users could start to interact with each other or the visualization displayed on each other's devices. Therefore, they need not necessarily be at the same location but can work together remotely. Besides the interaction possibilities mentioned above, the mobile device could show what the other users are seeing, their annotations or highlighted regions, or the like. This immediately suggests the transfer of settings between users. In the current setup, all users modify the same settings of the visualization shown at the display wall. For the remote cooperation scenario where each user has his own instance of the visualization or uses the mobile device (with the reduced

[4] https://www.google.com/glass (accessed 29.06.2015)

[5] http://www.oculusvr.com/ (accessed 29.06.2015)

variant of the data) for visualization, the transfer of the visualization settings could be implemented.

There is also ongoing research in the area of hands-free interaction. This could be combined with the glasses mentioned above, giving the user an interactive and immersive feeling of the visualization.

Last but not least, holographic projections might advance the stereoscopic projection of today. Here, the users could walk around the projection and view it from different angles simultaneously, without the need of an additional mobile device. Interactions could be based on reaching into the projection, giving the user the feeling of a real hands-on interaction.

CHAPTER 7

Conclusion

While a detailed outlook and conclusion of the topics was given in the respective chapters (see Sections 3.5.2, 4.7.1, and 6.5), this chapter will describe an integrated visualization system using the techniques and concepts introduced in chapters 3–6. Additionally, an outlook for future work and a summary of the thesis is given.

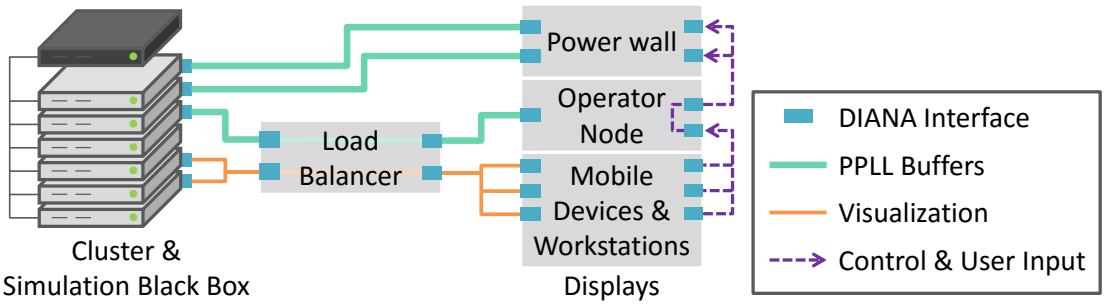
7.1 Big Picture

An integrated system using the parts presented in this theses could be completed as follows: an operator node for controlling the system, a visualization cluster based on render nodes equipped with a powerful graphics cards, a large display wall, driven by a subset of the cluster nodes, and user devices like mobile devices or workstations. The system is also connected to the Internet, making it available for remote visualization (see Figure 7.1).

In this scenario, the data to be visualized comes from a black box. So it can either be a replay of pre-computed simulation or computation files or it is an in-situ visualization of a live computation or simulation.

7.1.1 System

The software back end of this system is powered by DIANA. Small daemon applications run on the cluster nodes, receiving commands or parameter updates, and returning the computation or visualization result. The power wall



▲ **Figure 7.1** — Example scenario of how DIANA (cf. Chapter 3), PPLLs (cf. Chapter 4), and the mobile device interaction (cf. Chapter 6) can be used together in a visualization system. The operator node steers the visualization on the powerwall system. The simulation or computation is depicted as a black box system (upper right) which all cluster nodes can access. Mobile devices or workstations can connect to the operator node to participate in the presentation, or receive data or visualization streams.

nodes use DIANA to connect to the visualization nodes and to provide a control interface for the operator. The operator node also has a connection to the visualization nodes for its own visualization (alternatively, the local system can be used for rendering) and controls the power wall nodes. It also features a DIANA interface so external clients like the mobile devices can connect to the system and retrieve data or stream visualizations.

Different applications can connect to the daemons, thus some sort of load distribution has to be established. For example, when the large display powerwall is switched on, the driving nodes connect to the load balancing machine and in return get the connection details for a DIANA instance which has enough resources to handle it. The node-assignment does not have to be fixed and can vary over time. In case of the power wall nodes, one can imagine that a render node works exclusively for its respective client to deliver maximum performance. However, this is only an illustrative example. The realization highly depends on the implementation, hardware situation, and the visualization itself.

A main application controlling the whole system runs on the operator node. Its purpose is to initiate the startup of the applications on the cluster nodes and powerwall nodes. It controls the visualization on the large display and can also be used for all clients like the mobile devices to connect to this machine. This allows the operator to restrict the access to the visualization or override parameter choices of the users.

External applications running on mobile devices or workstations also connect

to the operator node. It distributes further internal information like how to connect to the load balancer, how to request access to a rendering nodes, or how to retrieve the data. As it is connected to the Internet, collaborations or presentations with partners world-wide can be supported. The interaction methods presented in Chapter 6 also apply in this scenario.

7.1.2 Operating Procedure

On startup, the daemon loads the available plugins and the operator node controls the further actions like loading the data set or setting the visualization parameters. An application, either the power wall display node, the workstation, or a mobile device, simply requests a frame from the render nodes and displays it. For the rendering, the distributed PPLLs algorithm can be used. It is flexible, easy to use, provides distributed OIT rendering, and is capable of rendering all sorts of data like geometry meshes, volumes, or implicit surfaces. Upon request, the visualization daemon sends a PPLLs buffer to the client which can then do further operations, e.g. merging with other buffers or post-process it. Finally, the visualization is displayed on the devices.

However, any other remote rendering compatible algorithm can be used in this scenario, too. Some sort of synchronization protocol (cf. Section 4.4) could be used to ensure the exact same time step for each display. While it is necessary for a multi-display device like the power wall to have a consistent and artifact-free image, it might not be of much use for a multi-client system. The reason is that the slowest client would dictate the frame rate if all displays would be synchronized. Therefore, for multi-client systems the synchronization needs could be relaxed.

7.1.3 Implementation

For the implementation of the DIANA plugins, the existing PPLLs applications can be used and wrapped in DIANA code. On the mobile side, DIANA can be added to the existing applications. If the mobile devices are powerful enough and support the necessary operations, the PPLLs algorithm can also be ported directly. Otherwise, the remote visualization and remote rendering methods mentioned in Chapter 6 can be used. However, as the size of the buffers is comparatively large here, a streaming video visualization or local rendering might be the better choice here, especially if the network bandwidth is limited.

The load balancing process can be implemented straight forward, using a fixed resource allocation for static assignments like the render nodes and a round-robin procedure for dynamic clients, e.g. the mobile devices or additional

workstations. If this simple system is not efficient enough, more sophisticated load balancing strategies [Willebeek-LeMair and Reeves, 1993] could be employed, too.

7.2 Future Work

The presented prototypes of this work were mainly used for visualization of biomolecules or FTLE visualization (cf. Section 3.4). Obviously, the system can also be used for other types of visualization. For volumetric rendering, the PPLLs can be used in the same way the PVLLs are combining geometry and volumes (cf. 5.2.2), and in combination with the distributed rendering, standard volume bricking methods [Parker et al., 1999] can be applied. For standard volume rendering, there might not be an advantage but when combining the volumes with mesh data, e.g. the volume representing the air flow around a meshed object, both rendered with transparency effects.

Therefore, the PPLLs and PVLLs are an ideal basis for a generic rendering framework. They both can handle arbitrary input data and, in the case of the PVLLs, can even be used for real-time ray tracing effects.

The current implementation of the display side of the PPLLs approach requires modern graphics memory functions provided by OpenGL 4 and its extensions. These functions are not available on all hardware platforms, especially on mobile devices. Extending the algorithm to use texture memory storage could make this available for mobile devices in a remote rendering setting, too. For this setting, render nodes transfer the data in chunks, prepared to fit into the texture memory of the devices. The display algorithm on the mobile device then interprets the data to render the correct visualization. In the case that future graphic APIs support those functions, of course, the algorithms can be used directly.

The current PPLLs system supports static and dynamic data sets. However, loading a new time step requires a complete rebuild of the data structures. Using the differential information of both time steps or any other mechanism that provides an preview from the current time step to the next could be used to update the current visualization without rebuilding from scratch. For example, particle data with velocity information could enabling the approach used for the two-sided transformation of PPLLs (cf. Section 4.4.3). The display node could update the particle representation and transfer it into another PPLL if necessary, or perform an interpolation or extrapolation instead of requiring a complete data update. This requires a tight coupling of the data delivering process, let it be pre-computed data, or a in-situ simulation or computation, and the

visualization. The visualization needs to know about the additional information and how to interpret the data. On one hand, this requires more computation time on the display nodes. On the other hand, the overall amount of data transferred, and thus the network waiting time, could be reduced. Adaptive algorithms could be used to find the sweet spot depending on the current hardware and network architecture, and visualization load.

The interaction concepts presented in Chapter 6 focus on the visualization part. Especially the mobile device interaction is used for an intuitive and seamless visualization experience for the user. The concepts could be advanced to also include computational steering mechanisms where applicable. Standard parameter settings can already be prepared for the simulation and computation scenarios. More complex operations like module connection or altering the data are not included in the current setting and are subject to future work. However, the goal should not be to replace the operator node or to include all functionality in the mobile device. Instead, the functionality needed for the target application has to be made available, e.g. for presentations or collaboration scenarios.

7.3 Summary

The main research questions of this thesis were how to compute and process the ever increasing data, visualize the results without losing details and how to interact with the visualization efficiently and in a user friendly way. In this context, specific solutions were proposed for computation and processing of the data, for distributed visualization with scenes containing transparency and novel user interaction concepts with the focus on mobile devices.

DIANA, a computation hardware abstraction layer, was presented (cf. Chapter 3) which was developed with the goals in mind to have a middleware with low performance impact, which is easy to integrate into existing applications, and which can easily be extended to future APIs. The core of DIANA is a database which provides interfaces to the devices and commands. The interfaces reside in run-time loadable plugins, allowing the user to easily replace them with other versions, e.g. when the underlying hardware changes or new features have to be added. The software itself is linked against the DIANA core library and thus requires no modification or recompilation. Evaluation for standard GPU performance computations have shown a very low overhead and DIANA has successfully been used for a distributed live computation of an FTLE field on the VISUS powerwall.

When distributed systems are used for simulations or calculations of decomposed objects, the final visualization is challenging, especially when semi-

transparent sections are present. For rendering these scenes, the previously published PPLLs approach has been extended to distributed rendering (cf. Chapter 4). It is able to collect all fragments in a scene during rendering and the final composition happens after the rendering process when the scene is to be displayed. The PPLLs mechanism can also be used for advanced rendering methods like CSG or depth of field effects. The method was evaluated using protein surface rendering and proved to be comparable to current OIT rendering algorithms. Additionally, a distributed mesh renderer and a comparative multi-variant visualization application make use of the PPLLs approach. They demonstrate the capability for distributed remote rendering and the artifact-free handling of semi-transparent, overlapping objects.

An extension to the PPLLs approach is the PVLLs method (cf. Chapter 5). It uses the PPLLs storage mechanism to store all fragments of a given scene. The PVLLs is a generic rendering framework which can visualize meshed data, volumes, and implicit surfaces using voxelization. The voxels are generated by rendering the scene from three orthographic projections, effectively generating a sparse volume with the voxels containing the fragment data. Therefore, it can also be used for volume visualization of sparse data when the full volume data set, including all surrounding cells, does not fit onto the GPU. Having all fragments available also enables global lighting effects like reflection, refraction, and shadows. This is achieved by using ray tracing with ray splitting when traversing the voxels within the volume. These capabilities turn the system into a generic rendering framework.

Using a large system with multiple display nodes is challenging with respect to user interaction. The distributed architecture requires special applications to control the environment. To give the user an easy to use and intuitive interface, mobile device interaction concepts were proposed (cf. Chapter 6). The mobile device joins the system as an additional device with display and input capabilities. The images for the mobile device can be rendered locally or by a remote system, e.g. the render cluster also driving the display wall. This makes it possible to use different visualizations or add annotations with additional information. Inputs on the mobile device can be forwarded to the visualization system, making direct inputs possible by the user. The system can be used e.g. for research, collaboration, or presentation scenarios.

An integrated system was proposed (cf. Section 7.1) which features DIANA as a common interface. The visualization is driven by the PPLLs method, allowing for distributed semi-transparent rendering. Interactions can take place for example using an operator node or using the mobile devices. Although the single components did arise from different motivations and were originally based on different assumptions, a system can be created where these components can

be integrated as sub-systems. This system is flexible, extensible, and offers its services to a broad spectrum of devices, let it be mobile devices, standard computers or workstations, or large display walls.

Bibliography

B. Badillo, D. A. Bowman, W. McConnel, T. Ni, and M. G. da Silva. Literature survey on interaction techniques for large displays. Technical Report TR-06-21, Virginia Polytechnic Institute and State University, 2006. 111

R. Ball, C. North, and D. Bowman. Move to improve: promoting physical navigation to increase user performance with large displays. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 191–200, 2007. 110

M. Balsa Rodríguez, E. Gobbetti, J. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter. State-of-the-art in compressed GPU-based direct volume rendering. *Computer Graphics Forum*, 33(6):77–100, 2014. 90

A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7, 2010. 24

L. Bavoil and E. Enderton. *Constant-Memory Order-Independent Transparency Techniques*. NVIDIA Corp., 2011. 44

L. Bavoil and K. Myers. *Order Independent Transparency with Dual Depth Peeling*. NVIDIA Corp., 2008. 43, 44, 59

L. Bavoil, S. P. Callahan, A. Lefohn, J. L. D. Comba, and C. T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Symposium on Interactive 3D graphics and games*, pages 97–104, 2007. 44, 59

H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne. The Protein Data Bank. *Nucl. Acids Res.*, 28:235–242, 2000, http://www.pdb.org. 73, 75

J. Beyer, M. Hadwiger, and H. Pfister. A Survey of GPU-Based Large-Scale Volume Visualization. In *EuroVis - STARs*, pages 105–123, 2014. 90

J. Blinn. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph.*, 1: 235–256, 1982. 69

M. Bourgoin, E. Chailloux, and J.-L. Lamotte. Efficient abstractions for GPGPU programming. *International Journal of Parallel Programming*, 42(4):583–600, 2014. 25

G. Bradski. The OpenCV library. *Dr. Dobb's Journal of Software Tools*, 2000. 115

T. Brady, J. Dongarra, M. Guidolin, A. Lastovetsky, and K. Seymour. SmartGridRPC: the new RPC model for high performance grid computing. *Concurrency and Computation: Practice and Experience*, 22(18):2467–2487, Dec. 2010. 25

K. Bürger, J. Krüger, and R. Westermann. Sample-based surface coloring. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):763–776, 2010. 90

K. Brodlie, D. A. Duce, J. R. Gallop, J. P. R. B. Walton, and J. Wood. Distributed and collaborative visualization. *Computer Graphics Forum*, 23(2):223–251, 2004. 17

K. Brodlie, J. Brooke, M. Chen, D. Chisnall, A. J. Fewings, C. Hughes, N. W. John, M. W. Jones, M. Riding, and N. Roard. Visual supercomputing: Technologies, applications and challenges. *Computer Graphics Forum*, 24(2):217–245, 2005. 17

A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18:1–33, Jan. 2010. 23

P. Brown. *OpenGL Extension: NV_shader_buffer_store*. NVIDIA Corp., 2012. 45

P. Brown, C. Dodd, M. Kilgard, and E. Werness. *OpenGL Extension: NV_shader_buffer_load*. NVIDIA Corp., 2010. 45

L. Carpenter. The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, 1984. 45, 88

G. Chen, P. V. Sander, D. Nehab, L. Yang, and L. Hu. Depth-presorted triangle lists. *ACM Trans. Graph.*, 31(6):160:1–160:9, Nov. 2012. 43

K. Cheng and D. Zhu. Tablet interaction techniques for viewport navigation on large displays. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, pages 2029–2034. ACM, 2014. 111

A. Cockburn, A. Karlson, and B. B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Comput. Surv.*, 41(1):2:1–2:31, Jan. 2009. 4, 111

M. L. Connolly. Analytical Molecular Surface Calculation. *J. Appl. Cryst.*, 16: 548–558, 1983. 70

C. Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, Universite de Grenoble, 2011. 89

R. Dolbeau. HMPP™ : A hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, page 1–5. 2007. 23

H. Doleisch. *Visual Analysis of Complex Simulation Data using Multiple Heterogenous Views*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2004. 120

B. Domonkos and G. Jakab. A programming model for GPU-based parallel computing with scalability and abstraction. In *Proceedings of the 25th Spring Conference on Computer Graphics (SCCG)*, pages 115–122, Apr. 2009. 24

J. Dongarra. *Trip Report to Changsha and the Tianhe-2 Supercomputer*. Oak Ridge National Laboratory, 2013. 2

R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics (Proceedings of SIGGRAPH 1988)*, 22(4):65–74, 1988. 89

J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing and Simulation (HPCS)*, pages 224–231. IEEE, July 2010. 24

H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, 1994. 70, 71

S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, June 2009. 17

E. Eisemann and X. Décoret. Single-pass GPU solid voxelization and applications. In *Graphics Interface (GI 2008)*, volume 322, pages 73–80, 2008. 89

M. Eissele, D. Weiskopf, and T. Ertl. Interactive context-aware visualization for mobile devices. In *SG '09: Proceedings of Smart Graphics*, pages 167–178, 2009. 111

C. Everitt. *Interactive Order-Independent Transparency*. NVIDIA Corp., 2001. 44

M. Falk, M. Krone, and T. Ertl. Atomistic visualization of mesoscopic whole-cell simulations using ray-casted instancing. *Computer Graphics Forum*, 32(8): 195–206, 2013. 88, 92

S. Fang and D. Liao. Fast CSG voxelization by frame buffer pixel mapping. In *IEEE Symposium on Volume Visualization*, pages 43–48, 2000. 89

M. Üffinger, S. Frey, and T. Ertl. Interactive high-quality visualization of higher-order finite elements. *Computer Graphics Forum*, 29:337–346, 2010. 15

M. Fife, F. Strugar, and L. Davies. Whitepaper: Adaptive Volumetric Shadow Maps. Technical report, Intel, 2013. 45

J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman, 1990. 43

S. Frey and T. Ertl. PaTraCo: a framework enabling the transparent and efficient programming of heterogeneous compute networks. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV10)*, pages 131–140, 2010. 24

S. Frey, F. Sadlo, and T. Ertl. Explorable volumetric depth images from raycasting. In *Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 123–130, Aug. 2013. 90

E. Gobbetti, F. Marton, and J. A. I. Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Visual Computer*, 24(7-9):797–806, 2008. 89

D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, 2007. 23

M. D. Godfrey and D. F. Hendry. The computer as von Neumann planned it. *IEEE Ann. Hist. Comput.*, 15(1):11–21, 1993. 9

S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012. 39

J. Greer and B. L. Bush. Macromolecular shape and surface maps by solvent exclusion. In *Proceedings of the National Academy of Science*, pages 303–307, 1978. 68

M. Gross and H. Pfister. *Point-Based Graphics*. Morgan Kaufmann Publishers Inc., 2007. 54

S. Grottel, M. Krone, C. Müller, G. Reina, and T. Ertl. MegaMol - a prototyping framework for particle-based visualization. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):201–214, Feb 2015. 16

S. Gumhold. Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings of VMV*, pages 245 – 252, 2003. 14, 67

P. Gupta, N. da Vitoria Lobo, and J. Laviola, JosephJ. Markerless tracking and gesture recognition using polar correlation of camera optical flow. *Machine Vision and Applications*, 24(3):651–666, 2013. 114

R. Hardy and E. Rukzio. Touch & interact: Touch-based interaction of mobile phones with displays. In *Proceedings of the 10th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '08, pages 245–254. ACM, 2008. 111

V. Havran, J. Zajac, J. Drahokoupil, and H.-P. Seidel. MPI Informatics Building model as data for your research. Research Report MPI-I-2009-4-004, MPI Informatik, Dec. 2009. 51, 81

J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, Aug. 2001. 114

M. Hoffmann and J. Kohlhammer. A generic framework for using interactive visualization on mobile devices. In *Intelligent Interactive Assistance and Mobile Multimedia Computing*, volume 53 of *Communications in Computer and Information Science*, pages 131–142. 2009. 112

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

R. Kähler, M. Simon, and H. Hege. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):341–351, 2003. 89

A. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *Journal of Graphics Tools*, 4(4):5–10, 1999. 89

A. Kaufman and E. Shimony. 3D scan-conversion algorithms for voxel-based graphics. In *Workshop on Interactive 3D Graphics*, pages 45–75, 1987. 89

D. Kauker, H. Sanftmann, S. Frey, and T. Ertl. Memory saving discrete fourier transform on GPUs. In *CIT*, pages 1152–1157. IEEE Computer Society, 2010. 7

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl. Evaluation of per-pixel linked lists for distributed rendering and comparative analysis. *Computing and Visualization in Science*, pages 111–121, 2013a. 5, 42

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl. Rendering molecular surfaces using order-independent transparency. In E. Association, editor, *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, volume 13, page 33–40, 2013b. 5, 42

J. Kehrer and H. Hauser. Visualization and visual analysis of multifaceted scientific data: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):495–513, 2013. 4, 75, 119

D. A. Keim, F. Mansmann, J. Schneidewind, J. Thomas, and H. Ziegler. Visual data mining. chapter Visual Analytics: Scope and Challenges, pages 76–90. Springer-Verlag, 2008. 119

T. K. Khan. A survey of interaction techniques and devices for large high resolution displays. In *Visualization of Large and Unstructured Data Sets - Applications in Geospatial Planning, Modeling and Engineering (IRTG 1131 Workshop)*, volume 19 of *OpenAccess Series in Informatics (OASIcs)*, pages 27–35, 2011. 111

P. K. Kitanidis. *Introduction to Geostatistics: Applications in Hydrogeology*. Stanford-Cambridge Program. Cambridge University Press, 1997. 105

G. Klein. *Visual Tracking for Augmented Reality*. PhD thesis, University of Cambridge, 2006. 115

V. Kämpe, E. Sintorn, and U. Assarsson. High resolution sparse voxel DAGs. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 32(3):101:1–101:13, July 2013, SIGGRAPH 2013. 89

A. Knoll, I. Wald, P. A. Navrátil, M. E. Papka, and K. P. Gaither. Ray tracing and volume rendering large molecular data on multi-core and many-core architectures. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, UltraVis '13, pages 5:1–5:8, 2013. 3

P. Knowles, G. Leach, and F. Zambetta. Efficient layered fragment buffer techniques. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, page 279–292. CRC Press, 2012. 45

Z. Koza, M. Matyka, S. Szkoda, and L. Miroslaw. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, 36(2):C219–C239, 2014. 90

M. Krone, K. Bidmon, and T. Ertl. Interactive visualization of molecular surface dynamics. *IEEE Trans. Vis. Comp. Graph.*, 15(6):1391–1398, 2009. 70, 71

M. Krone, S. Grottel, and T. Ertl. Parallel Contour-Buildup Algorithm for the Molecular Surface. In *IEEE Symposium on Biological Data Visualization (biovis'11)*, pages 17–22, 2011. 70

M. Krone, J. E. Stone, T. Ertl, and K. Schulten. Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *EuroVis Short Papers*, volume 1, pages 67–71, 2012. 14, 69

M. Krone, G. Reina, C. Schulz, T. Kulschewski, J. Pleiss, and T. Ertl. Interactive Extraction and Tracking of Biomolecular Surface Features. *Computer Graphics Forum*, 32(3):331–340, 2013. 119

M. Krone, M. Huber, K. Scharnowski, M. Hirschler, D. Kauker, G. Reina, U. Nieken, D. Weiskopf, and T. Ertl. Evaluation of visualizations for interface analysis of sph. In *EuroVis 2014 Short Papers*, volume 3, pages 109–113, 2014a. 7

M. Krone, D. Kauker, G. Reina, and T. Ertl. Visual analysis of dynamic protein cavities and binding sites. In *IEEE PacificVis - Visualization Notes*, volume 1, pages 301–305, 2014b. 5, 118

M. Krone, C. Müller, T. Ertl, D. Kauker, A. C. Silva, D. Salsa, M. Gräber, and M. Kallert. Remote rendering and user interaction on mobile devices for scientific visualization. In *International Symposium on Visual Information Communication and Interaction (VINCI)*, volume 8. ACM, 2015. 110

F. Lamberti and A. Sanna. A streaming-based solution for remote visualization of 3d graphics on mobile devices. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):247–260, Mar. 2007. 112

P. Laug and H. Borouchaki. Molecular Surface Modeling and Meshing. *Engineering with Computers*, 18(3):199–210, 2002. 70

M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988. 89

S. Lindholm, D. Jönsson, H. Knutsson, and A. Ynnerman. Towards data centric sampling for volume rendering. In *SIGRAD 2013*, pages 55–60, 2013. 48

N. Lindow, D. Baum, S. Prohaska, and H.-C. Hege. Accelerated Visualization of Dynamic Molecular Surfaces. *Computer Graphics Forum*, 29:943–952, 2010. 70

F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Efficient depth peeling via bucket sort. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 51–57. ACM, 2009. 44, 52, 59

A. Magni, D. Grewe, and N. Johnson. Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 66–75, New York, NY, USA, 2013. ACM. 39

M. Makhinya. *Performance Challenges in Distributed Rendering Systems*. PhD thesis, Institut für Informatik, Universität Zürich, 2012. 17

L. Marsalek, A. Dehof, I. Georgiev, H.-P. Lenhof, P. Slusallek, and A. Hildebrandt. Real-Time Ray Tracing of Complex Molecular Scenes. In *14th International Conference on Information Visualization (IV)*, pages 239–245, 2010. 70

C. Maughan and M. Wloka. Vertex shader introduction. Technical report, NVIDIA Corporation, 2001. 11

M. Maule, J. a. L. D. Comba, R. P. Torchelsen, and R. Bastos. Technical section: A survey of raster-based transparency techniques. *Comput. Graph.*, 35(6): 1023–1034, Dec. 2011. 43

D. C. McCallum and P. Irani. ARC-pad: Absolute+relative cursor positioning for large displays with a mobile touchscreen. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 153–156, New York, NY, USA, 2009. ACM. 111

A. Meligy. Parallel and distributed visualization: The state of the art. In *Computer Graphics, Imaging and Visualisation, 2008. CGIV '08. Fifth International Conference on*, pages 329–336, 2008. 17

C. Müller, G. Reina, and T. Ertl. The VVand: A two-tier system design for high-resolution stereo rendering. In *CHI POWERWALL 2013 Workshop*, 2013. 19

A. Moll, A. Hildebrandt, H.-P. Lenhof, and O. Kohlbacher. Ballview: A tool for research and education in molecular modeling. *Bioinformatics*, 22(3):365–366, 2006. 70

K. Moreland. *IceT Users' Guide and Reference Version 2.1*, volume Tech Report SAND 2011-5011. Sandia National Laboratories, Aug. 2011. 17

J. D. Mulder, J. J. v. Wijk, and R. v. Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119 – 129, 1999. 16

K. Museth. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics*, 32(3):27:1–27:22, 2013. 89

K. Myers and L. Bavoil. Stencil routed a-buffer. *ACM SIGGRAPH 2007 sketches*, 2007. 44

T. Ni, G. S. Schmidt, O. G. Staadt, M. A. Livingston, R. Ball, and R. May. A survey of large high-resolution display technologies, techniques, and applications. In *Proceedings of the IEEE Conference on Virtual Reality*, VR '06, pages 223–236, Washington, DC, USA, 2006. IEEE Computer Society. 111

M. Nießner, H. Schäfer, and M. Stamminger. Fast indirect illumination using layered depth images. *Visual Computer*, 26(6-8):679–686, 2010. 90

W. Nowak and A. Litvinenko. Kriging and spatial design accelerated by orders of magnitude: Combining low-rank covariance approximations with FFT-techniques. *Mathematical Geosciences*, 45(4):411–435, 2013. 105

NVIDIA Corp. *NVIDIA CUDA C Programming Guide*, 2014. 11

M. Oberhumer. LZO documentation, 2011. [Online]. Available: http://www.oberhumer.com/opensource/lzo/. 60

M. A. O'Neil and M. Burtscher. Floating-point data compression at 75 gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 7:1–7:7. ACM, 2011, ACM ID: 1964189. 59, 60

A. Panagiotidis, D. Kauker, S. Frey, and T. Ertl. DIANA: a device abstraction framework for parallel computations. In P. Iványi and B. Topping, editors, *Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Press, 2011. 5, 22

A. Panagiotidis, D. Kauker, F. Sadlo, and T. Ertl. Distributed computation and large-scale visualization in heterogeneous compute environments. In *2012 11th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 87 –94, June 2012. 5, 22

S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 5(3):238–250, 1999. 128

J. Parulek and I. Viola. Implicit Representation of Molecular Surfaces. In *IEEE Pacific Visualization Symposium*, pages 217–224, 2012. 70, 71, 72, 73

T. Porter and T. Duff. Compositing digital images. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, page 253–259. ACM, 1984. 45

F. Reichl, M. G. Chajdas, K. Bürger, and R. Westermann. Hybrid sample-based surface rendering. In *Vision, Modelling and Visualization (VMV)*, pages 47–54, 2012. 90

G. Reina. *Visualization of Uncorrelated Point Data*. PhD thesis, Visualization Research Center, University of Stuttgart, 2008. 14

G. Reina and T. Ertl. Hardware-accelerated glyphs for mono- and dipoles in molecular dynamics visualization. In *EuroVis05: IEEE VGTC Symposium on Visualization*, page 177–182, 2005. 67, 70

M. Repplinger, A. Löffler, D. Rubinstein, and P. Slusallek. DRONE: a flexible framework for distributed rendering and display. In *Proceedings of the 7th International Conference on Visual Computing 2009 (ISVC09).*, pages 975–986. Springer, 2009. 17

F. M. Richards. Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering*, 6(1):151–176, 1977. 68

C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. pages 233–248. Symposium on Operating Systems Principles (SOSP), 2011. 25

P. Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics (Proceedings of SIGGRAPH 1988)*, 22(4):51–58, 1988. 89

F. Sadlo and R. Peikert. Efficient visualization of lagrangian coherent structures by filtered AMR ridge extraction. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1456–1463, 2007. 31

F. Sadlo, M. Üffinger, C. Pagot, D. Osmari, J. L. D. Comba, T. Ertl, C.-D. Munz, and D. Weiskopf. Visualization of cell-based higher-order fields. *Computing in Science & Engineering*, 13(3):84–91, 2011. 15

M. Salvi, J. Montgomery, and A. Lefohn. Adaptive transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 119–126. ACM, 2011. 45, 90

M. F. Sanner, A. J. Olson, and J.-C. Spehner. Reduced Surface: An efficient way to compute molecular surfaces. *Biopolymers*, 38(3):305–320, 1996. 70, 71

J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 231–242. ACM, 1998. 90

S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics*, VG'05, pages 187–195, 2005a. 14

S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware–based raycasting. In *Eurographics/IEEE VGTC Workshop on Volume Graphics*, pages 187–195, 2005b. 89, 98

M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: compute unified device and systems architecture. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV08)*, pages 49–56, 2008. 24

C. Teitzel, M. Hopf, R. Grosso, and T. Ertl. Volume visualization on sparse grids. *Computing and Visualization in Science*, 2:47–59, 1999. 89

J. Thomas and J. Kielman. Challenges for Visual Analytics. *Information Visualization*, 8(4):309–314, 2009. 119

M. Totrov and R. Abagyan. The contour-buildup algorithm to calculate the analytical molecular surface. *Journal of Structural Biology*, 116:138–143, 1995. 70

A. Varshney, F. P. Brooks, and W. V. Wright. Linearly Scalable Computation of Smooth Molecular Surfaces. *IEEE Computer Graphics and Applications*, 14(5): 19–25, 1994. 70, 71

A. A. Vasilakis and I. Fudos. Z-fighting aware depth peeling. In *ACM SIGGRAPH 2011 Posters*, SIGGRAPH '11, pages 77:1–77:1. ACM, 2011. 44

I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 70

A. Wessels, M. Purvis, J. Jackson, and S. Rahman. Remote data visualization through WebSockets. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 1050–1051, Apr. 2011. 112

L. Westover. Footprint evaluation for volume rendering. SIGGRAPH '90, pages 367–376. ACM, 1990. 63

T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003. 117

M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *Parallel and Distributed Systems, IEEE Transactions on*, 4(9):979–993, Sept. 1993. 128

C. M. Wittenbrink. R-buffer: a pointerless a-buffer hardware architecture. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 73–80, 2001, ACM ID: 383529. 45

H. Wright, R. H. Crompton, S. Kharche, and P. Wenisch. Steering and visualization: Enabling technologies for computational science. *Future Generation Computer Systems*, 26(3):506 – 513, 2010. 16

R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *Computer Graphics and Applications, IEEE*, 12(5):19–28, 1992. 89

J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum*, 29(4):1297–1304, 2010. 42, 45, 53, 59, 87

N. Zhang. Memory-hazard-aware k-buffer algorithm for order-independent transparency rendering. *IEEE Transactions on Visualization and Computer Graphics*, 20(2):238–248, 2014. 44

Y. Zhang and R. Pajarola. Deferred blending: Image composition for single-pass point rendering. *COMPUTER & GRAPHICS*, 31:175–189, 2007. 43