Strategies For Efficient Parallel Visualization

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

Vorgelegt von

Steffen Daniel Frey

aus Göppingen

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. Kwan-Liu Ma
	Prof. Dr. Dirk Pflüger
Tag der mündlichen Prüfung:	7. November 2014

Visualisierungsinstitut
der Universität Stuttgart

2014

Acknowledgments

First of all, I especially want to thank my advisor Thomas Ertl, who guided and supported my work and gave me the opportunity to pursue my doctoral studies at VISUS. I am also grateful to Kwan-Liu Ma and Dirk Pflüger for agreeing to act as referees for this thesis.

I had the great pleasure to collaborate with all these nice people over the recent years: Marco Ament, Sven Bachthaler, Carsten Dachsbacher, Oliver Deussen, Oliver Fernandes, Sebastian Grottel, Daniel Kauker, Christoph Müller, Alexandros Panagiotidis, Guido Reina, Filip Sadlo, Harald Sanftmann, Thomas Schlömer, Magnus Strengert, Markus Üffinger, and Daniel Weiskopf (in alphabetical order). I also greatly enjoyed the time with my office mates Christoph Müller, Gregor Mückl, Gustavo Mallo Machado, and Sebastian Boblest.

I would also like to thank the remaining members of VIS and VISUS for the great time. Many thanks go to Marcel Hlawatsch, Guido Reina, Filip Sadlo, and Markus Üffinger, for proof-reading parts of my thesis. Moreover, I would like to thank the students I supervised and with whom I worked with over the last years.

Financial support was provided by the state of Baden Württemberg. Thanks to Kwan-Liu Ma for organizing my research stay at the University of California in Davis, USA, and to the Cluster of Excellence in Simulation Technology at the University of Stuttgart (EXC 310/1) for financing it. I want to thank the whole ViDi lab in Davis, and in particular my roommate Nate, for an awesome time.

Above all, I want to express my deepest gratitude to Mirjam, as well as my family, Sandra, Anton, and Liane, for their support and for putting up with me during this work.

Steffen Frey

Table of Contents

Glossary

API application programming interface. 147, 162, 163

APOD Assess, Parallelize, Optimize, Deploy. 157, 158

application programming interface Specifies how some software components should interact with each other. 147, 162, 163

ASID Assess, Select, Implement, Deploy. 157, 158

Assess, Parallelize, Optimize, Deploy Application optimization process for Graphics Processing Units (GPUs) proposed by NVIDIA Corporation [2013b]. 157, 158

Assess, Select, Implement, Deploy Cyclic development model built around the strategy tree. It is based on the Assess, Parallelize, Optimize, Deploy (APOD) scheme. 157, 158

CCVD Capacity-constrained Voronoi Diagrams. 43, 46, 47, 51, 52

cluster set of loosely or tightly connected computers that work together so that in many respects they can be viewed as a single system. 7, 10, 19, 21, 40, 105, 163

Compute Unified Device Architecture A parallel computing platform and programming model created by NVIDIA. 157, 161, 163

CPU Central Processing Unit. 7, 9, 10, 12, 16–18, 20, 21, 151, 152, 161

CT computed tomography. 24, 60, 117, 118, 123

CUDA Compute Unified Device Architecture. 157, 161, 163

data parallelism each processor performs the same task on different chunks of distributed data. 11, 24

DFT Discrete Fourier Transform. 161

distributed memory processors have their own local memory. Information is exchanged by passing messages using communication links. 12

General-Purpose computing on Graphics Processing Units Using GPUs for general-purpose scientific and engineering applications. 18, 166

GPGPU General-Purpose computing on Graphics Processing Units. 18, 166

GPU Graphics Processing Unit. 7–9, 16–21, 24, 28, 29, 33, 44, 46, 49–52, 55, 59, 60, 64, 105, 108, 117, 118, 123, 125–127, 138, 157–161, 163, 167

high-performance visualization High-performance visualization focuses on the subset of scientific visualization concerned with algorithm design, implementation, and optimization for use on today's computational platforms. 157, 164, 166, 168

HPV high-performance visualization. 157, 164, 166, 168

instruction-level parallelism measure of the average number of instructions in a program that, in theory, a processor might be able to execute at the same time. This is mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions. Typically, it means extracting parallelism from a single instruction stream working on a single stream of data. 17

k-d tree A space-partitioning binary tree for organizing points in a k-dimensional space. 30, 44, 45, 48–50, 61, 160

LCCVD Loose Capacity-constrained Voronoi Diagrams. 46–49, 51–55, 57, 141

LDI Layered Depth Image. 30, 88, 95, 97, 98, 100–102

level of detail Decrease the complexity of a 3D object representation as it with respect to metric such as distance from the viewer or object importance. 29, 31

LOD level of detail. 29, 31

MC Marching Cubes. 26, 27, 100–102

MIMD Multiple Instruction, Multiple Data. 8–11

MISD Multiple Instruction, Single Data. 8, 9

MRT magnetic resonance tomography. 24

MS Marching Squares. 38

Multiple Instruction, Multiple Data Every processor executes different instruction and data streams. 8–11

Multiple Instruction, Single Data Only a single data stream is fed into multiple processing units, whereas each processing unit operates on the data independently via separate instruction streams. 8, 9

nonuniform memory access While memory is uniformly addressable from all processors like in SMP, some blocks of memory may be physically more closely associated with some processors than others. 12

NUMA nonuniform memory access. 12

octree Each internal node of this tree tree data structure has exactly eight children. They are typically used to partition a three dimensional space by recursive subdivision into eight octants. They can bee seen as the three-dimensional analog of quadtrees. 29

SDK software development kit. 162

shared memory processors have direct access to the same memory. 12

SIMD Single Instruction, Multiple Data. 8, 9, 11

SIMT Single Instruction, Multiple Thread. 8, 9, 14, 19, 24, 28, 126

Single Instruction, Multiple Data All processing units execute the same instruction at any given clock cycle, while each can operate on a different data element. 8, 9, 11

Single Instruction, Multiple Thread The SIMT architecture is akin to Single Instruction, Multiple Data (SIMD) in that a single instruction controls multiple processing elements. A key difference is that SIMD expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. 8, 9, 14, 19, 24, 28, 126

Single Instruction, Single Data One stream of instructions processes a single data stream. 8

SISD Single Instruction, Single Data. 8

SM Streaming Multiprocessor. 18, 19, 44–46, 50, 55, 134

SMP symmetric multiprocessors. 12

strategy In the context of this work, a strategy describes a way to exploit potentials for performance improvement in the field of high-performance visualization (HPV). 158, 167

strategy tree Hierarchical organization of strategies toward HPV. 2, 4, 155, 157–159, 164, 166–168

Streaming Multiprocessor Each NVIDIA GPU contains a couple of streaming multiprocessors. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. 18, 19, 44–46, 50, 55, 134

symmetric multiprocessors All processors share a connection to common memory and all memory locations can be accessed at equal speeds. 12

task A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. 3, 24, 35–37, 147, 148, 152

task item Task items are subdivisions of tasks. When applying a task to solve a specific problem, task items are generated from a specific task, that collectively solve that problem. Executing task items belonging to the same task in parallel is depicted as data-parallel in the following, if the task items executed in parallel belong to different tasks it is referred to as task-parallelism. 21, 35, 37, 38, 45, 46, 50, 126–134, 136, 137, 139, 147–152

task parallelism each processor performs a different task. 11, 24

TIC task item context. 130–132

VDI Volumetric Depth Image. 88–94, 164

volume raycasting Image-based volume rendering technique accumulating values along rays from a virtual camera through a volume. The goal is to determine radiance leaving the volume for each pixel on the image plane. In contrast to raytracing, secondary rays, i.e., reflection, refraction, and shadow rays, are not considered. Additionally, the volume has to be sampled along each ray since there is no explicit geometry given to perform ray-geometry intersections inside the volume. Sampling along the viewing rays allows to directly evaluate the discretized volume rendering integral given in (2.17) . 4

warp When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps (or waves in AMD terminology) and each warp individually gets scheduled for execution. Threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. 9, 14, 15, 17–20, 44, 46, 127, 129–135, 161

Abstract

Visualization is a crucial tool for analyzing data and gaining a deeper understanding of underlying features. In particular, interactive exploration has shown to be indispensable, as it can provide new insights beyond the original focus of analysis. However, efficient interaction requires almost immediate feedback to user input, and achieving this poses a big challenge for the visualization of data that is ever-growing in size and complexity. This motivates the increasing effort in recent years towards high-performance visualization using powerful parallel hardware architectures.

The analysis and rendering of large volumetric grids and time-dependent data is particularly challenging. Despite many years of active research, significant improvements are still required to enable the efficient explorative analysis for many use cases and scenarios. In addition, while many diverse kinds of approaches have been introduced to tackle different angles of the issue, no consistent scheme exists to classify previous efforts and to guide further development.

This thesis presents research that enables or improves the interactive analysis in various areas of scientific visualization. To begin with, new techniques for the interactive analysis of time-dependent field and particle data are introduced, focusing both on the expressiveness of the visualization and on a structure allowing for efficient parallel computing. Volume rendering is a core technique in scientific visualization, that induces significant costs. In this work, approaches are presented that decrease this cost by means of a new acceleration data structure, and handle it dynamically by adapting the progressive visualization process on-the-fly based on the estimation of spatio-temporal errors. In addition, view-dependent representations are presented that both reduce the size and render cost of volume data with only minor quality impact for a range of camera configurations. Remote and in-situ rendering approaches are discussed for enabling the interactive volume visualization without having to move the actual volume data. In detail, an approach for the integrated adaptive sampling and compression is introduced, as well as a technique allowing for user prioritization of critical results. Computations are further dynamically redistributed to reduce load imbalance. In detail, this encompasses the tackling of divergence issues on GPUs, the adaptation of volume data assigned to each node for rendering in distributed GPU clusters, and the detailed consideration of the different performance characteristics of the components in a heterogeneous system.

From these research projects, a variety of generic strategies towards high-performance visualization is extracted, ranging from the parallelization of the program structure and algorithmic optimization, to the efficient execution on parallel hardware architectures. The introduced strategy tree further provides a consistent and comprehensive hierarchical classification of these strategies. It can provide guidance during development to identify and exploit potentials for improving the performance of visualization applications, and it can be used as expressive taxonomy for research on high-performance visualization and computer graphics.

German Abstract – Zusammenfassung

Visualisierung ist grundlegend für die Analyse von Daten und die Gewinnung von Einsichten in zugrundeliegende Prozesse. Insbesondere die interaktive Exploration hat sich hierbei als wertvoll erwiesen, da sie den Erwerb neuer Erkenntnisse jenseits des ursprünglichen Analysefokus erlaubt. Für die effiziente Interaktion werden jedoch schnelle Antwortzeiten auf Benutzereingaben benötigt. Diese zu gewährleisten stellt eine große Herausforderung für die Visualisierung von Daten dar, die sowohl in Größe als auch Komplexität stetig zunehmen. Dies ist die Motivation für die zunehmenden Bemühungen in den letzten Jahren hinsichtlich High Performance Visualisierung und dem Einsatz von leistungsfähigen parallelen Hardwarearchitekturen.

Die Analyse und das Rendering von großen Volumen und zeitabhängigen Datensätzen stellt hier eine besondere Herausforderung dar. Trotz vieler Jahre aktiver Forschung müssen für viele Anwendungsszenarien auch heute noch maßgeschneiderte Ansätze entwickelt werden, um die effiziente explorative Analyse zu ermöglichen. Obwohl verschiedene Lösungsansätze für unterschiedliche Teilbereiche vorgestellt wurden, existiert bislang kein konsistentes Schema, das die verschiedenen Arbeiten klassifizieren, und somit auch Neu- sowie Weiterentwicklungen unterstützen kann.

In dieser Dissertation werden Forschungsarbeiten vorgestellt, die die interaktive Analyse in verschiedenen Bereichen der wissenschaftlichen Visualisierung verbessern, oder gar erst ermöglichen. Es werden neue Techniken für die interaktive Analyse von zeitabhängigen Feld- und Partikeldaten eingeführt, die den Fokus sowohl auf eine hohe Aussagekraft der Visualisierung als auch eine parallelisierungsfreundliche Programmarchitektur legen. Hierbei kommt auch Volumenrendering zum Einsatz, eine grundlegende Technik in der wissenschaftlichen Visualisierung, die insbesondere bei größeren Datensätzen erhebliche Kosten verursacht. Die vorliegende Arbeit zeigt Ansätze, die diese Kosten zum einen durch die Einführung neuer Beschleunigungsdatenstrukturen verringern, und sie zum anderen durch die fehlerschätzungsbasierte Steuerung des progressiven Visualisierungsprozesses dynamisch anpassen. Zudem werden ansichtsabhängige Repräsentation vorgestellt, die sowohl die Datengröße als auch die Bilderzeugungskosten von Volumendaten reduzieren, und dabei ledigliche geringe Qualitätseinbußen für einen weiten Bereich von Kameraeinstellungen verursachen. Außderdem werden Remote und In-Situ Ansätze diskutiert, die interaktive Volumenvisualisierung ermöglichen ohne Volumendaten übertragen zu müssen. Hierfür werden unter anderem sowohl adaptives Sampling und Kompression integrativ gehandhabt, als auch die Priorisierung von zeitkritischen Teilergebnissen ermöglicht. Die Berechnungen werden zudem dynamisch verteilt, um die Ungleichheit der Lastverteilung auszubalancieren. Dies beinhaltet die Behandlung von Divergenzproblemen auf GPUs, die Anpassung der Verteilung der Bilderzeugung in GPU Clustern, und die explizite Berücksichtigung von unterschiedlichen Performanzcharakteristiken verschiedener Komponenten in heterogenen Systemen.

Aus diesen Forschungsprojekten werden verschiedene Strategien zur High Performance Visualisierung extrahiert. Diese reichen von der Parallelisierung der Programstruktur über die Optimierung von Algorithmen zur effizienten Ausführung auf parallelen Hardwarearchitekturen. Diese Strategien werden in einen neu eingeführten, umfassenden und konsistenten hierarchischen Strategiebaum eingeordnet. Dieser soll als Hilfe dienen bei der Entwicklung neuer Ansätze durch die Unterstützung der Identifikation und Ausnutzung von Verbesserungspotentialen bei der Performanz von Visualisierungsanwendungen. Außerdem kann der Strategiebaum als aussagekräftige Taxonomie für Forschung und Entwicklung im Bereich von High Permance Visualisierung und Computergrafik verwendet werden.

CHAPTER 1

Introduction

1.1 Introduction and Motivation

Visualization graphically illustrates scientific data to enable the examination and the understanding of inherent structures and processes. Beyond the inspection of a static image, dynamic interaction with the visualization allows for an exploratory approach. Ultimately, this enables a user to gain insight beyond the original focus [Ferster, 2012]. Interactive exploration requires fast response times to deliver immediate feedback to parameter changes, like changing viewpoints. However, the size and complexity of the data sets that need to be analyzed are ever growing. This is mainly driven by the advent of new simulation and data acquisition methods as well as substantial increases in parallel hardware processing power. Alongside with this, the complexity of the visualization techniques themselves is also increasing. As a consequence, achieving interactivity without significantly compromising the quality of the visualization is often times very challenging. To avoid critical shortcomings, visualization applications need to efficiently exploit the large potential of parallel hardware to enable high-quality responsive interactive analysis for today's and future real-world data sets and visualization techniques. Approaches toward this goal are commonly classified as high-performance visualization techniques [Bethel et al., 2012].

This work pursues two main objectives. First, it aims to introduce techniques that individually help to solve specific problems in scientific visualization efficiently. Second, from these techniques, strategies are extracted to approach high-performance visualization in general. As the complete development and implementation process is known for these techniques, effort and innovation spent in the development process can be considered more truthfully than this would be the case for external research with merely the published results being available. In more detail, the contributions of this work are as follows:

- Novel techniques are introduced for interactive parallel visualization in a variety of areas, including
  - the analysis of time-dependent field and particle data (Ch. 3),
  - the reduction of render cost (Ch. 4),
  - view-dependent representations, (Ch. 5)
  - remote and in-situ rendering (Ch. 6),
  - and the tackling of GPU divergence and load imbalance on clusters (Ch. 7).
- We extract generic strategies from these approaches and organize them in a consistent and comprehensive structure, the strategy tree. Possible applications of this strategy tree are discussed in Ch. 8, both as
  - a guideline for the development of new parallel visualization techniques,
  - and as a taxonomy to classify and analyze research efforts in the field.

1.2 Basic Scope and Structure

There is a variety of different aspects to consider to create an insightful visualization application. Most prominently, expressive images need to be generated that allow to gain a deeper understanding of the data. Furthermore, the visualization application should also be accessible and intuitive to enable users to leverage the visualization application's full potential. To enable explorative data analysis, the application further needs to respond quickly to requests.

The focus of this work is on the latter aspect, i.e., on improving the responsiveness of (scientific) visualization applications to allow for prompt data analysis and quick (optimally interactive) iteration cycles. In practice, improving the responsiveness typically goes along with enhancing the efficiency of the visualization in the parallel environments that are omnipresent nowadays. Various projects tackling different aspects of this overall goal are presented in this work. From this, generic strategies toward *High-Performance Parallel Visualization* (Strat. 1) are extracted. These are classified into three fundamental strategies around the pivotal goal of *High-Performance Parallel Visualization* (Strat. 1), forming the basis of our strategy tree (Fig. 1.1). First, *Structure* (Strat. 2) basically encompasses the parallelization-friendly structuring of the application. Regarding the parallelization terminology used in this context, the different functionalities in which a technique can be partitioned are denoted as tasks. Tasks are further divided into task items, each of which perform the same programs on different data elements. Second, *Cost* (Strat. 3) comprises the reduction of cost induced by the visualization technique. Third, *Resources* (Strat. 4) contains strategies for the efficient handling and distribution of the load.

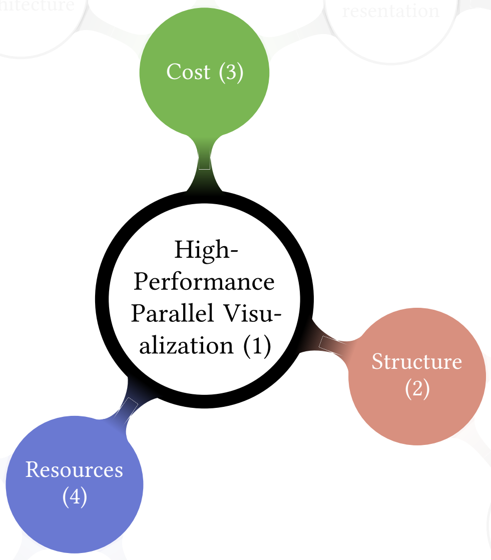


Figure 1.1: Categories of basic generic visualization strategies toward high-performance visualization.

Strategy 1 ***High-Performance Parallel Visualization***
Solve visualization problems using parallel computing such that the available resources are utilized efficiently to achieve responsive visual data exploration.

Strategy 2 ***Structure***
Structure an approach for good efficiency characteristics, with a focus on the best possible parallelization properties.

Strategy 3 ***Cost***
Minimize the overall work required to complete a task, i.e., reduce the amount, complexity or extent of tasks and their respective task items.

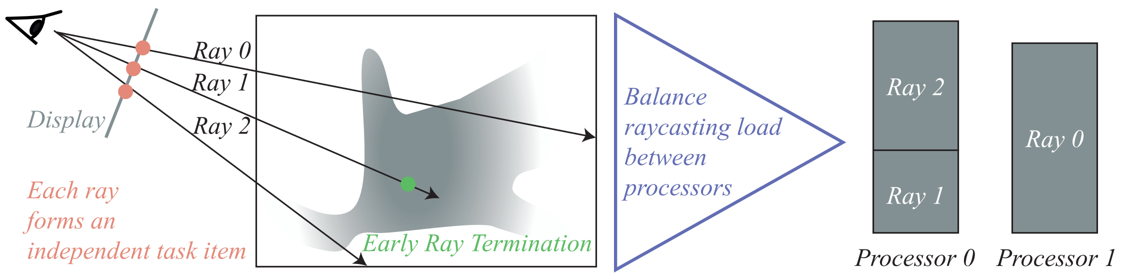
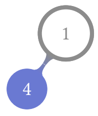

Ray 0
Ray 1
Ray 2
Display
Each ray forms an independent task item
Early Ray Termination
Balance raycasting load between processors
Ray 2
Ray 1
Ray 0
Processor 0
Processor 1

Figure 1.2: Illustration of fundamental strategy categories at the example of basic volume raycasting.

Strategy 4 ***Resources***
Optimize the investment of hardware resources to maximize key factors, like responsiveness, energy efficiency, etc., by adjusting the distribution, cost, and the number of task items, among others.



This basic segmentation of strategies toward *High-Performance Parallel Visualization* (Strat. 1) is primarily based on the experiences made during the development of the different approaches presented in this work. It also reflects separate subject areas tackled by different communities as well as the structure of parallel programming development guidelines [Mattson et al., 2004; NVIDIA Corporation, 2013b]. Ch. 8 elaborates on this in more detail.

A simple example demonstrating the basic strategies by means of volume raycasting is illustrated in Fig. 1.2. Here, a ray is sent through each pixel, accumulating color and opacity along its way through the volume. Rays can be processed independently from each other (Strat. 2, *Structure*). A simple technique to reduce the cost for each ray is early ray termination [Levoy, 1990c], i.e, the computation of the rays is stopped when the opacity is saturated (Strat. 3, *Cost*). Load balancing then can be employed to distribute the computation of different rays to processors such that the overall execution time is minimized (Strat. 4, *Resources*). Raycasting-related techniques are discussed in more detail in Sec. 2.2.

1.3 Document Organization

Ch. 2 covers the basics of parallel programming (Sec. 2.1) and visualization (Sec. 2.2) which are fundamental for the remainder of this work. In the following chapters 3 to 6, novel approaches for parallel visualization are discussed. Along the way, strategies are extracted from these approaches and added to the strategy tree. In detail, Ch. 3 introduces new techniques for the analysis of time-dependent scalar fields and the data reduction of results from molecular dynamics simulation. In this context, the focus lies on structuring the approach such that it has favorable properties when it comes to parallel computing. Ch. 4 presents techniques for reducing the computational effort of volume raycasting. Ch. 5 presents approaches for the view-dependent representation of volumes and their rendering. Ch. 6 discusses interactive in-situ volume generation and remote rendering. Ch. 7 introduces techniques for alleviating the load-imbalance occurring with parallel volume raycasting. An overview on the discussed projects is given at the beginning of the respective chapter. Ch. 8 discusses possible applications of the strategy tree that has been constructed along the presentation of the different research projects. In this chapter, also the properties, characteristics and limitations of the strategy tree are discussed, before finally concluding this work.

CHAPTER 2

BASICS

A major aspect of this work is the integration of parallel computing and scientific visualization. The fundamentals for both of these areas are covered in this chapter in Sec. 2.1 and Sec. 2.2, respectively.

2.1 Parallel Computing

Parallel computing denotes a form of computation in which many calculations are performed simultaneously. It operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. Kirk and Hwu [2012, chapter 10.1] give three goals for adopting parallel computing: solve a given problem in less time, solve bigger problems, and achieve better solutions for a given problem and a given amount of time. Applications that are good candidates for parallel computing typically involve large problem sizes and high computational cost, i.e., a large amount of data is processed, many iterations are performed on the data, or both. In this section, we discuss fundamental models of parallel computing (Sec. 2.1.1), modern hardware architectures (Sec. 2.1.2), and finally tools and frameworks that make them accessible to the programmer (Sec. 2.1.2). An overview on hardware architectures and some of their characteristics that are discussed in the following is given in Tab. 2.1.

	CPU	GPU	Cluster
Flynn's Taxonomy (Sec. 2.1.1)	MIMD (Shared Memory)	SIMD (SIMT)	MIMD (Distributed Memory)
Programming Model (Sec. 2.1.1)	Task & Data	Data (& Task)	Task & Data
Classification (Sec. 2.1.2)	Parallel, Multicore	Parallel, Manycore	Distributed
Programming (Sec. 2.1.2)	OpenMP, OpenCL	CUDA, OpenCL, GLSL	MPI, Sockets

Table 2.1: Overview on the characteristics and properties of different hardware architectures that are discussed in this section.

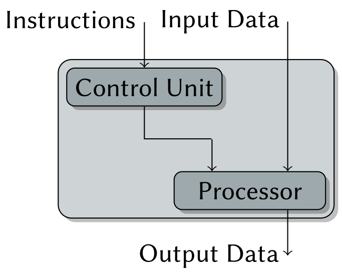


Figure 2.1: SISD: One stream of instructions processes a single data stream

2.1.1 Fundamentals

In the following, several fundamental models of parallel computing are discussed. Starting with Flynn's taxonomy, different models are outlined regarding programming, memory, and performance considerations.

Flynn's Taxonomy

Flynn's Taxonomy [Flynn, 1972] is the most widely used classification for parallel computers. It distinguishes multi-processor computer architectures with respect to their classification along the two independent dimensions of instruction (I) and data (D). Each of these dimensions has two possible states: single (S) and multiple (M). Thus, there are four possible classifications: SISD, MISD, SIMD, and MIMD. For SIMD, a variant called SIMT (T stands for threads) has been introduce to more adequately describe the architecture of modern GPUs. In visualization, SIMD (SIMT) and MIMD are by far the most commonly employed categories.

SISD simply depicts a serial (non-parallel) machine (Fig. 2.1). A single data stream is processed by one stream of instruction processes. In contrast, in MISD, such a single data stream is fed into multiple processing units, each of which operates on the

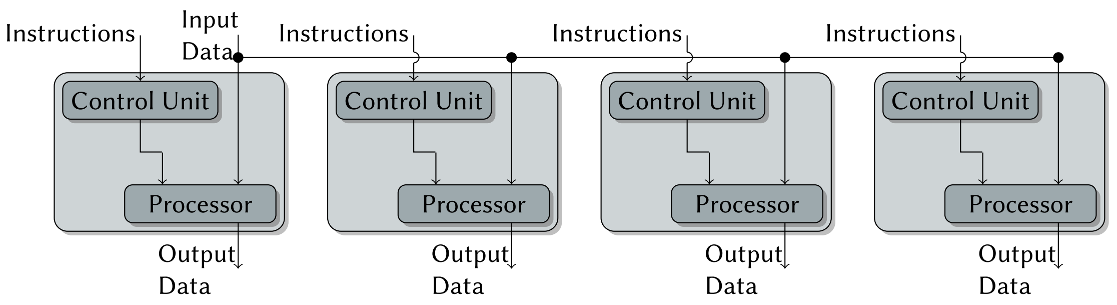


Figure 2.2: MISD: Only a single data stream is fed into multiple processing units, whereas each processing unit operates on the data independently via separate instruction streams

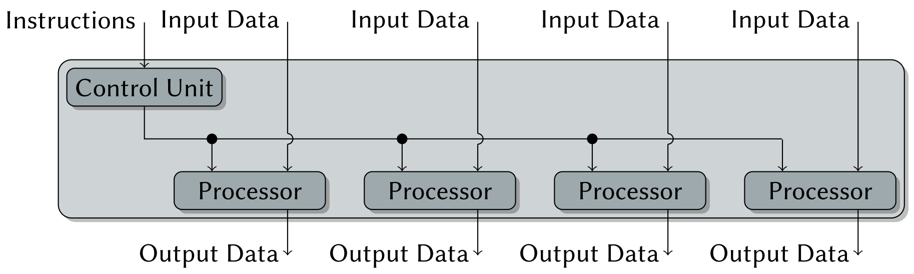
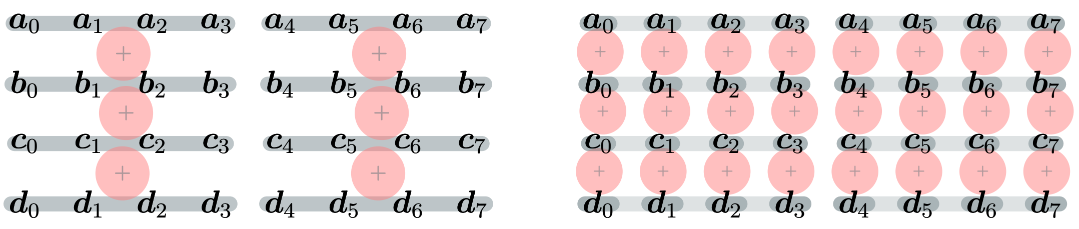


Figure 2.3: SIMD: All processing units execute the same instruction at any given clock cycle, while each can operate on a different data element

data independently via separate instruction streams (Fig. 2.2). In practice, only few implementations of such a machine haven been implemented. The most prominent application scenarios include space shuttle flight control, code cracking in cryptography, and fault-tolerant computers executing the same instructions redundantly.

In SIMD, all processing units execute the same instruction at any given clock cycle, while each can operate on a different data element (Fig. 2.3). Such machines exploit data level parallelism (discussed in detail below) and are best suited for problems with a high degree of regularity, which is typically the case for the processing of graphics, image, or audio data. Most modern CPU designs include SIMD instructions (e.g., MMX, SSE or AltiVec) to improve the performance of multimedia operations. Modern GPUs are also frequently classified as SIMD architectures. However, there are some differences to classical SIMD. The SIMT classification has been introduced to adequately address these [NVIDIA Corporation, 2013a].

SIMT is akin to SIMD in that a single instruction controls multiple processing elements. A key difference is that SIMD exposes the SIMD width to the software (Fig. 2.4(a)), whereas SIMT instructions specify the execution and branching behavior of a single thread (Fig. 2.4(b)). In SIMT, programmers basically write thread-level parallel code for independent, scalar threads (similar to MIMD that is explained in detail below). For the purpose of correctness, the underlying characteristics of SIMT can essentially be



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7
+

(a) SIMD: The SIMD-width of 4 explicitly needs to be considered, and the vectors are added up in chunks which were specified by the programmer.

(b) SIMT: Threads (dark gray) process one channel only, the underlying thread organization (light gray) does not need to be considered explicitly by a programmer.

Figure 2.4: Differences from a programmers perspective of SIMD and SIMT. In this example, four vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ and $\boldsymbol{d}$ are summed up, each of which features eight elements.

ignored, however, substantial performance improvements can be realized by explicitly taking them into account. Similar to SIMD, certain groups of threads (so-called warps) share the same instruction stream. Threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. Nevertheless, threads of a warp run in lockstep, i.e., if threads of a warp diverge via a data-dependent conditional branch, the warp is forced to serially execute each branch path taken, disabling threads that are not on that path. Thus, full efficiency is only realized when all threads of a warp agree on their execution path.

In MIMD, processors execute different instruction on different data streams independently (Fig. 2.5). MIMD machines belong either to the shared memory or the distributed memory category, depending on whether they share the same address space. In either case, processors can communicate over interconnects. Both multicore CPUs and compute clusters can be classified as MIMD.

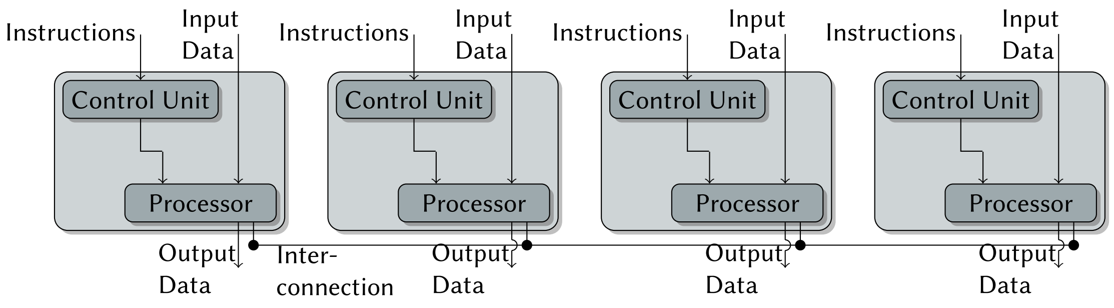


Figure 2.5: MIMD: Every processor executes different instruction and data streams

Programming Models

Algorithm 1 Data parallelism with data array d and two processors p_0 and p_1. Processors are assigned different data ranges $\{r_{\min}, \ldots, r_{\max}\}$ for which they evaluate a function f.

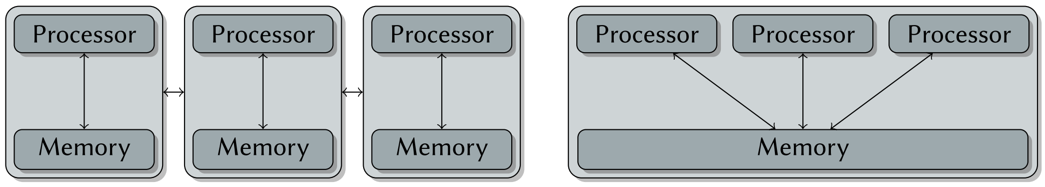
```
1: if $p_0$ then                              ▷ Data element assignment for processor 0
2:     $r_{\min} \leftarrow 0$
3:     $r_{\max} \leftarrow |d|/2$
4: if $p_1$ then                              ▷ Data element assignment for processor 1
5:     $r_{\min} \leftarrow |d|/2 + 1$
6:     $r_{\max} \leftarrow |d| - 1$
7: for $i \leftarrow r_{\min}$ to $r_{\max}$ do                  ▷ Perform task $f$ on every element of $d$
8:     $f(d[i])$
```

There are two basic forms of parallelization across multiple processors in parallel computing environments: data parallelism and task parallelism. In data parallelism, each processor performs the same task on different chunks of distributed data (Alg. 1). It is the most widely used model in visualization [Moreland, 2013]. The extent of concurrency is limited only by the number of pieces the data can be split into, which can be very high for large-scale data. It is well-suited for large, homogeneous problems, but may suffer from severe efficiency drops in the case of irregularity. Data parallelism is the typical model for SIMD but also works well on MIMD architectures. It has been shown to be very scalable on current supercomputers [Childs et al., 2010].

Algorithm 2 Task parallelism with tasks f and g, two processors p_0 and p_1, and data d.

```
1: if $p_0$ then                              ▷ Task assignment for processor 0
2:     $f(d)$
3: if $p_1$ then                              ▷ Task assignment for processor 1
4:     $g(d)$
```

In task parallelism, each processor performs a different task (Alg. 2), i.e., independent portions of a program are executed concurrently. Task parallelism can be applied to almost any type of algorithm and scale well even with irregular problems. However, there are typically significant practical limits on how much concurrency can be achieved with each processor performing a different task. In particular, visualization applications in real working environments can seldom be broken into more than a handful of independent parts [Moreland, 2013].



(a) Distributed memory: processors have their own local memory. Information is exchanged by passing messages using communication links

(b) Shared memory: processors have direct access to the same memory

Figure 2.6: Different memory models for hardware architectures with multiple processors.

Memory Models

The terms “parallel computing”, and “distributed computing” significantly overlap, and no clear distinction exists between them [Ghosh, 2007; Keidar, 2008]. However, popularly, parallel computing is regarded as a tightly coupled form of distributed computing [Peleg, 2000], and distributed computing may be seen as a loosely coupled form of parallel computing [Ghosh, 2007; Mattson et al., 1996; Adiga et al., 2002]. Accordingly, in parallel computing, all processors may have access to a shared memory to exchange information between processors (SIMD, MIMD with shared memory) [Keidar, 2008]. In distributed computing, each processor has its own private memory (MIMD, distributed memory) (Fig. 2.6). The combination of the two is commonly denoted as hybrid [Mattson et al., 1996].

In a shared memory system, all processes share a single address space and communicate with each other by writing and reading shared variables. This class is further decomposed into SMP and NUMA. In SMP, all processors share a connection to common memory and all memory locations can be accessed at equal speeds. SMP systems are arguably the easiest parallel systems to program, but the limited processor-to-memory bandwidth heavily impedes their scaling properties. As a result, SMP systems are typically restricted to small numbers of processors. Memory is also uniformly addressable from all processors in NUMA, but some blocks of memory may be physically more closely associated with some processors than others. This reduces the memory bandwidth bottleneck and allows systems with more processors. In addition, nearly all CPU architectures use a small amount of very fast non-shared (cache) memory to exploit locality in memory accesses, along with a protocol to keep cache entries coherent (this is also called cache coherent NUMA or ccNUMA). A NUMA system can be programmed in the same ways as an SMP system, but data locality and cache effects need to be taken into account to obtain the best performance.

In a distributed memory system, each process has its own address space and communicates with other processes by message passing (sending and receiving messages).

Typically, communication and data distribution need to be handled manually by the programmer, although experimental frameworks have been proposed that are able to automatically take care of this under certain circumstances (Sec. 2.1.2).

Performance Models

Simple analytic models for an idealized model problem can be used to illustrate some of the factors that influence the performance of a parallel program.

Processing. Let a computation consist of two tasks $\breve{t}_s$ and $\breve{t}_p$, with the total total run time for one processor $\breve{t}_\Sigma(1)$ given by

$$\breve{t}_\Sigma(1) = \breve{t}_s + \breve{t}_p \tag{2.1}$$

Now, there are p (identical) processors available for computation. Assuming that $\breve{t}_s$ can only be run serially (e.g., typical setup and finalization code) and solely $\breve{t}_p$ can be run in parallel, this yields

$$\breve{t}_\Sigma(p) = \breve{t}_s + \frac{\breve{t}_p}{p} \tag{2.2}$$

A widely used measure of how useful additional processors are is strong scaling. , i.e., the relative speedup s that is defined as follows: It denotes the speedup s for a varying number of processors and a fixed problem size:

$$s(p) = \frac{\breve{t}_\Sigma(1)}{\breve{t}_\Sigma(p)}. \tag{2.3}$$

Another widely used notion is weak scaling, in which the problem size assigned to each processor stays constant, and more processors are used to solve a larger overall problem. The goal is typically to achieve a perfect linear speedup (i.e., $s = p$ for strong scaling). Unfortunately, this can only rarely be achieved due to several factors.

One of these reasons hindering perfect scaling are serial parts of program that cannot be parallelized. In the following, σ is used to denote the ratio of the time spent in the serial parts with respect to the total run time.

$$\sigma = \frac{\breve{t}_s}{\breve{t}_\Sigma(1)} \tag{2.4}$$

With $(1 - \sigma)$ denoting the ratio of the parallel parts accordingly, plugging this into Eq. 2.2 yields:

$$\breve{t}_\Sigma = \sigma \breve{t}_\Sigma(1) + (1 - \sigma)\frac{\breve{t}_\Sigma(1)}{p}. \tag{2.5}$$

Finally, reformulating speedup s (Eq. 2.3) using Eq. 2.5 results in Amdahl's law, giving the maximum speedup that can be achieved with respect to serial computing by using p processors:

$$s(p) = \frac{\breve{t}_\Sigma(1)}{(\sigma + \frac{1-\sigma}{p})\breve{t}_\Sigma(1)} = \frac{1}{\sigma + \frac{1-\sigma}{p}}. \tag{2.6}$$

As a consequence, this means that the obtainable speedup is ultimately restricted by the serial part.

$$\lim_{p\to\infty} \frac{1}{\sigma + \frac{1-\sigma}{p}} = \frac{1}{\sigma} \tag{2.7}$$

Dependencies. Data dependencies can have a major impact on the computation time. No program can run more quickly than the longest chain of dependent calculations (also denoted as the critical path). Let $O_0 = f_0(I_0)$ and $O_1 = f_1(I_1)$ be two sequences of operations, performing computations on input data I and producing output data O. The conditions of Bernstein [1966] state that they are independent and can be processed in parallel if they satisfy the following:

$$I_1 \cap O_0 = \varnothing \tag{2.8}$$
$$I_0 \cap O_1 = \varnothing \tag{2.9}$$
$$O_0 \cap O_1 = \varnothing. \tag{2.10}$$

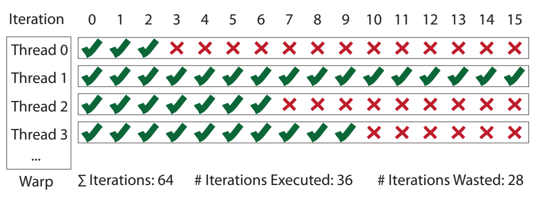
Violation of the first two input-output conditions (Eqs. 2.8 and 2.9) means that one sequence produces results that the other sequence depends on. Eq. 2.10 describes an output dependency, in that f_0 and f_1 write potentially different results to the same location.

Communication. In parallel systems, dependencies typically translate into communication effort. The total time for message transfer $\tilde{t}_\Sigma$ can be modeled as the sum of a fixed cost (latency $\tilde{t}_l$) plus a variable cost that depends on the size of the message n and the bandwidth b.

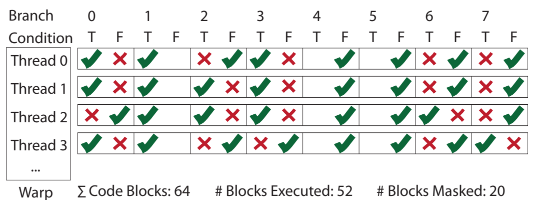
$$\tilde{t}_\Sigma = \tilde{t}_l + \frac{n}{b} \tag{2.11}$$

Latency $\tilde{t}_l$ is essentially the time it takes to send an empty message over the communication medium, from the time the send routine is called to the time the data is received. It includes overhead due to software and network hardware plus the time it takes for the message to traverse the communication medium. The bandwidth b is a measure of the capacity of the communication medium.

Load Imbalance. There are two basic causes for load imbalance in parallel applications [Yelick, 2007, chapter 10]: different processor performance and different load across processors. Different processor performance can not only occur due to heterogeneous hardware (in hybrid systems), but also stem from load on the respective



(a) *Termination Divergence:* All threads run until all threads of their warp are finished.



(b) *Branch Divergence:* All threads need to step through a branch if (at least) one thread of their warp needs to enter it.

Figure 2.7: Threads of a warp are executed in lockstep, resulting in termination divergence (a) and branch divergence (b), among others.

Spectrum	Easy	Harder	Hardest
Cost	Uniform	Different, but known	Unknown until after execution
Dependency	None	Predictable structure	Dynamically changing structure

Table 2.2: Different causes of load imbalance, and the difficulty of resolving them (based on Yelick [2007, chapter 10]).

machine. Different load can be due to variations in computation, dependencies and communication (Tab. 2.2). Different manifestations of this issue arise for different hardware architectures.

In SIMT, for instance, the negative impact of heterogeneous load is made even worse by the lockstep characteristic forcing threads of a warp to execute the same code path. This is commonly denoted as warp divergence and can be further classified with respect to the cause.

Memory Divergence Some threads of a warp perform costly memory accesses and stall, forcing the other threads to idle.

Termination Divergence Terminated threads waste compute cycles until all threads of their warp are finished with their computation steps (Fig. 2.7(a)).

Branch Divergence A warp serially executes each branch path taken, masking threads that are not on that path (Fig. 2.7(b)).

The effort and cost for resolving load imbalance depends on when certain information about the load balancing problem is available, how costly the acquisition of this information is, and how expensive it is to actually use this information (Tab. 2.2). Depending both on these costs and their effectiveness in reducing the imbalance, different approaches for load balancing can be useful. These approaches can be distinguished into centralized and distributed load balancing with respect to where they operate. They are further popularly classified according to when they are applied:

Static The schedule is determined before the execution. This requires all relevant information to be available beforehand.

Semi-Static The schedule is defined between major steps (well-defined points), e.g., at the beginning of each timestep or frame.

Dynamic The schedule is determined in mid-execution. It can thus utilize the latest information and react very flexibly to rapidly changing behavior.

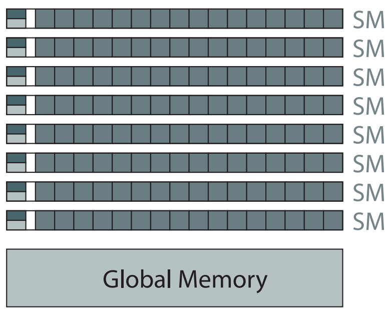
2.1.2 Parallel and Distributed Computing Systems

Traditionally, the vast majority of software applications are written as sequential programs. The performance of single core processors has grown several orders of magnitude over the past 25 years, driven by transistor speed and energy scaling, as well as by microarchitecture advances that exploited the transistor density gains from Moore's law. However, diminishing transistor-speed scaling and practical energy limits create new challenges for continued performance scaling. As a result, the frequency of operations only increases slowly, with energy being the major limiting factor [Borkar and Chien, 2011]. To address this, several variations of parallel designs have been introduced to still enable larger improvements in compute performance.

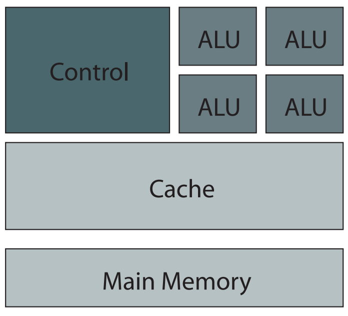
Shared Memory Systems: Multicore and Manycore

One way to increase the amount of work performed per clock cycle is to clone single cores multiple times on the chip [Gaster et al., 2011]. Two different kinds of design for parallel processors have evolved in the past years [Hwu et al., 2008]. The multicore (CPU) direction basically seeks to maintain the execution speed of sequential programs while moving into multiple cores (currently typically 2 to 16). In contrast, the manycore (GPU) direction focuses more on the execution throughput of parallel applications with thousands of cores. At the time of this writing, the peak double precision floating-point calculation throughput differs between manycore GPUs and multicore CPUs by about one order of magnitude. Currently, GPUs also have approximately six times the memory bandwidth of CPUs, which is particularly important in visualization as many typical applications are memory-bound (i.e., their speed is limited by data fetches from graphics memory).

The gap in floating-point performance stems from differences in the fundamental design philosophies of the two types of processors (Fig. 2.8). The design of a CPU is optimized



(a) GPU



(b) CPU

Figure 2.8: Schematic illustration of relative transistor expenditure. (a) The GPU is specialized for compute-intense, highly parallel computation. This means that more transistors are employed for data processing (b) rather than for data caching and flow control (based on NVIDIA Corporation [2013a, figure 7]).

for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. Additionally, large cache memories are provided to reduce the instruction and data access latencies of complex applications, with neither control logic nor cache memories contributing to the peak calculation speed. As the goal of this design is the minimization of the execution latency of a single thread, it is commonly denoted as latency-oriented design [Kirk and Hwu, 2012].

In contrast, the application software for manycore architectures like GPUs is expected to be designed with a large number of parallel threads. Instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading. However, unlike CPU cores, instructions are not issued out of order, and there is no branch prediction or speculative execution. A warp scheduler periodically selects a warp that is ready to execute its next instruction, if possible. Full utilization is achieved when all warp schedulers always have some instruction to issue for some warp. This allows to hide latency stemming from memory accesses or register dependencies (i.e., some of the input operands are written by some previous instruction(s) whose execution has not completed yet) [NVIDIA Corporation, 2013a, chapter 4]. To reduce memory access latency, small cache memories are employed such that multiple threads that access the same memory data do not all need to access graphics memory directly. This design style is commonly referred to as throughput-oriented design since it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute [Kirk and Hwu, 2012].

Thread Hierarchy Level	Grid	⊂ Thread Block	⊂ Warp	⊂ Thread
Processor Hierarchy Level	GPU	⊂ SM		⊂ Processor
Memory Hierarchy Level	Global Memory	+ Shared Memory		+ Private Memory
Execution Synchronization	Asynchronous		Synchronous	

Table 2.3: Organization and properties of GPU hierarchy levels.

As a result, for a low degree of parallelism, CPUs with lower operation latencies can achieve much higher performance than GPUs. However, for a high degree of parallelism, GPUs with higher execution throughput can achieve much higher performance than CPUs. As larger programs typically feature a combination of the two degrees of parallelism, typically a combination of the two is used. This work particularly focuses on GPUs, which in many cases are a good match for compute-intense visualization approaches.

GPUs as Generic Manycore Processors

Graphics chips started as fixed-function graphics processors but became increasingly programmable and computationally powerful. During the 90s, graphics processors moved beyond being just simple rasterizers by adding functionality in hardware that previously had to be done in software on the CPU. From 2000–2005, pixel and vertex shaders were introduced and iteratively improved with respect to both speed and flexibility. With these, looping and complex floating point math could be implemented, and certain operations, like the modifications of image-arrays, took significantly less time to execute on the GPU than on the CPU. Stream processors for generic applications have been introduced around 2006. Using GPUs for general-purpose scientific and engineering applications is commonly referred to as GPGPU.

Modern GPUs are organized as an array of Streaming Multiprocessors (SMs) (Fig. 2.8(a)). Each SM has a number of streaming processors that share control logic and instruction cache (Tab. 2.3). Memory is also organized hierarchically [NVIDIA Corporation, 2013a, chapter 2.3]. On the lowest level, each processor can access private local memory. In hardware, this typically maps to registers and provides storage space in the order of hundreds of bytes. Each SM further features so-called shared memory (tens of kilobytes) and all streaming processors have direct access to the same global memory (hundreds of megabytes to several gigabytes). Shared memory resides on-chip and is very fast, with access speeds comparable to registers [NVIDIA Corporation, 2013a]. In contrast, global memory is located off-chip. It provides high bandwidth, yet exhibits relatively high latency. However, for massively parallel applications, the higher bandwidth makes up for the longer latency. In addition to these types of memory that can be directly controlled via reads and writes by the processors, there are two read-only memory spaces accessible by all threads: the constant and texture memory spaces. In contrast to private and shared memory, the global, constant, and texture memory spaces are persistent across kernel launches by the same application. Texture memory further offers different addressing and filtering modes for a number of data formats. Note that depending on the implementation, these distinctions between memory spaces can be merely logical and not directly represented in hardware (i.e., constant memory uses the same on-chip memory as shared memory).

A task that the programmer wants to carry out on the GPU is implemented in a so-called kernel. As specified by the programmer, threads executing the kernel are grouped into thread blocks that are executed on the same SM. These thread blocks are organized in a so-called grid (Tab. 2.3). For execution, the respective blocks of the grid are enumerated and distributed on-the-fly to SMs with available execution capacity. Multiple thread blocks can execute concurrently on one SM. As thread blocks terminate, new blocks are launched on the vacated SMs [NVIDIA Corporation, 2013a, chapter 4]. For execution, a SM partitions thread blocks into warps, which are then scheduled individually. Note that the SIMT divergence issue discussed above occurs only within a warp, i.e., different warps execute independently regardless of whether they are executing common or disjoint code paths.

Distributed Memory Systems

There are numerous examples of distributed systems used in everyday life in a variety of applications, like the internet or peer-to-peer networks [Ghosh, 2007]. In the context of this work, we focus on compute clusters that have proven to be a good match for interactive visualization. A compute cluster is a set of loosely or tightly connected computers that work together so that in many respects they can be viewed as a single system. Homogeneous clusters consist of many of the same or similar types of machines while heterogeneous systems can offer a wide variation of nodes. The components of a cluster are usually connected to each other through fast network interconnects, with each node running its own instance of an operating system. All machines also typically share a common file system.

Apart from clusters, there is a variety of other approaches with a more distributed nature [Mattson et al., 1996; Adiga et al., 2002]. With the original idea being the linking of multiple supercomputers over wide area networks like the internet, grid computing has evolved into a general way to share heterogeneous resources, like compute time, storage, application servers, information services, or even scientific instruments [Foster and Kesselman, 2003]. Cloud computing can be seen as a specialized form of distributed computing that offers compute resources as a service, with the underlying hardware and software being largely abstracted from the consumer [Erl et al., 2013].

Programming Tools and APIs

The emergence of various novel parallel hardware architectures resulted in significantly improved efforts toward parallel program development. This has been referred to as the concurrency revolution [Sutter and Larus, 2005]. Numerous programming tools and APIs have been proposed to make parallelism accessible to the programmer, and alleviate issues such as load imbalance. Tools and frameworks are typically targeted at a specific kind of architecture.

Multi and Manycore Systems. OpenMP is a widely used API that supports multi-platform shared memory multiprocessing programming on a wide range of multi-processors and operating systems [OpenMP Architecture Review Board, 2011]. A programmer specifies directives (commands) and pragmas (hints) about a loop to the OpenMP compiler. With these, compilers generate parallel code that is executed with the OpenMP runtime transparently managing parallel threads and resources. While OpenMP was originally designed for CPU execution, a variation called OpenACC has been proposed for programming homogeneous and heterogeneous computing systems in general [OpenACC, 2013].

CUDA is an extension to the C programming language designed to take advantage of NVIDIA GPUs [NVIDIA Corporation, 2013a]. Similar to CUDA, OpenCL has been developed primarily for massively parallel processors. In contrast to CUDA, OpenCL is a standardized programming model, and applications developed in OpenCL can potentially run on any device if an implementation exists [Stone et al., 2010]. However, an applications typically needs to be modified manually to achieve high performance for a new processor [Kirk and Hwu, 2012]. Addressing this drawback, RapidMind can automatically divide computations among the available processing cores, if they are expressed as a sequence of functions applied to arrays [Christadler and Weinberg, 2010]. Similar to RapidMind, HMPP is a tool to enable device-transparent programming and execution of applications on machine level featuring multi-core processors [Dolbeau et al., 2007]. Both these frameworks base on a runtime system that provides a uniform execution model, scheduling policies and automated data transfers, as discussed by Augonnet et al. [2009] among others.

As discussed in Sec. 2.1.2, warp divergence can be a problem with GPUs, and several (mostly application-specific) approaches have been proposed to alleviate this issue. For tackling termination divergence, persistent threads have been introduced by Aila and Laine [2009] to deal with strongly varying iteration counts in ray tracing. Just enough threads are launched to occupy the hardware, and thread blocks are allowed to fetch new tasks from a task pool in global memory. Tzeng et al. [2010] also employ persistent threads to address irregular parallel work in their GPU task management system. In a GPU-based path tracer, Novák et al. [2010] dynamically generate new rays for terminated rays, thus allowing for a higher ray throughput and result in improved image quality overall. Han and Abdelrahman [2011] target the problem of divergent branches within loops and propose a software solution that only executes one branch per loop iteration, and discuss different strategies for selecting the active branch. Zhang et al. [2010] handle conditional branches by runtime data remapping between threads using a CPU-GPU pipelining scheme. Besides branches within loops, their approach can also handle differing loop iteration counts. Another line of research concerns itself with finding hardware solutions for the divergence problem. For ray tracing, Aila and Karras [2010] focus on handling incoherent rays efficiently. Meng et al. [2010] tackle cache-hit-induced divergence in memory access latency by utilizing several independent scheduling entities that allow divergent branch paths to interleave their execution. Fung et al. [2007] dynamically regroup threads into new warps when discovering a divergent branch. Other approaches for branch divergence include compiler-generated priority [Lorie and Strong, 1984], hardware stacks [Woop et al., 2005], task serialization [Moy and Lindholm, 2005], regrouping [Cervini, 2005], and micro-kernels [Steffen and Zambreno, 2010].

Distributed Computing. The most widely used approach today for communication between cluster nodes is MPI (Message Passing Interface) [Gropp et al., 1994]. As nodes in a cluster do not share memory, all data sharing and interaction is done through explicit message passing. Multiple MPI compute threads can reside on the same physical device and/or across an arbitrary number of devices. As a modern compute node typically features multi-/manycore processors, MPI is frequently used in a hybrid fashion in combination with OpenMP or CUDA for instance. MPI is a specification which has been implemented in systems such as MPICH and Open MPI [Prabhu, 2010]. A popular predecessor that has largely by supplemented by MPI was PVM (Parallel Virtual Machine) [Beguelin et al., 1991]. Besides these very generic techniques, there are frameworks focusing on a certain type of computing, like BOINC [Anderson, 2004] on large-scale grid batch computing. Additionally, many application-specific environments for user transparent computation on clusters have been proposed, especially in the area of image and multimedia processing [Baker et al., 1993; Li et al., 2002; Park et al., 2009; Seinstra et al., 2007].

One of the simplest and most common approaches of scheduling algorithms is to assign open task items to the first idle compute device. However, in many problem scenarios the differences of the heterogeneous processors in performance and bandwidth need to be taken into account to achieve full efficiency. Such resource-aware distributed scheduling strategies for large-scale grid/cluster systems were proposed by Viswanathan et al. [2007]. Teresco et al. [2005] worked on a distributed system in which every CPU requests task items from the scheduler which are sized according to the device's measured performance score. Wang et al. [2008] proposed a simple task scheduling algorithm for single machine CPU-GPU environments that not just uses the first idle compute device but chooses the fastest device from all idling devices. There has also been a lot of research on application-specific load balancing strategies. Zhou et al. [2009] propose a multi-GPU scheduling technique based on work stealing to support scalable rendering. In order to allow a seamless integration of load balancing techniques into an application, object-oriented load balancing libraries and frameworks were developed [Devine et al., 2000; Stankovic and Zhang, 2002].

The data dependencies of distributed parallel applications can be described by a graph structure (e.g., [Diekmann, 1998]). Here, it is assumed that the vertices of this graph represent data elements and that the edges denote the data dependencies. A distribution of the vertices across the available compute devices is achieved by partitioning the graph. Based on this graph, schedulers have been proposed that employ the critical path method, which was originally developed for scheduling project activities [Kelley and Walker, 1959]. An early application of the critical path scheduling to computation considering resource and processor constraints was presented by Lloyd [1982]. Kwok and Ahmad [1996] discuss the mapping of a task graph to multiprocessors.

2.2 Visualization Fundamentals

The goal of visualization is to create images that convey salient information about the underlying data and processes. This data can come from medicine, earth and space sciences, fluid flow and biology among many others areas [Johnson and Hansen, 2004]. Visualization has been used throughout history to provide a visual representation that explains complex phenomena, like the famous illustration of spatio-temporal movement of Napoleon's troops (cf. [Friendly, 2002]). However, the modern, compute-driven field of visualization was created with the advent of scientific computing and the use of computer graphics for depicting computational data. Sensor devices from medical scanners to satellites have also driven the field. The visualization of such simulated or measured spatial data is commonly referred to as scientific visualization. In contrast, information visualization works with data that typically lacks a (predominant) spatial domain, like data bases, social networks or text collections [Johnson and Hansen, 2004]. In this work, we focus on scientific visualization, and in particular the area of volume rendering.

2.2.1 Visualization Fundamentals and Characteristics

The concept of a visualization pipeline to describe the process of visualizing data was first proposed by Haber and McNabb [1990] (Fig. 2.9). *Raw data*, the input to the visualization pipeline, is obtained during *data acquisition* from different sources, like numerical simulations, databases, or sensors. Subsequently, *filtering* transforms raw

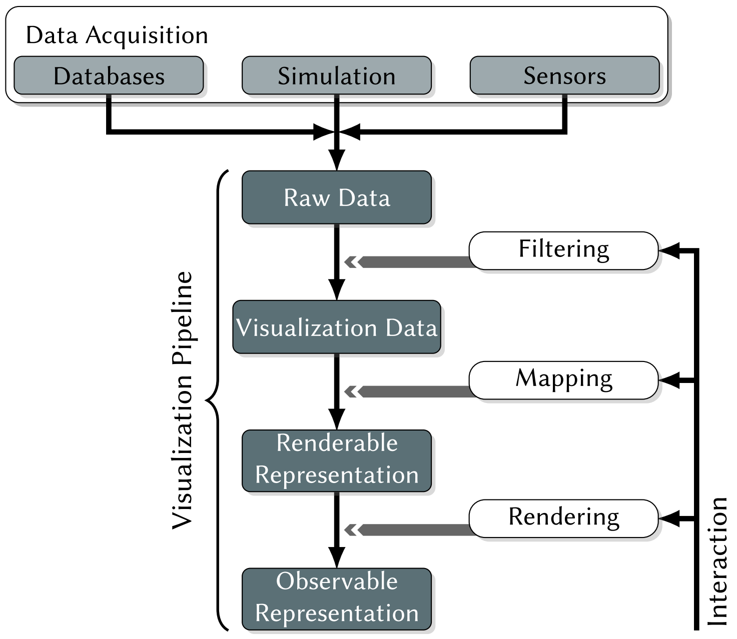


Figure 2.9: The visualization pipeline (based on [Weiskopf, 2006, chapter 1]).

data into *visualization data* with operations like smoothing, denoising, resampling, segmentation, classification, and selection. Next, *mapping* creates graphical primitives from visualization data, the so-called *renderable representation.* Attributes of a renderable representation may include geometry, time, color, surface texture, and opacity. Finally, *rendering* generates an *observable representation* that is intended to be used for analysis by a human, like (sequences of) images.

Traditionally, interactive post-processing is used for the visualization process. This means that data is generated or collected beforehand and stored on permanent storage. In this setup, visualization is completely decoupled from *data acquisition* and the data is loaded on demand into the visualization pipeline. Another approach is *computational steering* [Mulder et al., 1999], also known as *interactive steering*, with a tight connection between *data acquisition* and the visualization process. This implies that parameters for both, visualization and simulation, can be tweaked interactively, and is particularly popular with numerical simulations.

Throughout all stages of the pipeline, there are many degrees of freedom that have a critical influence on the outcome. Due to this large parameter space and the fact that oftentimes there is no detailed knowledge of the data beforehand, it is almost impossible to optimally adjust the parameters of the pipeline a priori. User interaction is therefore crucial to explore the large parameter space of simulation and visualization. Productive interactivity requires fast response times—in the order of tens to hundreds of milliseconds—to requested changes, e.g., a fast image update for changed camera configurations. As a consequence, the need for interactive visualization triggers the need for efficient algorithms and techniques.

Overall, while individual use cases might differ, scientific visualization applications typically share a number of characteristic properties.

Interactivity At least the *rendering* step of the visualization pipeline is typically required to be interactive, i.e., the computation of results is required to be fast with low latency.

Coherence In many cases, visualization routines on spatially or temporally adjacent data execute mostly similar computational steps and deliver largely similar results.

Large Data In scientific visualization, the data often consists of a large number of elements (currently typically in the order of millions to billions of elements). This poses challenges not only with respect to data handling in general, but also concerning the creation of an adequate *mapping* that is able to deliver expressive representations.

Data Parallelism Data parallelism is typical for most visualization problems [Moreland, 2013], e.g., the stages *Filtering*, *Mapping* and *Rendering* in the visualization pipeline potentially execute the same tasks for each incoming data element. In contrast, task parallelism is typically not enough to achieve the high degree of parallelism required for massively parallel systems. For instance, networks of process objects in the Visualization Toolkit [Avila, 2010] can reach a certain size with tasks that may be executed independently [Schroeder et al., 1998], but this is commonly by far not enough to occupy the thousands of cores of a GPU.

GPUs Today's visualization techniques commonly make use of one or multiple GPUs. With their SIMT architecture (Sec. 2.1.1), they are well-suited for the data parallelism inherent in typical visualization applications, and they are also able to deliver the high memory bandwidth that is required for data-intense computations.

2.2.2 Volume Rendering

Volume rendering is a core technique in scientific visualization that describes a set of techniques used to display a 2D projection from a 3D volume. It is not only applied to medical data from MRT or CT, but also to simulation and measurement data from numerous areas in engineering and science. A volume can be seen as a map that assigns a scalar value to positions in 3D Euclidean space. This data is typically given by means of discretely sampled data points, which necessitates the reconstruction of a continuous representation during rendering. For uniform data, this is most commonly accomplished by means of trilinear interpolation, that is directly supported by modern GPUs.

Optical Models

A volume can be visualized directly by evaluating an optical model which describes how it emits, reflects, scatters, absorbs and occludes light. The most important optical models for direct volume rendering are described as follows (according to Max [1995]):

Absorption only The volume consist of cold, perfectly black particles that absorb, but do not emit or scatter any light.

Emission only The volume consists of particles that emit, but do not absorb or scatter light.

Absorption plus emission Particles emit light, and occlude, i.e., absorb, incoming light. This is by far the most commonly used model in scientific visualization.

Scattering and shading/shadowing On top of absorption and/or emission, this model additionally includes the scattering of light. Scattered light can either be assumed to come undisturbed from a source, or it can be shadowed by particles in between the light and spatial position under consideration.

Multiple scattering This model extends single scattering to incident light that has already been scattered by multiple particles before it is directed toward the eye.

Note that this model is based on geometrical optics. It neglects effects such as diffraction, interference, polarization, etc.

The Volume Rendering Integral

In direct volume visualization, the light propagation is computed by integrating light interaction effects along viewing rays based on the emission-absorption model. For this, the change of radiance L at distance t from the eye can be formulated as

$$\mathrm{d}L(t) = g(t) - \kappa(t)L(t)\,\mathrm{d}t. \tag{2.12}$$

The amount of emitted light at t is given by the source term $g(t)$. The extinction coefficient $\kappa(t)$ defines how much of the incoming light is attenuated by the participating medium.

Integrating Equation 2.12 for a viewing ray starting at $t = 0$ and ending at $t = D$ yields the volume rendering integral [Max, 1995]:

$$L(D) = \underbrace{L(0)\mathrm{e}^{-\int_0^D \kappa(t')\,\mathrm{d}t'}}_{\text{absorption term}} + \underbrace{\int_0^D g(t)\mathrm{e}^{-\int_t^D \kappa(t')\,\mathrm{d}t'}\,\mathrm{d}t}_{\text{emission term}}. \tag{2.13}$$

With $L(0)$ giving the radiance at the start of the ray ($t = 0$), the accumulated radiance leaving the volume is provided by $L(D)$. Using the definition of optical transparency,

$$T(t_1, t_2) = \mathrm{e}^{-\int_{t_1}^{t_2} \kappa(t')\,\mathrm{d}t'}, \tag{2.14}$$

the volume render integral (Eq. 2.13) can be rewritten as

$$L(D) = L(0)T(0, D) + \int_0^D g(t)T(t, D)\,\mathrm{d}t. \tag{2.15}$$

During rendering, Eq. 2.15 is typically solved numerically, i.e., it is approximated by a Riemann sum. In the simplest and most popular case, n equidistant segments of length δ are used. Variable (adaptive) step sizes $\delta(i)$ are also commonly employed. With this, the approximation of optical transparency (Eq. 2.14) between segment i at $t_i = \sum_{j=0}^{i} \delta(j)$ and the end point at $t = D$ yields

$$T(t_i, D) \approx \mathrm{e}^{-\sum_{k=i}^{n-1} \kappa(t_k)\delta(i)} = \prod_{k=i}^{n-1} \mathrm{e}^{-\kappa(t_k)\delta(i)}. \tag{2.16}$$

With the transparency of the ith segment given by $T(i) = \mathrm{e}^{-\kappa(t_i)\delta(i)}$, the volume rendering integral from Eq. 2.15 can be approximated as follows with n segments ($g(\cdot)$ provides non-premultiplied color):

$$L(D) \approx L(0)\prod_{i=0}^{n-1} T(i) + \sum_{i=0}^{n-1}\left(T(i)g(i)\prod_{j=i+1}^{n-1} T(j)\right). \tag{2.17}$$

Raycasting is widely considered to be the most direct numerical method for evaluating the (discretized) volume rendering integral. In essence, raycasting shoots a single ray from the eye through the pixel's center into the volume, and integrates the optical properties obtained from the encountered volume densities along the ray. Transparency is typically provided by means of the transfer function lookup $\alpha(i)$, which is independent of segment length δ(t). Thus, $T(i)$ is computed as follows:

$$T(i) = \omega(\alpha(i), \delta(i)) = 1 - \big(1 - \alpha(i)\big)^{\delta(i)}, \tag{2.18}$$

Classification	Direct Volume Rendering	Indirect Volume Rendering
Image-space (Backward Mapping)	Raycasting [Levoy, 1990a]	Raycasting Isosurfaces [Parker et al., 1998]
Object-space (Forward Mapping)	Texture Mapping [Krüger and Westermann, 2003], Splatting [Westover, 1991]	Marching Cubes [Lorensen and Cline, 1987]

Table 2.4: Classical techniques of different classes of volume rendering approaches.

with $\omega(\alpha(i), \delta(i))$ being the opacity correction operator. Similar to transparency, the source term $g(i)$ is also determined by means of a transfer function lookup $c(i)$ that typically represents an RGB color. Eq. 2.17 can be implemented with a simple compute kernel that is executed for each ray (Alg. 3).

Algorithm 3 Simple front-to-back raycasting kernel implementation. C denotes the final RGB color for a pixel (akin to radiance L).

```
1: function VOLUMETRIC RAYCASTING KERNEL
2:     $C \leftarrow (0,0,0)$                         ▷ One channel for red, green and blue
3:     $T \leftarrow 1$                               ▷ Initially no radiance is absorbed.
4:     for $i = 0 \to N-1$ do                         ▷ step along ray (forming a new segment)
5:         $T(i) \leftarrow 1 - (1-\alpha(i))^{\delta(i)}$   ▷ compute opacity of segment
6:         $C \leftarrow C + T(i)c(i)$                ▷ composit color
7:         $T \leftarrow T \cdot (1 - T(i))$          ▷ accumulate opacities
```

Classification of Volume Rendering Approaches

Volume rendering techniques are popularly classified into direct and indirect as well as in image and object space techniques (Tab. 2.4). Indirect volume rendering methods are typically implemented by making use of isosurfaces, i.e., surfaces that represent points of constant value. Isosurfaces can either be rendered using raycasting or explicitly be extracted in the form of a mesh by techniques like Marching Cubes (MC). In contrast, direct volume rendering utilizes the discretized volume rendering integral (Eq. 2.17) in different variations. While object-space techniques compute the contribution of each element in the volume individually, image-order techniques sample the 2D image plane and determine the radiance leaving the volume for each pixel. Different techniques for all four combinations are briefly outlined in the following.

Direct Object-space Techniques The contribution of each grid cell is computed and then projected onto the image plane. Texture-based slicing by Westermann and Ertl [1998] blends object-space-aligned stacks of layers on the graphics card. As an extension using capabilities of more recent graphics hardware, texture-based rendering via 3D textures slices the texture block in back-to-front order with planes oriented parallel to the view plane [Krüger and Westermann, 2003]. Shear-warp factorization transforms the volume to sheared object space by translation and resampling, and projects this representation to an intermediate image that is warped to produce the final image [Lacroute and Levoy, 1994]. Projected tetrahedra renders partially transparent polygons based on the projected profile of tetrahedral cells [Shirley and Tuchman, 1990; Röttger et al., 2000]. Splatting accumulates data points by projecting flat disc-like kernels for each voxel to the image plane [Westover, 1991]. Many adjustments have been proposed to improve both the quality and speed of splatting (e.g., [Mueller et al., 1999; Vega-Higuera et al., 2005]). Cell-projection techniques project volume elements to the image plane one after the other in their order of their visibility. Unfortunately, due to this, non-convex cells cannot be handled, and a correct sorting even of convex cells is not always possible due to visibility cycles [Kraus and Ertl, 2001].

Indirect Object-space Techniques: Isosurface Extraction One of the classical visualization techniques is Marching Cubes (MC) due to Lorensen and Cline [1987]. It generates a triangle mesh from volume data given on uniform grids. Numerous variants have been proposed for improving different properties of the original algorithm [Kobbelt et al., 2001; Theisel, 2002; Dietrich et al., 2009; Bommes et al., 2012]. For instance, dual MC generates quad patches that tend to eliminate the common issue of poorly shaped triangles of the MC isosurfaces [Nielson, 2004]. MC adaptations were also proposed for tetrahedral meshes [Zhou et al., 1997; Anderson et al., 2005], and supporting adaptive reconstruction [Grosso and Ertl, 1998; Westermann et al., 1999]. For isosurface extraction from higher-order data, quad mesh generation techniques [Remacle et al., 2012], contouring [Wiley et al., 2003], and approximate isocontouring [Pagot et al., 2011] have been proposed. Other approaches use Voronoi diagrams [Dey and Levine, 2007], advancing front techniques [Schreiner et al., 2006], and meshing from point clouds [Scheidegger et al., 2005]. Some isosurface extraction techniques provide disjoint meshes as a result. Several approaches have been proposed for combining these, including sewing [Kobbelt and Botsch, 2000], volumetric methods [Curless and Levoy, 1996], zipping overlapping meshes [Turk and Levoy, 1994], laser range images from different views [Rocchini et al., 2004], and polygon triangulation [Held, 2000].

Indirect Image-space Techniques: Isosurface Rendering For isosurface visualization, point-based [Rosenthal and Linsen, 2006; Meyer et al., 2007] and raycasting-based [Stegmaier et al., 2005] visualization approaches have been presented. Point-based techniques generate a point cloud representation of the isosurface which is then visualized using point-based rendering techniques, most prominently splatting [Pfister et al., 2000]. Raycasting-based techniques trace a ray until the isosurface is hit, typically employing intersection refinement to determine a more exact position on the isosurface. Different techniques have been proposed for various areas of application. Sadlo et al. [2011] presented a technique for rendering cell-based higher-order fields. Nelson et al. [2011] proposed a raytracer for cut-surfaces. Knoll et al. [2009b] discuss the raycasting of classical polynomials, while Gamito and Maddock [2007] render radial basis functions from smoothed-particle hydrodynamics.

Direct Image-space Techniques: GPU Volumetric Raycasting Nowadays, GPU-based raycasting is the state-of-the art technique for interactive volume rendering [Rezk-Salama et al., 2009]. In typical scenarios, the computation of each ray is completely independent, and rays are largely coherent, which makes raycasting well-suited for the SIMT architecture of GPUs. An early implementation of GPU raycasting required multiple passes due to hardware restrictions [Krüger and Westermann, 2003]. With more flexible GPU programming capabilities, this has been superseded by single-pass volume raycasting [Hadwiger et al., 2005; Stegmaier et al., 2005]. To initiate the rendering, the bounding box of the volume is rendered as proxy geometry. The object space coordinates of the bounding box vertices are interpolated for each render primitive and are then used in each covered fragment to set up a ray. The individual rays are sampled equidistantly in the fragment shader until they leave the bounding box. The volumetric data is stored in a 3D texture and accessed using hardware-accelerated trilinear interpolation. Subsequently, the transfer function is applied, which is typically implemented by means of another texture lookup. Illumination is often employed to improve spatial perception, typically using the gradient of the volume at the respective sampling position for this purpose [Hadwiger et al., 2008; Rezk-Salama et al., 2009]. The gradient is either obtained on-the-fly, e.g., by applying central differences, or by a lookup to a precomputed texture.

Advanced Volume Rendering Techniques

Raycasting for direct volume rendering is one of the fundamental techniques for this work, and scientific visualization in general. In the following, optimized variants of this technique are outlined, focusing on different aspects.

Object-space Acceleration Many diverse approaches exist for accelerating volume rendering. A classical method is early ray termination, which stops the integration of a ray when the opacity exceeds a certain threshold [Levoy, 1990c]. Another classical technique is empty space skipping. For this, proximity clouds employ a distance transform of the object to accelerate the rays in regions far from object boundaries [Cohen and Sheffer, 1994; Zuiderveld et al., 1992]. Distance transform values are stored in place of density values in these empty regions, and therefore storage is not increased. Freund and Sloan [1997] additionally utilize the fact that medical imaging scanners only produce twelve bit intensity values for each voxel, and use the remaining four bits to store adaptive step sizes in occupied regions. Instead of using a dedicated data structure, Klein et al. [2005] utilize the spatial coherence of consecutively rendered images to determine the initial sampling point of a ray on the GPU. Additionally, numerous approaches employing hierarchical data structures like octrees were presented (e.g., Levoy [1990a]). Non-leaf nodes usually store an entropy metric of its children to be able to traverse occupied space faster when the entropy is low. Employed metrics popularly include standard deviation [Danskin and Hanrahan, 1992] and the minimum-maximum range of values [Wilhelms and Van Gelder, 1992]. Guthe and Strasser [2001] store a measure of the error that is committed when its children are not rendered in each non-leaf node of the octree. Crassin et al. [2009] use a dynamic sparse octree approach to exploit occlusion information. Furthermore, adaptive techniques have been proposed that take entropy into account. Object-space importance sampling techniques typically rely on precomputed data structures (e.g., LOD volumes [Ljung et al., 2006]). Viola et al. [2004] present an importance-driven automatic focus and context display technique.

Image-space Acceleration Levoy [1990b] proposed to not necessarily cast one ray per pixel for volumetric raycasting, but to use one retrieved value for multiple pixels. Kratz et al. [2011] use an error estimator from the field of finite element methods for adaptive screen-space sampling. For unstructured volumes, Callahan and Silva [2009] propose to employ the combination of a low resolution image of the whole dataset and a high resolution image of the boundary geometry. Qu et al. [2000] and Shen and Johnson [1994] (among others) exploit frame coherency by reutilizing pixel values from the previous frame by warping the positions to the current frame. Bolin and Meyer [1995] propose an adaptive raytracing approach that utilizes perception-based metrics in frequency space. While for interactive volumetric raycasting typically undersampling schemes are employed that send maximally one ray per pixel, Monte Carlo rendering techniques for photorealistic rendering employ oversampling to reduce noise. Overbeck et al. [2009] adaptively distribute Monte Carlo samples to reduce the variance in wavelet space. Farrugia and Péroche [2004] present a perceptually based approach for progressive rendering for global illumination. Bolin and Meyer [1998] employ perceptually based adaptive sampling based on an extended image procession vision model. Ramasubramanian et al. [1999] utilize a physical error metric with global illumination algorithms.

View-Dependent Representations Image-based rendering infers new images from existing ones, e.g., with changed lighting or camera configuration [Shum and Kang, 1999]. A number of techniques has been proposed to construct such different representations from multiple views, like view-dependent texture maps [Debevec et al., 1996], warping [Mark et al., 1997; McMillan and Bishop, 1995], light fields [Levoy and Hanrahan, 1996] or Lumigraphs [Gortler et al., 1996]. Rezk-Salama et al. [2008] employ depth layers to generate light field representations from volumetric data. Other techniques use multiple images to synthesize new surface-based views of volume data (e.g., [Choi and Shin, 1998; Chen et al., 2001]). Such techniques allow the adaptation of color and lighting parameters [He et al., 1996], or transfer functions [Wu and Qu, 2007].

Shade et al. [1998] introduced Layered Depth Images (LDIs) that represent one camera view with multiple pixels along each line of sight. Since then, multi-layered representations have been popularized in commercial rendering software to simulate complex materials like skin on synthetic objects [Donner and Jensen, 2005]. In volume rendering, layer-based representations have been used to defer operations such as lighting and volume classification [Ropinski et al., 2008; Rautek et al., 2007]. Such representations have also proven effective to cache results [Luke and Hansen, 2002; LaMar and Pascucci, 2003] or certain volumetric properties along view rays [Ma et al., 1991] that can be later reused for efficient transfer function exploration. Also for deferred transfer function exploration, Tikhonova et al. [2010a] convert a small number of volume renderings to a multi-layered image representation. In another work, Tikhonova et al. [2010b] use an intermediate volume data representation which encodes the distribution of samples along each ray. Shareef et al. [2006] use image-based modeling to render unstructured grids based on parallel sampling rays and 2D texture slicing.

Distributed Volume Rendering Most of the existing systems for distributed volume rendering fit into the sort-first or sort-last category according to the classification due to Molnar et al. [1994]. In the sort-first category, different sections of the screen are rendered in parallel by different nodes. In the sort-last category, data is split between the nodes, and each node renders its own portion. For sort-last approaches, a compositing step combining each node's rendering is additionally required to generate a final image. Wylie et al. [2001] demonstrated that sort-last volume rendering techniques are able to handle very large datasets by statically distributing these datasets among the nodes. The predominant hierarchical compositing schemes that are used in sort-last architectures aiming at rendering large data sets are the Direct Send approach by Neumann [1993] and the Binary-Swap algorithm due to Ma et al. [1993]. Palmer et al. [1997] discussed how to efficiently exploit all levels of the deep memory hierarchy of a cluster system. Using the clusters that compute the simulation also for volume rendering has been investigated by Peterka et al. [2008b].

There has been a lot of work in the past years on data structures that can be used to address dynamic load balancing issues in distributed volume rendering systems. Wang et al. [2004] proposed a hierarchical space-filling curve for that purpose. Lee et al. [2005] employ a combination of octree and BSP tree to both distribute workload and skip empty regions. Müller et al. [2006] and Marchesin et al. [2006], among others, employ a k-d tree to dynamically reorganize the data distribution in a cluster. They demonstrated that zooming on parts of the data sets critically impairs the balance of the load. To achieve good load balancing, they dynamically adapt the k-d tree according to the load of the previous frame. An approach to statically avoid significant load imbalance has been proposed by Peterka et al. [2008a]. They generate more volume bricks than there are devices and assign every brick to one device during initialization using a round-robin scheme. Frank and Kaufman [2009] use data dependency information to automate and improve load balanced volume distribution and ray-task scheduling. A directed acyclic graph of bricks is employed and a cost function is evaluated to create a load balanced network distribution.

Remote Rendering Remote rendering has been used for a multitude of different applications, including complex visualization on mobile devices [Diepstraten et al., 2004], cloud gaming (OnLive or Gaikai), protection of proprietary data [Koller et al., 2004], in-situ visualization of time-varying volume data [Ma and Camp, 2000], and visualization in medical applications [Engel et al., 2000]. The data transfer size is typically reduced by employing lossy compression, e.g., by means of JPEG [Koller et al., 2004] or MPEG [Herzog et al., 2008]. In more detail, Herzog et al. [2008] propose an approach to couple global illumination rendering and MPEG compression for non-interactive 3D animation rendering. To reduce latency in cloud gaming, Lee et al. [2014] propose to produce speculative rendered frames of future possible outcomes on the server and deliver them to the client in advance. A CUDA-based parallel entropy encoding method targeted at remote rendering was presented by Lietsch and Marquardt [2007]. Pajak et al. [2011] discuss efficient compression and streaming of frames rendered from a dynamic 3D model. LOD techniques are frequently employed in this context to handle excessive server load [Moreland et al., 2008; Koller et al., 2004].

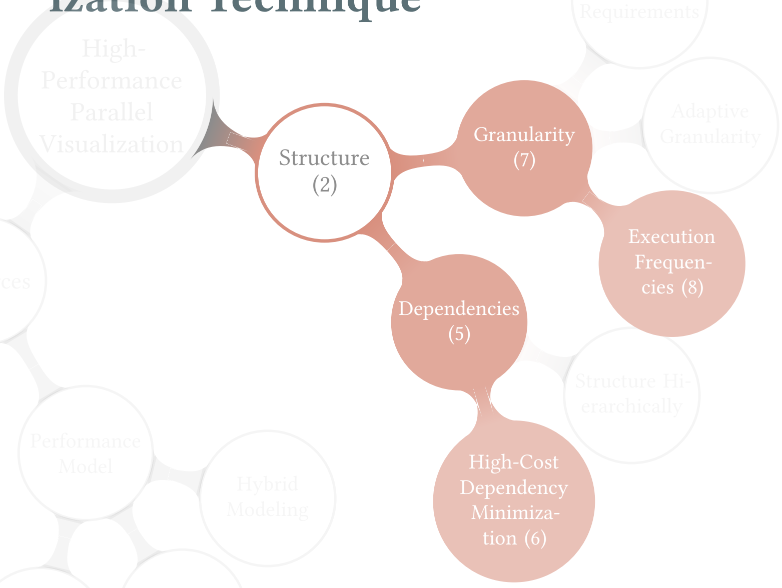
Parallel Visualization Frameworks Scientific visualization has a long history of using high performance parallel computing to handle large-scale data [Moreland, 2013; Ament et al., 2012], and several architectures and frameworks have been introduced that do not limit themselves to a single application but can be applied to a more generic class of visualization problems. Haimes and Edwards [1997] discuss an early interactive visualization system based on MPI combining visualization and parallel processing environments for the analysis of engineering applications. Ellsworth et al. [2006] describe a concurrent visualization pipeline for a supercomputing environment. Even more generically, ParaView [Ahrens et al., 2005] provides a graphical user interface for the creation and dynamic execution of visualization tasks.

Chapter 3

Analysis of Time-Dependent Data

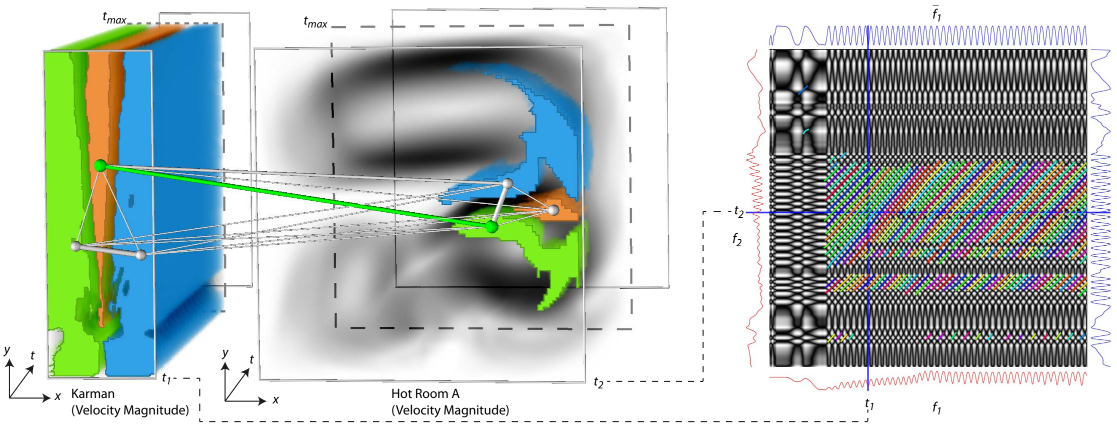
The analysis of time-dependent data is challenging, not only in terms of finding meaningful representations, but also in the context of efficient data processing. This often requires solving a number of different problems, eventually leading to comparatively complex applications. However, this also means that there are numerous degrees of freedom in the design phase of the algorithm. Utilizing this flexibility, the goal here is to structure the overall approach such that it can be executed efficiently on parallel hardware, optimally with the ability to execute the most compute-intense parts independently from each other in parallel. Among others, one key question arising in this context is what interaction possibilities can or should be provided to the user, and what parts needs to be precomputed.

3.1 Parallel Design of a Temporal Similarity Visualization Technique


High-Performance Parallel Visualization
Structure (2)
Granularity (7)
Execution Frequencies (8)
Dependencies (5)
High-Cost Dependency Minimization (6)
Functionality Requirements
Adaptive Granularity
Structure Hierarchically

Two-dimensional grids of signals, so-called time-dependent field data, are a common representation in large parts of science and engineering. Detecting and exploring spatio-temporal similarity or patterns in this data may help to detect new correlations. On the level of individual signals, similarity matrices have been a popular tool for the examination of temporal similarity for many years. This representation is capable to depict the self-similarity of one signal or the cross-similarity of two signals. The similarity between the different states in time for signals f_1 and f_2 is shown as a dense gray level 2D plot in which both abscissa and ordinate represent the same time interval (Fig. 3.1*(b)*). Dark values represent high similarity, and accordingly, similar processes represent similar sequences of states and appear as dark lines in these plots. While similarity matrices primarily provide a visual representation, they also allow the extraction of certain structures. In this project, similar sequences of states are extracted by means of so called similarity lines (colored lines in Fig. 3.1*(b)*). These are then put into spatial context and finally visualized using volume rendering supported by further annotations (Fig. 3.1*(a)*).

From an algorithm design point of view, the requirement for global similarity analysis inherently introduces dependencies that cannot be avoided completely. However, an adverse dependency structure is one of the main factors hindering parallel scaling by forcing frequent waiting, synchronization, and data transfers among others. The extraction of similarity information using similarity matrices is by far the most compute-intense part, and therefore needs to be carefully designed for independent parallel computation in this project. In contrast, the subsequent usage of this information to



(a) Space-time view of the time-dependent von Kármán (left) and Hot Room A (right) data sets.

(b) Similarity matrix (gray) and lines (colored) for the selected cluster connection in *(a)*.

Figure 3.1: *(a)* Clusters of space-time similarity are depicted in blue, red, and green. Similarity between cluster masters (spheres) is depicted by thickness of their links (tubes). These can be interactively selected (green). *(b)* Their normalized signals $\bar{f}_1$ and $\bar{f}_2$ are then used to determine and visualize their similarity by means of a similarity matrix.

generate clusters of similarity across signals exhibits complex interdependencies, but is cheap to compute in comparison.

Strategy 5 ***Dependencies***
Minimize the dependencies between task items to reduce both synchronization overhead and load imbalance.

Strategy 6 ***High-Cost Dependency Minimization***
Keep the dependencies within and between costly tasks as low as possible, and instead swap more complex dependency relations to cheaper tasks whenever possible.

Another goal is to generate enough work to fully harness highly parallel computation devices, but to keep the ratio of induced synchronization overhead low at the same time. For this, task items are chosen to reach the granularity of a single spatial data element (i.e., signal) level in the early, costly stages of the procedure. Here, a finer grained subdivision with single spatio-temporal points would induce redundant computation or dependencies. Additionally, basic similarity extraction and visualization are decoupled to achieve a separation of interactive and non-interactive parts of the algorithm.

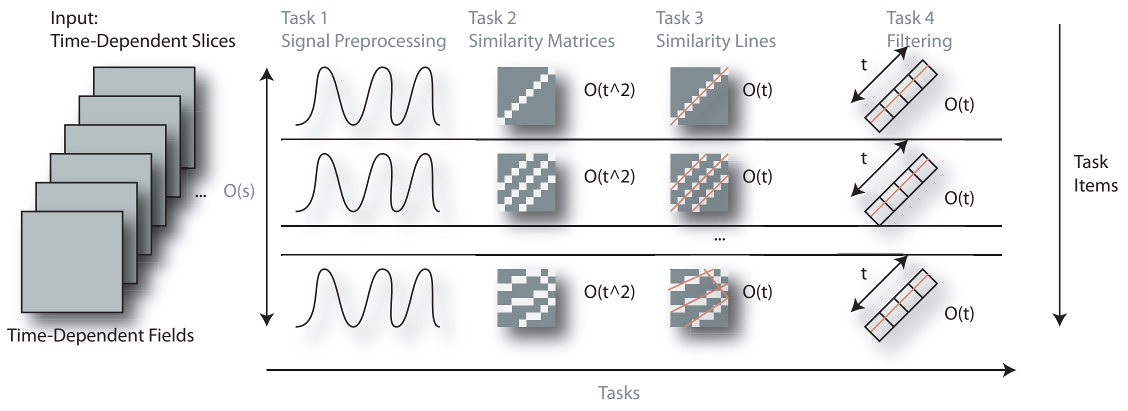


Figure 3.2: Partitioning of the similarity extraction into tasks one to four (based on this, tasks five to eight generate a similarity visualization (Fig. 3.3)). $O(\cdot)$ depicts the number of task items. Variables s and t stand for the number of spatial elements and time steps, respectively.

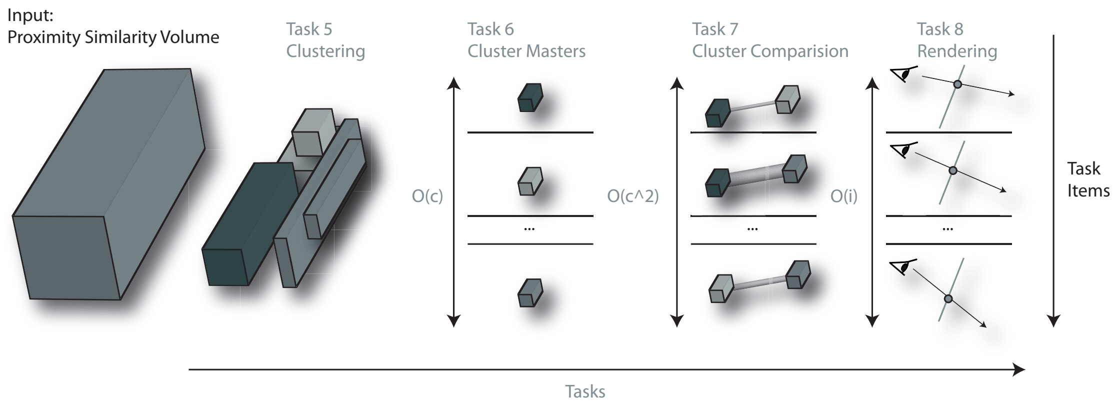
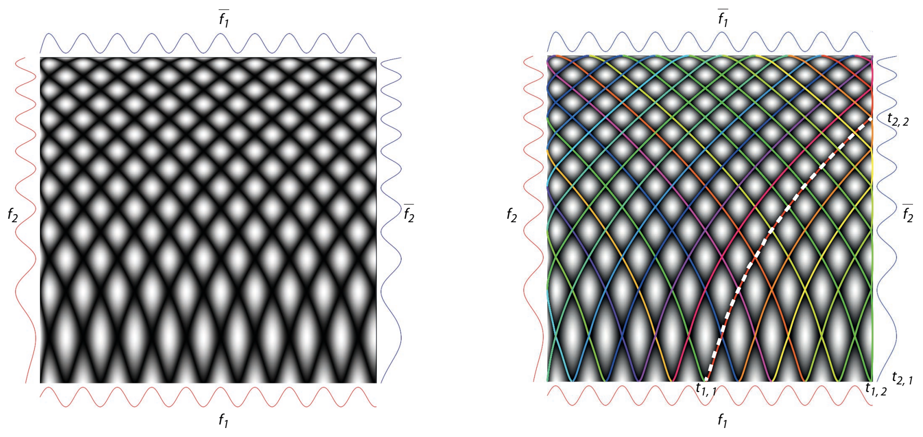


Figure 3.3: Partitioning of the spatio-temporal similarity visualization into tasks five to eight. This is based on the extracted similarity information from tasks one to four (Fig. 3.2). c represents the number of clusters and i denotes the number of pixels in the rendering.

Strategy 7 ***Granularity***
Optimize the granularity to achieve the most efficient ratio of computation to communication. The finer the granularity, the bigger the potential for parallelism and hence speed-up, but also the larger the overhead for synchronization and communication.

Strategy 8 ***Execution Frequencies***
Partition the program structure into different frequencies of execution, e.g., seperate preprocessing from interaction-capable parts.



(a) Similarity matrix (b) Extracted similarity lines

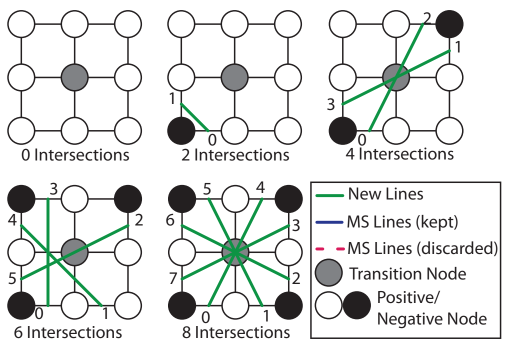
Figure 3.4: Similarity matrix (black ≃ similar, white ≃ different) and (multicolored) lines. Red graphs (left and bottom of the matrix) show the smoothed signals f_1 and f_2 of the test functions $\sin(x)$ and $\sin(0.01 \cdot x^2 + 0.1 \cdot x)$, respectively. Blue graphs (right and top) depict the normalized versions $\bar{f}_1$ and $\bar{f}_2$ used for similarity matrix computation.

3.1.1 Parallel Structure

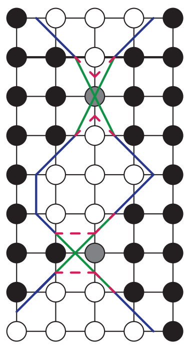
Our approach to reveal temporal similarity in field data is broken down into separate tasks. The expensive computations necessary to extract the similarity information are partitioned into the parallelization-friendly tasks one to four (Strat. 6, *High-Cost Dependency Minimization*). This cost-aware splitting also divides the system into a batch part and a part that allows for interactive exploration (Strat. 8, *Execution Frequencies*) (Figs. 3.2 and 3.3). Here, our discussion mainly focuses on computational aspects. Please refer to the respective paper for more details and a closer evaluation of the visualization properties [Frey et al., 2012b].

Task 1 **Signal Preprocessing** Signals are preprocessed individually to make them directly comparable to each other. For this, we first reduce noise using triangular smoothing. Second, the signals are normalized with respect to homological persistence [Edelsbrunner et al., 2000; Reininghaus et al., 2011] of its local extrema (blue and red signal in Fig. 3.1(b), respectively) to make them invariant toward offset and amplitude. These operations can be carried out for all signals in parallel without any inter-dependencies.

Task 2 **Similarity Matrices** Similarity matrices are generated by comparing all states of signal f_1 to all states of signal f_2. For every signal of a data set, we generate



(a) Cases around a transition node.



(b) Example

Figure 3.5: Adapted Marching Squares (MS) algorithm for similarity line extraction. Both transition nodes in this figure were generated from negative values.

one self-similiarity matrix as well as two cross-similarity matrices (each consisting of scalar entries), one with its bottom neighbor and one with its top neighbor. For parallelization, independent operations can be as small as the similarity computation of any two points of the signal(s) in space-time. However, here, one task item is mapped to the generation of a whole similarity matrix to avoid unnecessary overhead (Strat. 7, *Granularity*).

Task 3 Similarity Lines Similar sequences of states can be geometrically extracted from similarity matrices. These are denoted as *similarity lines* (Figs. 3.1(b), 3.4(b)). They directly represent important characteristics regarding the temporal variation of the underlying signals, e.g., the slope of similarity lines shows the temporal scale of similar processes (or the "ratio of their frequencies"). As similarity lines often intersect, contours as obtained by the traditional isoline extraction technique Marching Squares (MS) tend to form "islands" instead of the desired long intersecting lines (Fig. 3.5(b)). Our modified version introduces *transition nodes* (Fig. 3.5(a), gray) and provides an additional set of MS cases to handle them. A node becomes a transition node when both its two horizontal (or vertical) neighbors have an opposite sign (e.g., both are negative and the considered node is positive), and when no other transition nodes are adjacent. The introduction of these transition nodes introduces dependencies between pixels. However, this does not matter for parallelization as, like in Task 2, the granularity of task items is on the level of whole similarity matrices.

Task 4 Filtering Similarity lines are filtered, as shown in Fig. 3.1(b), to select certain similarity characteristics of interest according to recurrence quantification analysis (RQA) [Zbilut and Webber, 1992; Marwan et al., 2007]. Each line can be processed in parallel without having to consider any dependencies. We found two criteria to be particularly useful in our experiments. First, the *length* of a similarity line relates to the duration of the respective similar process. Second, the *slope* reflects the respective

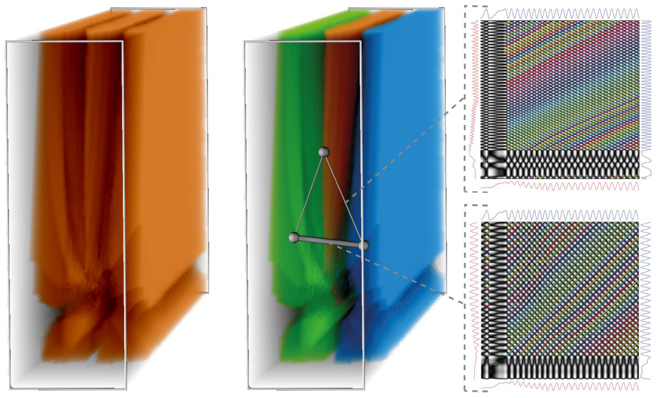


Figure 3.6: (a) Visualizing self-similarity directly and (b) via clusters for the von Kármán data set. Thick lines connect regions with similar frequency (defined by $s_{\min}$ and $s_{\max}$), while thin lines connect regions that are not as closely related (*top right:* the lines for the blue and the orange cluster are filtered out for similarity determination here and are only shown for demonstration purposes).

(a) Self-similarity (b) Similarity clusters

relative time scale of a similar process. For instance in Fig. 3.4(b), the dashed line has a high slope close to $(t_{1,1}, t_{2,1})$, because the frequency of f_1 is much higher there than that of f_2. Close to $(t_{1,2}, t_{2,2})$ the slope is approximately $\pi/4$, which means that the frequencies are similar.

Task 5 Clustering Using the similarity information generated so far, we aggregate space-time voxels to regions of similar time variation. As a heuristic, the number of lines present after filtering at a given time step is used as an indicator for similarity. Fig. 3.1(a) shows the space-time voxels exceeding this similarity threshold with respect to their self-similarity matrix. With the generated cross-similarity matrices, each space-time voxel also has similarity information for its neighbors. This allows a simple region-growing approach to cluster different areas of similarity (blue, orange and green volumes in Fig. 3.6(b)). Separating the structures using clustering helps the visualization and enables further analysis techniques. While the clustering step features many inherent dependencies, it is relatively cheap to compute, even serially, as it mostly relies on the previously computed similarity information.

Task 6 Cluster Masters The relation between the clusters can provide valuable insight. For efficient cluster comparison, a space-time voxel is determined to be the representative for its cluster (the so-called master, depicted as sphere in Fig. 3.6(b)). The cluster master is simply identified as the space-time voxel that is most strongly connected to its six neighbors, according to the same criteria that is also used for clustering in Task 5.

Task 7 Cluster Comparison The cross-similarity matrix between pairs of cluster masters is computed to provide a measure for the similarity of clusters. As a heuristic, we used the length of the longest similarity line in the respective matrix, i.e., the duration of the longest common process. The result is then visualized via the width of the cluster-connecting line (e.g., tubes in Fig. 3.6(b)). Cluster-connecting lines can be selected by the user (e.g., green line in Fig. 3.1(a)) for a detailed investigation of the similarity relation by means of the cross-similarity matrix (Fig. 3.1(b)). The filter criteria on which the cluster comparison is based can be adjusted interactively. In Fig. 3.6(b) for instance, the thick link between the left and the right cluster shows strong temporal similarity (with respect to frequency) as their processes are basically mirrored horizontally. In contrast to this, the two links to the middle cluster are thin because the frequency of the signals differs.

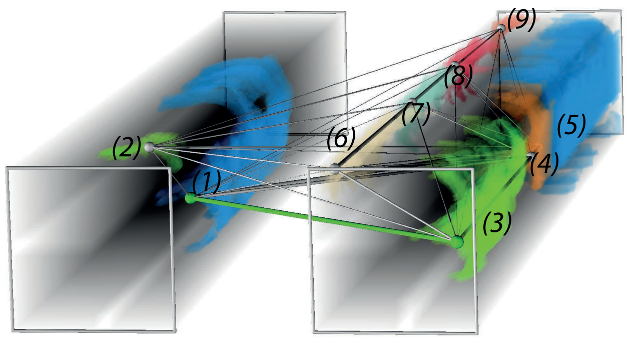
Task 8 Space-Time Volume Rendering The time-dependent fields in space-time representation are visualized using a CUDA-based volume raycaster, combined with OpenGL geometry to draw the links between clusters as well as the domain outlines (Fig. 3.1(a)). The data can be explored interactively, e.g., by changing the camera position, modifying the clustering parameters, or shifting the fields along the time axis and clipping them at the front plane at times t_1 and t_2 to reveal the respective state of the field (see also the dashed end of time indicators $t_{\max}$ in Fig. 3.1(a)).

3.1.2 Results

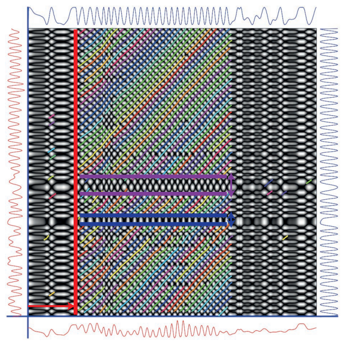
The approach is evaluated by means of measured and simulated data sets (Table 3.1) on a machine equipped with a Core i7 with 2.66 GHz using OpenMP, 8 GB of main memory, and an NVIDIA GTX580 with 3 GB of memory. The space-time visualization part runs at interactive rates in our implementation (Task 8). Tasks 5–7 exhibit computation times in the order of a couple of seconds on the CPU, which could be brought to interactive rates as well with a more tuned implementation (e.g., utilizing the GPU). However, extracting

Table 3.1: Data sets used in the evaluation. Timings are given for connectivity extraction (Tasks 1–4) on a cluster featuring eight nodes, each equipped with an eight core Intel Xeon CPU running at 2.4GHz. The ocean temperature data set is discussed in [Frey et al., 2012b].

Name	Width × Height	Timesteps	Timing
von Kármán Vel. Mag.	301 × 101	800	4 Mins
Hot Room A Temp.	101 × 101	1600	8 Mins
Hot Room A Vel. Mag.	101 × 101	1600	7 Mins
Hot Room B Temp.	101 × 101	1600	8 Mins
Ocean Temp.	360 × 180	1826	89 Mins



(a) Temperature in Hot Room A (*left*) and Hot Room B (*right*).



(b) Similarity matrix for green line in (a)

Figure 3.7: Cross-comparison between temperature development in Hot Room A and Hot Room B. In (a), the von Kármán Vortex Street sets in much later in A than in B. B further exhibits two interruptions of temporal similarity, resulting in an additional cluster (orange). (b) Both, the time frame of the cluster in A (red label) as well as the two interruptions in B (blue and purple) also show in the cross-similarity matrix of the respective masters.

the similarity information required for the detection of self-similarity and clustering is expensive (Tasks 1–4). Yet, these computations can be carried out independently from each other and distributed across a cluster (timings in Table 3.1).

The Hot Room data set results from a time-dependent 2D simulation of air flow within a closed container, driven by buoyant forces imposed by a heated bottom plate and a cooled top plate. To provoke transient aperiodic flow, the container exhibits two barriers. Two variants of this data set are used for evaluation here. They only differ in the size of the barrier at the bottom wall: variant “Hot Room A” has a square obstacle, whereas “Hot Room B” features a quadrangular obstacle of double height. Both data sets include velocity and temperature. The data sets consist of 1600 timesteps, but we skip the first 800 in our analysis to focus on the interesting effects between time steps 800 and 1600 (Fig. 3.7).

For the temperature fields, the difference of the shape of the obstacle at the bottom wall results in a significant change in the time frame in which recurrent processes occur (Fig. 3.7(a)). Most prominently, data set A features only two clusters of sufficient size, compared to seven in B. It can also be seen that the recurrent processes in B start earlier (front-most clusters *(1)* and *(3)* in (a) as well as their similarity lines in (b)). The cross-similarity lines show that the processes run at approximately the same frequency. The temporal disruption of clusters *(3)*, *(4)* and *(5)* as well as the temporal offsets of the clusters also appear in the cross-similarity lines (note that the masters of clusters *(3)*, *(4)* and *(5)* are virtually identical). The thickness of the cluster-connecting lines indicates that there is high similarity between clusters *(1)* from A and *(3)*, *(4)*, and *(5)* from B.

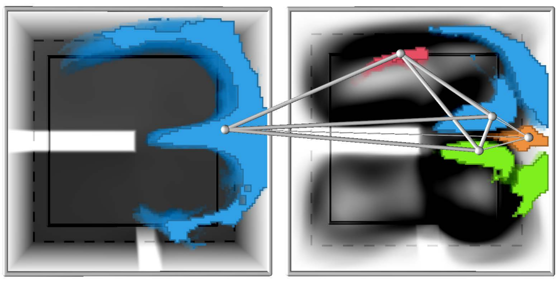


Figure 3.8: Cross-comparison between temperature *(left)* and velocity magnitude *(right)* of Hot Room A. The cluster connecting lines show that two major and one minor clusters of the velocity magnitude are similar to the major temperature cluster, while one velocity cluster (orange) substantially differs.

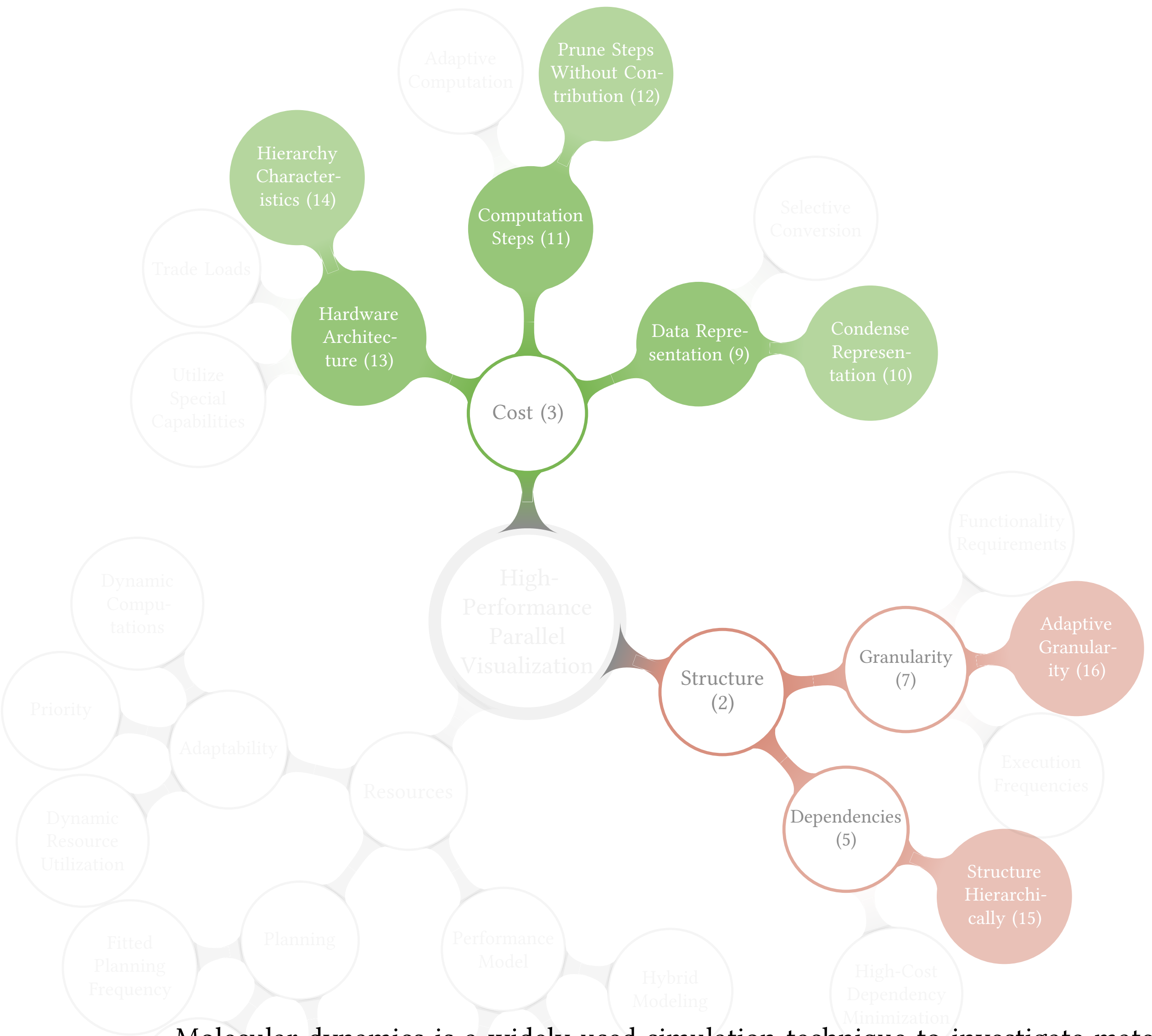
This means that despite all differences between the two data sets, the spatial points right of the horizontal obstacles in both data sets exhibit similar variation, however, at different points in time.

Comparing velocity magnitude against temperature, the velocity magnitude splits into three regions while there is only one large cluster for temperature (Fig. 3.8). The thickness of the connecting lines shows that the behavior of the large temperature cluster is similar to the bottom and top clusters in the velocity data, but not to the middle cluster. This visualizes that the middle cluster exhibits twice the frequency (as in Fig. 3.6).

3.1.3 Directions for Future Work

A simple way to accelerate the implementation significantly would be to move the similarity matrix-related tasks to the GPU. Further evaluation of the visual representation is also required, particularly including the comparison with other temporal feature detection approaches as well as a more detailed analysis with a larger range of data sets. Additionally, the approach can be extended to not only handle similarity lines but also similarity structures. The method could further be applied to data along trajectories in vector fields to also account for the Lagrangian frame. Finally, the presented approach breaks down the similarity detection to single signals or signal pairs, such that they can be handled computationally efficiently by standard 2D similarity matrices. However, similarities could be detected without any conversion directly in high dimensional space (6D for time-dependent 2D fields, 8D for time-dependent 3D volumes). This similarity extraction scheme would allow to identify more generic correlations.

3.2 Reduction of Point Data Sets



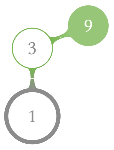
Adaptive Computation
Prune Steps Without Contribution (12)
Hierarchy Characteristics (14)
Computation Steps (11)
Selective Conversion
Trade Loads
Hardware Architecture (13)
Data Representation (9)
Condense Representation (10)
Utilize Special Capabilities
Cost (3)
Functionality Requirements
High-Performance Parallel Visualization
Dynamic Computations
Adaptive Granularity (16)
Structure (2)
Granularity (7)
Priority
Adaptability
Resources
Execution Frequencies
Dynamic Resource Utilization
Dependencies (5)
Structure Hierarchically (15)
Fitted Planning Frequency
Planning
Performance Model
Hybrid Modeling
High-Cost Dependency Minimization

Molecular dynamics is a widely used simulation technique to investigate material properties and structural changes under external forces. Like in temporal similarity visualization (Section 3.1), the increasing spatial and temporal extent (millions of particles and thousands of time steps) of the simulation domain poses a particular challenge for the visualization of the underlying processes. Reduction of these results is both important from a data handling perspective, and for improving the occlusion problem from a visualization perspective as exemplified in Fig. 3.9(a).

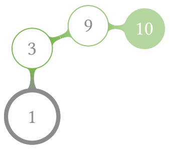
In this project, the key idea is to replace points by a smaller set of representatives (so-called sites) with a technique building upon K-means [Duda and Hart, 1973] and

CCVD [Balzer et al., 2009] (Fig. 3.9). Here, a group of points is represented by a site that is located at their average position (Fig. 3.9(b) and (c)). The size of the group varies in a user-defined capacity range. Starting from a random assignment of points to sites, two sites exchange a point if it is farther from the site it is originally assigned to and the capacity constraints are not violated. The exchange of points between sites proceeds in an iterative process until convergence.

Strategy 9 ***Data Representation***
Reduce the amount of data that is required to carry out a computation. This typically decreases data-induced cost like storage space and transfer load, as well as visualization effort.



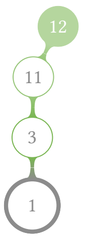
Strategy 10 ***Condense Representation***
Decrease the data size with the goal to preserve its contained information as far as possible. This can typically be applied to generic types of data and does not require manual selection or higher-level content-awareness.



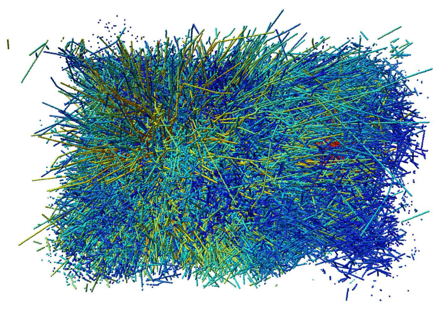
Potentially, all sites need to exchange points with all other sites. However, most do not exchange any points, but processing these superfluous cases alone would induce significant cost. To determine beforehand some site pair comparisons with no impact, we employ criteria on the basis of distances and recent activity.

Strategy 11 ***Computation Steps***
Reduce the computational steps to achieve a certain result by algorithmic optimization. It is crucial to carry out the additional computations required to achieve this in a cost-effective manner as well.

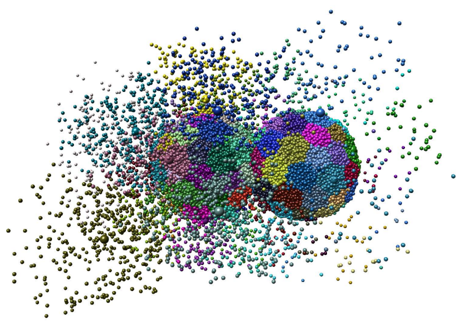
Strategy 12 ***Prune Steps Without Contribution***
Prune operations that are known beforehand to have no influence on the final result.



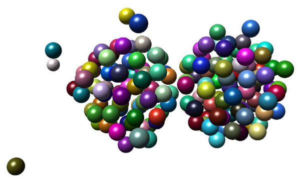
We utilize the hierarchical structure of a GPU and its execution characteristics for a cost-efficient mapping of the point-swapping algorithm. Processing the swaps between two sites in parallel needs to assure that no site is used twice to avoid conflicts. For this purpose, sites are partitioned using a k-d tree, and a swapping network is created for each site partition with the requirement that no site appears twice within one pass and the goal of a minimum total amount of passes. By executing spatial partitions of



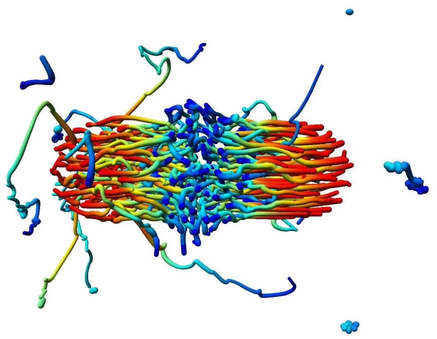
(a) Path lines of particles



(b) Grouped molecules for single time step



(c) LCCVD representatives for groups of (b)

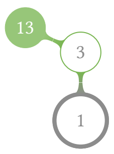


(d) Path lines of representatives

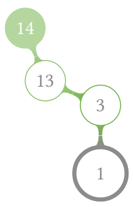
Figure 3.9: Reduction at the example of a molecular dynamics simulation data set featuring 81672 molecules over 600 time steps.

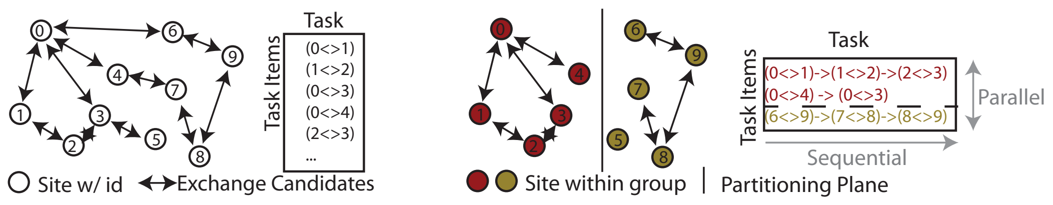
sites on the same SM by the threads of one warp, this is handled very efficiently by exploiting the lockstep property, i.e., no explicit synchronization is required to ensure that they are all in the same iteration.

Strategy 13 ***Hardware Architecture***
Explicitly consider the hardware architecture and its characteristics for optimizing an algorithm.



Strategy 14 ***Hierarchy Characteristics***
Adapt the algorithm to exploit the different characteristics present in different levels of hierarchical hardware architectures.



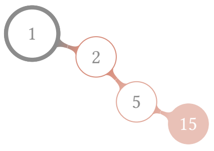


(a) Standard task items (b) Sequenced and Partitioned task items

Figure 3.10: Task item sequencing (Strat. 16, *Adaptive Granularity*) and partitioning for reducing dependencies (Strat. 15, *Structure Hierarchically*).

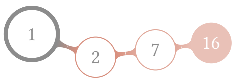
In our implementation, one task item is generated for each candidate site pair (Fig. 3.10(a)). Task items have a complex dependency structure, that needs to be mapped efficiently to the multi-level hardware architecture of a GPU. For this, the sites are spatially partitioned into sets of a certain size using a k-d tree (Fig. 3.14). This allows to restrict swapping operations to groups of adjacent sites (Fig. 3.10(b)), i.e., each set can be processed independently from the other sets on different SMs. This partitioning needs to be varied, still grouping nearby sites, such that all sites are able to at least once exchange points with all sites in their proximity. Swapping with more distant sites occurs indirectly by successively handing over points from site to site.

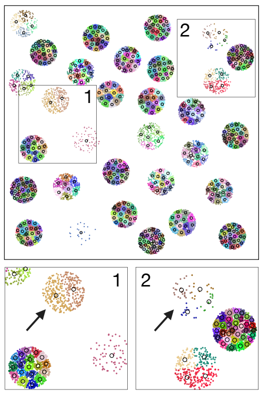
Strategy 15 ***Structure Hierarchically***
Partition the task items into dependency groups that can be processed independently. This can be accomplished by dividing the spatial or computational domain.



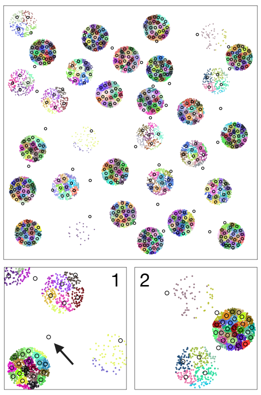
Several task items are further merged in order to reduce the dependencies that need to be handled, yet still leaving enough parallel work to fully occupy a GPU (Fig. 3.10(b), sequential). In detail, we consolidate them such that there are exactly as many task items as threads in a warp (32 or 64 for modern GPUs). This allows a SM to process such a consolidated task item in parallel without having to consider any dependency relations.

Strategy 16 ***Adaptive Granularity***
Consolidate task items to reduce interdependencies between task items (within the same or across different tasks), i.e., by doing an explicit serialization. This can significantly increase the effort for dealing with dependencies, but also decreases the degree of parallelism.

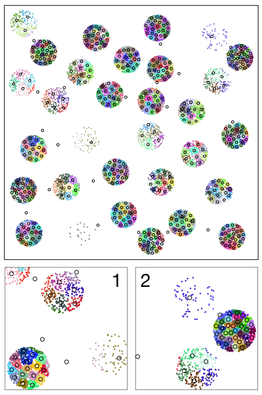




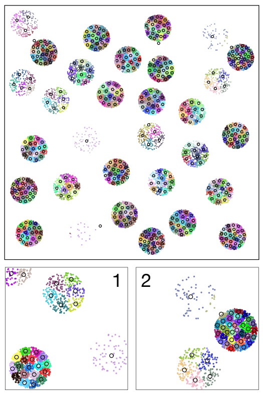
(a) No Constraints (K-Means) $l = \infty$



(b) Strict Constraints (CCVD) $l = 0$



(c) Loose Constraints (LCCVD) $l = 0.15$



(d) LCCVD, $l = 0.15$ and c_{min}-relaxation

Figure 3.11: Two-dimensional point data set represented by sites using LCCVD. Sites are depicted by black circles while points are shown as colored dots. Different colors depict that points belong to different sites. *No constraints* and CCVD do not represent the density faithfully due to a largely varying number of representatives for a certain number of points (a) or representatives in inadequate locations (b). These issues can be avoided with LCCVD (c)–(d).

3.2.1 Loose Capacity-constrained Voronoi Diagrams

The goal of Loose Capacity-constrained Voronoi Diagrams (LCCVD) is to drastically reduce the amount of points very quickly using a GPU-friendly algorithm, still preserving the basic structure of the data set (Strat. 9, *Data Representation*). Loose capacity constraints means that the number of points assigned to each site is not fixed, but may reside within an interval $[c_{min}, c_{max}]$. In the following, this is also given in terms of the *capacity looseness* l which translates to the interval by $c_{min|max} = \max(m \cdot (1 \pm l), 1)$, with m denoting the average number of points per site. A setting that has proven useful during our experiments is $l = 0.2$, i.e., the capacity interval allows a 20% deviation from m. Accordingly, LCCVD can be seen as a flexible hybrid between the CCVD-based method and the K-means approach.

The advantages of LCCVD over its originating techniques CCVD [Balzer et al., 2009] and K-means clustering [Duda and Hart, 1973] are demonstrated exemplarily in Fig. 3.11. Applying strict capacity constraints as in CCVD may result in sites being located inappropriately in-between accumulations of points, making them poor representatives for their sets of associated points. K-means clustering does not share this problem, but instead does not allow to draw any conclusions about the underlying point density. This is emphasized by the closeup images where the top groups of points are represented by either too few or too many sites. Figs. 3.11(c) and (d) show that these issues can be avoided with our LCCVD approach.

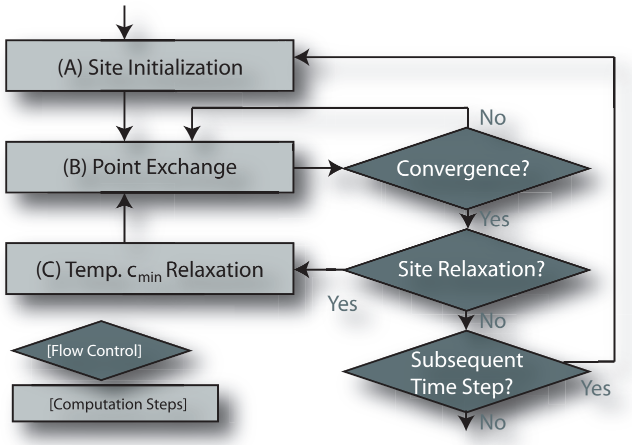


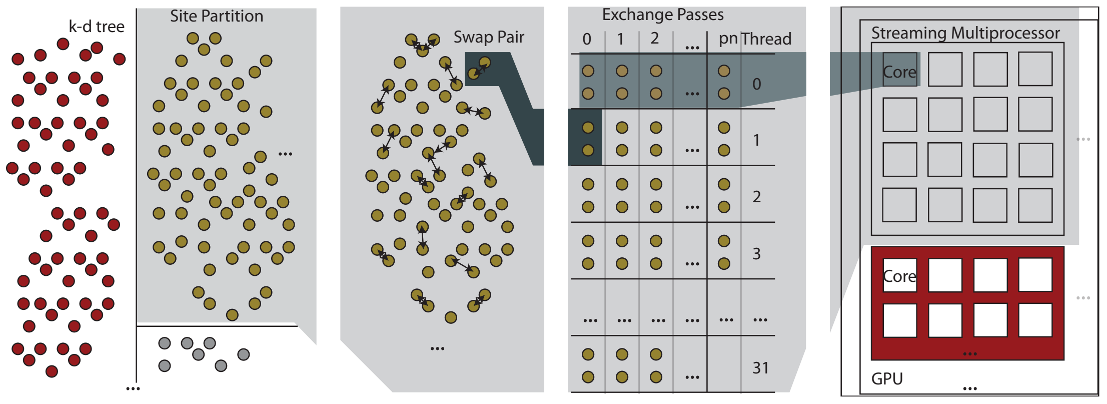
Figure 3.12: Computation steps and control flow of LCCVD.

3.2.2 Generating Representative Sites

Prior to the computation, the number of generated sites is calculated based on the desired average number of points that should be represented by a site. The user-defined capacity interval allows to span the whole range from the pure distance-based K-means approach ($c_{min} = 0$ and $c_{max} = \infty$) to the strict capacity-constrained approach ($c_{min} = c_{max}$). Then, points are initially assigned to sites (Fig. 3.12). Points are exchanged between sites until convergence (i.e., no points are exchanged anymore, sites only move insignificantly, or a certain time or iteration limit is reached). This is the most expensive part of the algorithm, thus most optimization effort goes here. When it is acceptable to spend more time on the computation to achieve better results, we further perform a step called temporary c_{min} relaxation which temporarily ignores the minimum constraint to allow an even better adaptation of sites to points (Fig. 3.11(d)). This is followed by another point exchange phase. For time-dependent data sets, the whole procedure is performed for each time step, using the results from the previous time step as initialization to exploit coherency.

(A) Site Initialization

We use the input point set to determine the initial site positions. For static data sets (and for the first step of a time-dependent series), sites are initially placed at the locations of randomly chosen points. To fulfill its minimum capacity constraint, each site then searches for the c_{min} nearest points that have not yet been assigned to another site. The remaining points are then assigned to the closest site which has not yet reached its maximum capacity constraint c_{max}. The necessary nearest-neighbor queries are



(a) Partition Sites (b) Identify Swap Pairs (c) Create Swap Network (d) Swap Points and update sites on GPU

Figure 3.13: Overview on the LCCVD parallel point exchange. Selected connections between the three steps are indicated by dashed lines.

efficiently performed using a k-d tree, removing points which have been assigned to sites, or sites which have reached c_{max}, respectively. For subsequent steps of a time-dependent data set, the assignments of points to sites are passed on from the previous time step, just updating the site positions using the mean position of all assigned points.

(B) Point Exchange

Sites are partitioned into groups for parallel processing (illustrated in Fig. 3.13(a)). This partitioning is implemented by enumerating each site s such that its index i gives its respective leaf in a k-d tree of all sites S (Fig. 3.14), and then sorting them accordingly. To determine its index i, we search for the enclosing k-d tree node of site s by traversing the tree (starting from the root with $i = 0$). Whenever we descend to the left child, i remains unchanged, whenever we descend to the right child, i is increased by $P \cdot 2^{-h}$ where h denotes the level of the tree ($h = 0$ for the root node), and P stands for the underlying set of points. The traversal is stopped as soon as we reach a node that contains $m = |P|/|S|$ points or less. Subsequently, sites are sorted with respect to i using the in-place GPU radix sorting algorithm of Satish et al. [2009].

In order to avoid that sites always belong to the same group, we displace the k-d tree splitting planes in each iteration by applying an offset. For the displacement directions, we alternate between the main axial directions and the diagonal directions, while the displacement magnitude for each site—according to our experiments—should be roughly half the extent of the site group's bounding box (as determined without any displacements). Since it is impractical to displace the whole tree with all different group

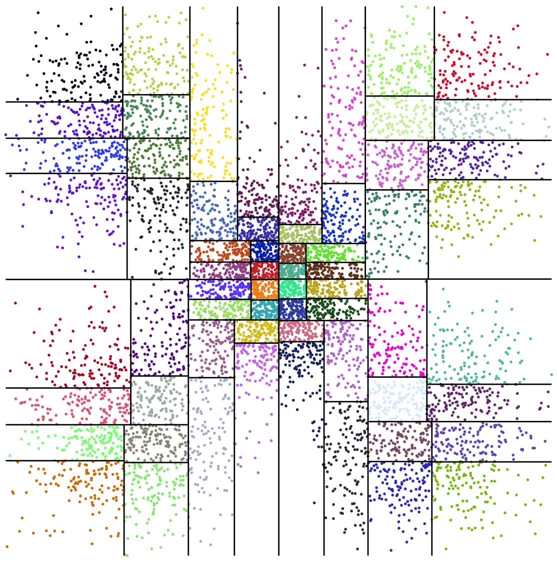


Figure 3.14: The partitioning of sites into site groups using a k-d tree.

extents (i.e., the size of the respective leaves) along all directions, we consolidate similar displacement magnitudes.

Instead of considering all pairs of sites within a group for point exchange, the set of pairs is pruned (Strat. 12, *Prune Steps Without Contribution*) by excluding sites which are guaranteed not to swap points Fig. 3.13(b)). For this, two criteria are used:

Bounding Sphere The distance between two sites is larger than the sum of the distances to their farthermost points. This means that all points are closer to the site they are assigned to than to the other one.

Stability A pair of sites has not exchanged points the last time it was processed, and the point assignment has not changed since for either of the sites.

The bounding sphere criterion is particularly beneficial when sites are roughly at their final position but not yet stable. The stability criterion has a strong impact during the final steps of the optimization when many sites have already reached stable positions.

While site groups are distributed over SMs, the swapping operations between sites (within a group) are executed in parallel by each SM (Fig. 3.13(d)). This design was chosen to optimally leverage the GPU architecture (Strat. 13, *Hardware Architecture*). For each site group, a set of consolidated task items is considered, each of which consists of some site exchange directives. In order to make use of the lockstep property of threads belonging to the same warp, for each site group as many (consolidated) task items are generated as there are threads of a warp (implying a 1-to-1 mapping of threads to task items). As the lockstep property already provides an implicit synchronization of processing steps within a task item, it further only needs to be ensured that no site is processed in more than one thread at a time (i.e., in the same step). To achieve this, a swap network is generated that schedules which site pairs are to be processed in parallel, and which are to be processed successively (Fig. 3.13(c)). It further maximizes the utilization of threads by generating task items with approximately equal numbers of sites.

In our GPU implementation, points are stored in an array with c_{max} elements or *slots*. Slots are placeholders for points from the data set such that each site can have at most c_{max} points. Some of these slots are *free slots* in the case that the number of points is smaller than c_{max}. Free slots can be exchanged between two sites instead of points as long as the minimum capacity constraint c_{min} for points is not violated. Initially, all points are located at the beginning of the array while free slots are located at the end. The *free slot index* indicates the slot from which no points are stored in the remaining array. A site is able to trade free slots for points as long as the free slot index does not point to the end of the array. When a point is exchanged for a free slot, the free slot is stored in the former location of the point. These free slots form gaps between occupied slots and are fixed the next time the algorithm iterates over the array: either the free slot forming the gap is used to store a point of another site, or it is swapped internally with the point just before the current free slot index. The free slot index is subsequently decremented until it points to the first free slot.

(C) Temporary Minimum Constraint Relaxation

One problem from CCVD is partly inherited by our LCCVD approach: sites may get positioned between adjacent point clouds (Fig. 3.11(c)), making this site a bad representative. This is due to the minimum constraint c_{min} which can prevent that points are removed from sites between two such clouds. Points cannot be swapped to another site either, since other points are even further away. We denote these problematic sites *bad sites* in the following.

We found that temporarily relaxing the minimum constraint for bad sites largely resolves this problem which is why we interpose an optional correction step after each exchange phase (cf. Fig. 3.12). During this correction step, we perform the following substeps:

1. **Identify bad sites:** A site s is considered a bad site when it is at its minimum capacity and its farthest point p is much closer to any other site s_{other}: $|s_{other} - p|/|s - p| < z$. In our experiments across all our data sets, $z = 0.85$ proved to reliably detect bad sites with only a small amount of false positives.

2. **Assign points of bad site to closest sites:** Release points to closer sites while constantly updating the site position. While this may temporarily violate c_{min}

for the bad site, the c_{max} constraint for the receptive sites remains intact. After this step, the bad site only represents the points it is closest to.

3. **Bad site takes on points from nearby sites:** Identify the closest sites and insert them into a priority queue based on their distance to the bad site. Take the first site from the queue and—as long as its minimum constraint is not violated—reassign points from it to the bad site in the order of proximity. Proceed with the next site from the queue until the bad site has reached its minimum capacity.

We initiate this correction step after each exchange phase (cf. Fig. 3.12), but no more than five times per time step (this proved to be a good tradeoff between speed and quality). This avoids infinite loops since it is not always possible to resolve a bad site without permanently violating c_{min}. Overall, the temporary relaxation of the minimum constraint allows to reduce the number of bad sites without losing the flexibility of loose capacity constraints.

3.2.3 Evaluation

This section presents the evaluation of our approach in different scenarios. First, we compare the performance of our method for computing strict CCVDs on the GPU to the original CPU-based method by Balzer et al. [2009] to show the advantages of our parallel algorithm. We then present LCCVD results, including comments from application domain experts, for real-world data sets using 3D molecular dynamics simulation and flow data. All measurements were done using a NVIDIA GTX 480 and an Intel Core i7. The partitioning of sites into groups and the swapping algorithm were implemented in CUDA using a block size of 128 threads and a group size of 128 sites. The remaining computational steps of LCCVD were executed on the CPU using OpenMP. For visualization, MegaMol is used, a package tailored toward molecular dynamics simulation using raycasting [Grottel et al., 2014].

CCVD of 2D Point Distributions

In their original work, Balzer et al. [2009] generated initial 2D point data sets by rejection sampling of a given density function. Among these were a constant density functions as well as $\rho = e^{(-20x^2-20y^2)} + 0.2\sin^2(\pi x)\sin^2(\pi y)$. Table 3.2 lists timings and quality results computed by the metrics *normalized radius* α [Lagae and Dutré, 2008] which should be around 0.75, and *capacity error* δ_c [Balzer et al., 2009] which optimally should be close to zero. Both metrics underline that our improved parallel algorithm does not sacrifice the quality of the resulting site distributions.

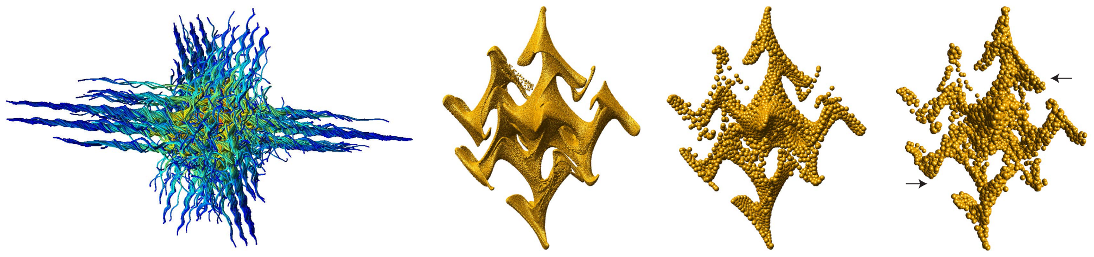
Sites Points Per Site	**Computation Time**		**Normalized Radius** α	
Constant	Balzer et al. [2009]	our	Balzer et al. [2009]	our
1024 4096	237.9s	129.7s	0.7628	0.7543
2048 4096	451.9s	152.3s	0.7481	0.7451
4096 4096	991.1s	175.6s	0.7470	0.7454
8192 4096	2413.3s	241.6s	0.7455	0.7588
16384 4096	6361.8s	525.7s	0.7367	0.7382
8192 8192	8319.1s	1258.0s	0.7576	0.7588
24576 1500	6720.4s	125.3s	0.7072	0.7035
ρ	**Computation Time**		**Capacity Error** δ_c	
1024 4096	214.6s	231.3s	0.00349	0.00346
2048 4096	421.9s	235.2s	0.00291	0.00318
4096 4096	876.6s	338.4s	0.00263	0.00304
8192 4096	1927.0s	542.4s	0.00245	0.00259
16384 4096	4911.7s	857.3s	0.00239	0.00246
8192 8192	6543.7s	2836.5s	0.00204	0.00220
24576 1500	2734.5s	158.7s	0.00333	0.00327

Table 3.2: Computation times and quality metrics (see [Frey et al., 2011] for details) for varying numbers of sites and points per site m using a constant and a non-constant two-dimensional density function. Optimally, $\alpha = 0.75$ and $\delta_c = 0$.

As expected, our parallel approach becomes more and more beneficial timing-wise as the number of sites increases. It does not slow down as drastically as the original implementation for large numbers of sites as the higher utilization of the GPU cushions the increased computation costs. Naturally, for a small number of points per site m the parallel nature of our approach shows its strength much more clearly, with speedups of one to two orders of magnitude. For large m, speedup is hampered by the less sophisticated selection of point swapping candidates, which is due to the requirements of an efficient GPU implementation.

LCCVD of 3D Molecular Dynamics Data Sets

The effectiveness of our approach is demonstrated by means of a data set from particle tracing for vector field visualization—Arnold-Beltrami-Childress (ABC, with $A = \sqrt{3}$, $B = \sqrt{2}$, $C = 1$ and $T = -8$) shown in Fig. 3.15—and two molecular simulation data sets: compressed argon surrounded by vacuum (Fig. 3.16), and two colliding liquid droplets (methane and ethane) (Fig. 3.17). Refer to [Frey et al., 2011] for a more detailed evaluation including expert feedback. Particle numbers and the amount of time steps per data set are listed in Table 3.3. Visualizing sites instead of points has numerous benefits apart from rendering speed and storage requirements. For example, Fig. 3.15(a) illustrates the structure of the flow of the ABC data set. It can be seen that sites move smoothly over time as long as there are no rapid, incoherent movements in the data set. Fig. 3.15(c) shows that the density in different regions of the data set can be



(a) Sites temporal lines $l = 0.2$, $m = 8192$, $t = (0, 400)$ (b) Points, $t = 90$ (c) Sites, $m = 512$, $l = 0.2$, $t = 90$ (d) Sites, $m = 512$, random, $t = 90$

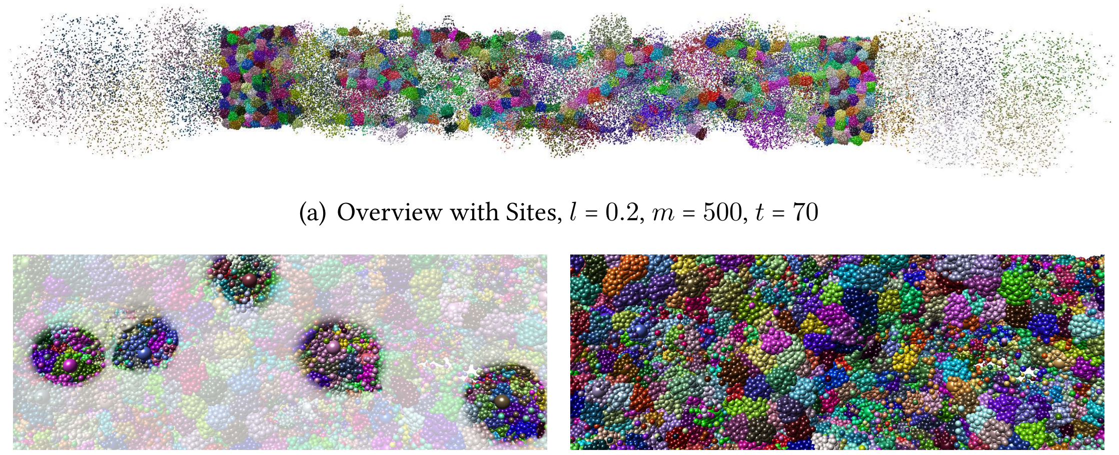
Figure 3.15: Arnold-Beltrami-Childress flow. *(a)* Data set represented by a set of sites over numerous time steps. *(b)* Full data set for a single time step t = 90. *(c)* Reduced version for t = 90 sites which represents m = 512 points on average. It fully preserves the basic structure and allows better insight into the data set, e.g. the point density at the left is much lower than in the middle or on the right. *(d)* Reduced version based on random sampling exhibits an irregular structure that does not preserve densities and results in the loss of smaller features (e.g. thin structures on the bottom left and top right indicated by arrows).

estimated much better with a set of site representatives than with rendering points directly (Fig. 3.15(b)). It also demonstrates that the basic structure of the data set is preserved even when using a drastically reduced amount of points. Random sampling (Fig. 3.15(d)) does not preserve the data set structure well and results in the loss of many small features.

Fig. 3.16 illustrates that a strict capacity constraint ($l = 0$) potentially forces sites to represent points from two or more dense clusters. This leads to sites floating in-between clusters of points such that they are located where no associated points are (e.g. the purple site on the left, or the green site on the right in Fig. 3.16(b)). Loosening the capacity constraint using a value of $l = 0.2$ and temporarily relaxing the minimum constraint as described in Section 3.2.2 largely avoids these issues (see Fig. 3.16(c)).

However, constraints that are too loose may lead to an over- or underrepresentation of point density, i.e., regions where a site either represents a too small or too big portion of the data set. Fig. 3.17 demonstrates overrepresentation for the methane-ethane collision data set with a very loose constraint of $l = 5$ (Fig. 3.17(a)). Thus, when displaying sites only, the surrounding of the droplets appears much more dense than it actually is. This way, temporal lines generated with a very loose constraint (Fig. 3.17(b)) give the impression of a much larger amount of points being spread (Fig. 3.17(d)).

These observations from the example data sets are underlined by our quality metric assessing point coverage. Roughly speaking, it only operates on the sets of points and sites without any additional information like point-to-site and assignments, and determines how adequately the sites represent the point distribution (see [Frey et al.,



(a) Overview with Sites, $l = 0.2$, $m = 500$, $t = 70$

(b) Closeup for $l = 0$, $m = 50$ with inappropriate sites highlighted

(c) Closeup for $l = 0.2$, $m = 50$

Figure 3.16: Argon in vacuum. Points belonging to the same site are shown with the same color. (a) Overview over the reduced version. (b) Strict capacity constraints (l = 0) force inappropriate site locations between dense point groups, falsely creating the impression of occupied space. (c) Loose constraints (l = 0.2) largely remedy this issue.

2011] for details). For each data set, Table 3.3 lists both the quality q and the associated computation times while varying the capacity looseness l from $l = 5$ (almost unconstrained) to $l = 0$ (strictly constrained). For better comparability, we omitted the temporary relaxation of the minimum constraint as described in Section 3.2.2 for this test series. Across all data sets, the best results were obtained by applying a loose capacity constraint of $l \approx 0.2$ despite the variations due to different data sets or site configurations; $l = 0.1$ delivered nearly as good results and might be favorable if stricter bounds are required. Note that a difference in the quality metric of 0.001 is equal to the difference of a thousand points being completely covered or uncovered in a data set of a million points. Smaller quality values thus either indicate poorly located sites, or an inappropriate amount of sites covering a particular part of the data set. As demonstrated in the examples, these cases typically occur in regions with a rapid change in point density. In turn, we measure negligible differences for our example data sets for regions of approximately constant density. Quality results for a statistical sampling-based approach (provided for comparison) are typically around ≈ 0.62.

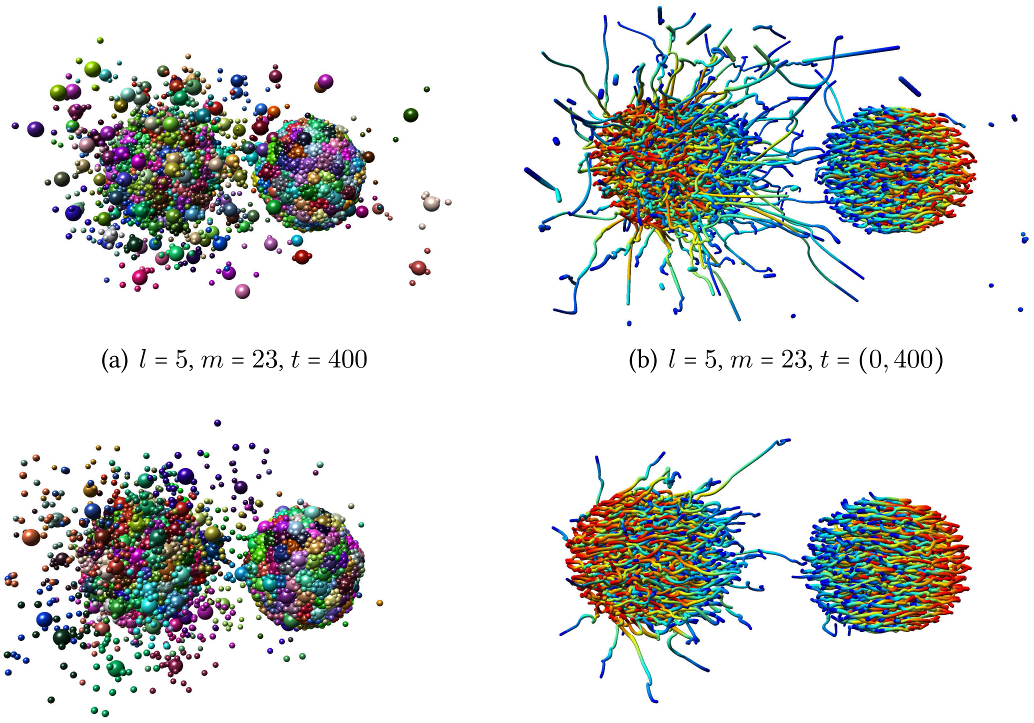
The timing results in Table 3.3 underline that the computation time for LCCVD strongly depends on the amount of points per site m. Here, the main reason for this is that the GPU load decreases with a decreasing number of sites. For example, a GTX 480 features 15 SMs, each of which can execute two warps concurrently. As each warp processes point swapping operations in groups of 128 sites, any number of sites below $15 \cdot 2 \cdot 128 = 3840$ is theoretically not able to fully utilize the GPU. In order to hide

Capacity Looseness l					
$l = 5.0$	$l = 0.5$	$l = 0.3$	$l = 0.2$	$l = 0.1$	$l = 0$
Argon in vacuum, 2000000 points, 100 time steps, periodic					
4000 sites, m = 500 points per site (random sampling: .61829)					
.02709	.00549	.00159	.00006	**.86951**	.01683
11162.6s	17157.7s	16371.1s	15289.4s	14469.7s	14028.0s
20000 sites, m = 100 points per site (random sampling: .62090)					
.03593	.00806	.00182	**.86262**	.00040	.02100
9005.7s	7476.3	7374.2s	7306.4s	7012.6s	6412.8s
40000 sites, m = 50 points per site (random sampling: .62269)					
.04639	0.00957	.00183	**.85439**	.00073	.02050
9003.1s	10038.3s	10159.0s	9986.1s	9846.4s	9572.3s
Laser Ablation, 562500 points, 400 time steps, periodic					
1125 sites, m = 500 points per site (random sampling: .63140)					
.00723	.00234	0.00057	**.89489**	.00136	.02374
10545.7s	10066.8s	9419.4s	8753.3s	9460.2s	4559.7s
11250 sites, m = 50 points per site (random sampling: .63249)					
.03189	.00698	.00190	**.89575**	.00026	.03704
4497.3s	4529.9s	3801.6s	4476.6s	4438.1s	5954.0s
Methane-Ethane Collision, 81672 points, 1782 time steps, periodic					
3403 sites, m = 24 points per site (random sampling: .61536)					
.17199	.019660	.00369	**.85184**	.00149	.07216
1586.8s	1566.4s	1575.2s	1575.0s	1984.2s	1476.4s
1992 sites, m = 41 points per site (random sampling: .61234)					
.17063	.02065	.00299	.00007	**.84938**	.02328
1894.3s	1847.6s	2228.1s	1865.8s	1837.3s	1633.8s
ABC, 2097152 points, 400 time steps					
16384 sites, m = 128 points per site (random sampling: .62235)					
.02554	.01221	.00335	**.85882**	.00018	.01205
4002.4s	4195.0s	4178.9s	4135.7s	3974.0s	4217.0s
4096 sites, m = 512 points per site (random sampling: .62183)					
.06424	.02968	.01034	.00307	**.86803**	0.00810
6566.4s	8393.2s	8180.8s	7886.8s	7027.9s	5124.9s

Table 3.3: Performance of LCCVD for different data sets, loose constraints l and no c_{min} relaxation. The top rows depicts the best (meaning largest) quality results in bold while the other results are offset with respect to this reference value. The bottom rows give the computation times in seconds. Additionally, quality results for random sampling are provided for comparison.

latencies, the actual number of sites should even be significantly higher since warps may be paused or stalled. In contrast to the number of points per site m, the capacity looseness l only has minor influence on the runtime.

Lastly, we measured the effect of the temporary c_{min} relaxation compared to the best quality values listed in Table 3.3. In general, the technique is most beneficial for data sets which induce the generation of bad sites—e.g., due to multiple groups of points of varying density—as discussed in Section 3.2.2. This particularly applies to the argon in vacuum data set (Fig. 3.16). In this scenario, the quality q can approximately be improved by 0.01 for $m = 50$. At the same time, however, the execution time is almost tripled to 28409s. For data sets with significantly less bad sites, e.g., the laser ablation data set, the quality improvement is only about $.001$ on average at roughly twice the execution time. The coverage quality of the methane-ethane collision data set with 3403 and 1992 sites increases by $.00153$ and $.00055$ respectively, while the runtime roughly doubles. Furthermore, we observed that the processing time as well as the quality value achieved for a single time step is largely independent of whether it has been computed



(a) $l = 5, m = 23, t = 400$
(b) $l = 5, m = 23, t = (0, 400)$
(c) $l = 0.2, m = 23, t = 400$
(d) $l = 0.2, m = 23, t = (0, 400)$

Figure 3.17: Collision of methane and ethane. (a)–(b) Very loose or no capacity constraints lead to points that are highly overrepresented by sites which gives the false impression of a substantial amount of particles in the outer regions. (c)–(d) A loose constraint of l = 0.2 yields a much more genuine result.

as part of a time series or individually. In some cases, however, time steps which are part of a series are processed significantly faster if the changes between two subsequent time steps are rather small. In such a case, the site-to-point assignment of the new time step only requires minor adjustments compared to the previous step of the series.

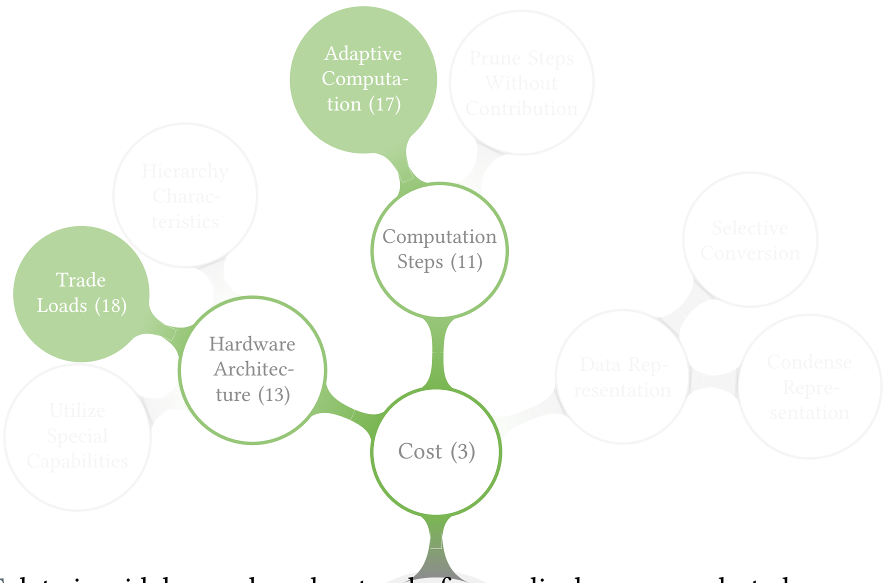
3.2.4 Directions for Further Research

In the current approach, there are only two levels of detail: points and sites. The extension of LCCVD to build multi-level hierarchical structures from large point data sets would be able to capture the density distribution adequately for a wide range of usage scenarios, even when considering very large data sets. Furthermore, reducing the amount of points not only in the spatial but also across the temporal dimension would allow for smaller data sizes, and might further be helpful in the analysis of time-dependent particle movement. Sites and temporal lines could further be enhanced such that they visually indicate the extent of the spatial distribution of the points they represent. Temporal lines could additionally illustrate their coherency with respect to their respective points, e.g., by means of opacity. Finally, more elaborate site partitioning schemes could be investigated, e.g., based on connected components analysis.

Chapter 4
Accelerating Raycasting

Volume raycasting is a fundamental core technique in scientific visualization. While it has some favorable properties, like great flexibility and good parallelization characteristics, it is also computationally expensive. Accordingly, even when using high-end GPUs, it is challenging to achieve interactivity for larger volumes on a single machine. A simple approach to alleviate this issue is an overall reduction in quality, e.g., by uniformly increasing the sampling distance. To ensure a more efficient resource usage, rendering effort can also be focussed on regions with higher entropy. Furthermore, even when used efficiently, the available processing capabilities might not suffice to consistently meet certain criteria, like an acceptable frame rate for fluent interaction. In such cases, dynamic approaches are required to adapt on-the-fly to runtime conditions.

4.1 Exploit Volume Segmentation of Industrial CT Data

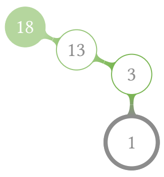

Adaptive Computa-tion (17)
Prune Steps Without Contribution
Hierarchy Charac-teristics
Computation Steps (11)
Selective Conversion
Trade Loads (18)
Hardware Architec-ture (13)
Data Rep-resentation
Condense Repre-sentation
Utilize Special Capabilities
Cost (3)

Visual inspection of CT data is widely employed not only for medical purposes, but also for industrial material testing and quality assurance. However, raycasting of big volumes only achieves low frame rates, thus impairing interactive exploration, even when using modern high-end GPUs. To overcome this issue, we exploit the property that industrial CT data typically exhibits large homogeneous regions which can be sampled sparsely without introducing significant errors. In detail, homogeneous regions are detected by means of volume segmentation, and rays use larger step sizes when passing through them. The acceleration information derived from these detected homogeneous regions is encoded in a data structure. It is accessed during rendering to check how far a ray can be safely advanced to take its next sample.

Strategy 17 ***Adaptive Computation***
Adapt the invested computation in certain areas or regions with respect to the underlying data, previous results, or intermediate outcomes.

Volumetric raycasting is typically memory bound, i.e., the bottleneck are the texture accesses required for sampling the volume, applying a transfer function, performing lighting and so on. Additional data fetches are usually required for the acceleration structure, which can have a significant negative impact on performance. For this reason, we design our acceleration approach to cause no extra texture lookups, at the cost of only marginal computational and storage overhead by integrating volume data and acceleration information in the same data structure. Despite this integration, the graphics card's hardware can still be used for trilinear interpolation of density values without producing incorrect results, although care must be taken to not interpolate between density and acceleration values.

Strategy 18 ***Trade Loads***
Substitute computational work for memory lookups and vice versa to optimize the inherent trade-off with respect to the hardware architecture and the program characteristics.



4.1.1 Integrated Data Structure

Our data structure consists of three types of voxels (Fig. 4.1). *Leap* voxels give the distance that the ray can safely skip without leaving the homogeneous region. *Unmodified* voxels simply keep their density value from the original volume. In any case, the most significant bit of each voxel is reserved to classify its value either as density (0) or leap value (1). Leap and guard voxels are situated in homogeneous regions, which we generate by using a fully automatic variant of the 3D flood fill algorithm (any other approach would work as well). We use the maximum range of values as segmentation criterion, but also more specialized constraints could be applied. *Guard* voxels contain the (average) density value of homogeneous regions and are situated in a layer around the leap voxels. Their purpose is to detect the transition between *unmodified* and *leap* voxels. The thickness of the layer depends on the standard sampling distance for *unmodified* areas.

During raycasting, density values are sampled with trilinear interpolation, but interpolation between leaping and density values needs to be prevented as this would lead to invalid results. It is thus critical for the proposed acceleration technique that no leap voxel can be reached without switching to nearest neighbor interpolation. For this purpose, guard values are located at the border of each homogeneous region, such that it can be guaranteed that two consecutive samples of a ray entering the region exhibit their very density value. This then triggers the switch.

To determine the values of leap voxels, we first determine the voxels at the boundary of the respective homogeneous region. Next, these boundary voxels are organized in a k-d tree, which allows to efficiently determine the value of leap voxels by a nearest point request. As even these k-d tree nearest neighbor requests consume a significant amount of pre-processing time, we employ an approximation that reuses a leap value for a whole group of voxels around a so-called center voxel. Voxels that are classified as leap voxels only become center voxels with a certain, user-defined probability. A radius that is defined relative to the center voxel leap value determines its associated group

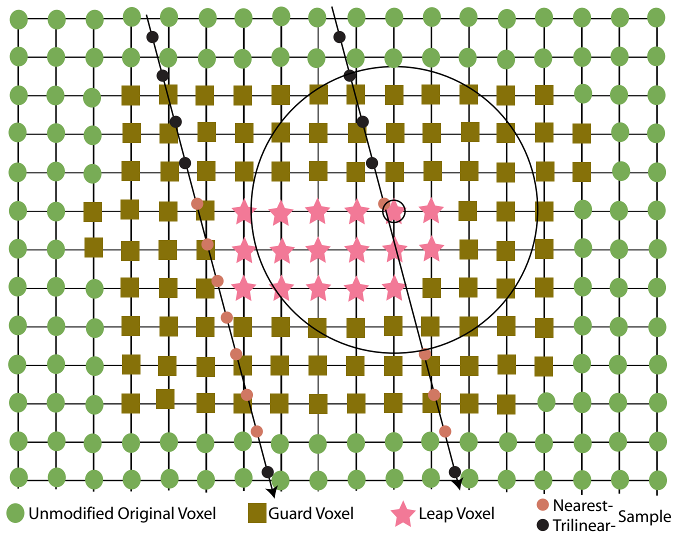


Figure 4.1: Two rays passing through a small region. The right one leaps and the associate leap value is displayed by the big circle around the respective leap voxel. Boundary voxels are merely used to calculate leap values.

of voxels. For the voxels of the group, the actual leap values can be underestimated, but must not be overestimated as this would allow a ray to leap out of the volume and thus potentially skip critical data. Thus, leap values for group voxels are determined by subtracting the leap value determined for the center voxel with the distance of the considered group voxel to the center voxel. Should they be part of another voxel group later on, the maximum of the already applied and the newly calculated leap value is utilized. In general, the bigger the center voxel probability or the smaller the relative radius is, the larger are the leap values but the more time is consumed by the classification and especially the leap value determination.

4.1.2 Raycasting the Modified Volume

Slight modifications to a standard raycaster are required to enable it to use the integrated data structure. First of all, it needs to correctly handle variable adaptive step sizes along a ray. In particular, the opacity has to be corrected depending on the length of a leap [Engel et al., 2004]. Furthermore, during raycasting, trilinear interpolation must exclusively be used between density voxels as the result is invalid when leap voxels are involved. Thus, it needs to be switched off virtually in some cases and a nearest neighbor access scheme must be used instead. This is triggered with the help of the guard voxels which enclose the leap voxels of their region. When passing through these guards from the outside, the interpolation mode is switched to nearest neighbor. Trilinear interpolation is switched back on again when first encountering a voxel that does not belong to the current region. A region voxel is either a leap voxel as indicated by the MSB or a guard voxel containing the regions average density (Figure 4.1). If a

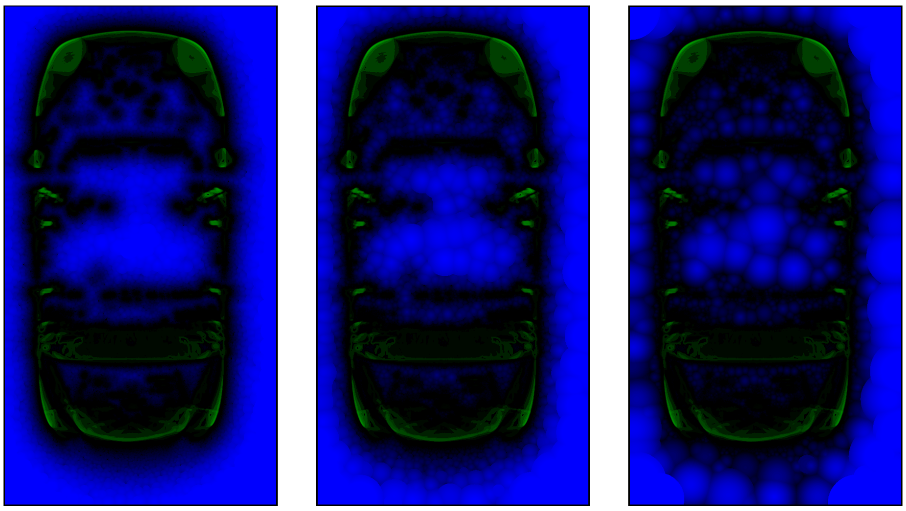

(a) Voxel Radius 0.25 (b) Voxel Radius 0.5 (c) Voxel Radius 1.0

Figure 4.2: Slices of the toy car dataset with different voxel radius settings. Blue and green denote leap and density values, respectively, with color intensity indicating their magnitude.

sample is taken in nearest-neighbor interpolation mode, the sample value is classified as leap or density value using the most significant bit. Accordingly, the raycaster needs to switch between nearest-neighbor and trilinear interpolation on the fly. This can be done efficiently by enabling trilinear interpolation per default and rounding texture access coordinates in the case for nearest-neighbor access.

4.1.3 Results

We tested our approach on a machine equipped with an Intel Core2 Quad CPU 2.4 GHz, 4 GB of RAM and a NVIDIA GeForce GTX 280. We use five data sets for evaluation: toy car ($559 \times 1023 \times 347$), engine ($256^3$) and ellipses ($256^3$), as well as Zeiss512 ($512^3$) and Zeiss768 ($768^3$). The toy car, engine and ellipses data sets are relatively noise free. The materials have homogeneous values, and transitions can clearly be distinguished. In contrast, Zeiss512 and Zeiss768 were reconstructed from noisy X-ray images without any smoothing, filtering or post-processing applied.

We evaluated the performance of generating the acceleration structure at the example of the Zeiss512 data set with varying segmentation ranges, group voxel radii and center voxel probabilities in a series of 125 measurements. The classification of the voxels and the determination of the leap values utilize four CPU cores with OpenMP, while the region segmentation employs a single core only. Without the center voxel modification, guard voxel classification takes five minutes and leap value determination one hour.

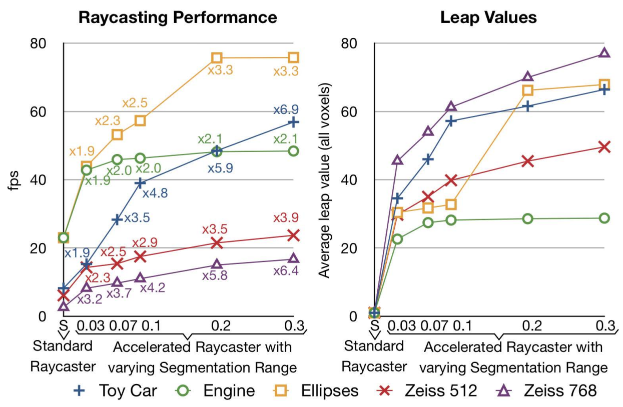
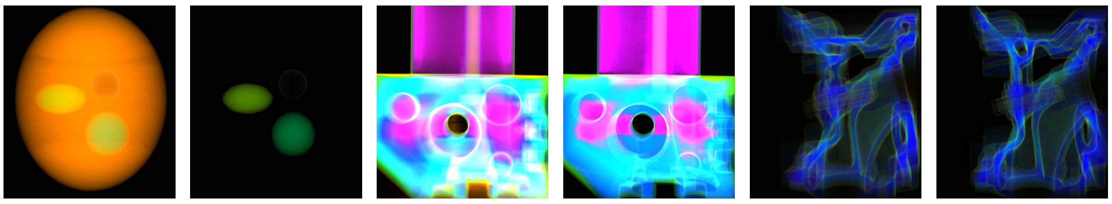


Figure 4.3: Performance of accelerated raycasting with respect to different data sets and segmentation ranges. *Left:* The increase of frame rate with wider segmentation ranges varies depending on the data set. Speedup compared to standard raycaster noted. *Right:* The speedup is directly linked with the leap values determined for the data set (given with respect to edge length of a voxel).

When employing the modification, computation time is reduced significantly to approximately 2.5 minutes and slightly varies with different group voxel radius and probability settings as well as the segmentation range. Contrary to the voxel classification and leap value determination, our exemplary region segmentation algorithm heavily depends on the segmentation range and takes from 1.5 minutes for big segmentation ranges (few regions) up to one hour for a tiny segmentation range (many regions).

The raycasting procedure was implemented using CUDA. The performance was measured for different region segmentation ranges with a fixed center voxel probability 0.25 as well as a fixed group voxel radius of 0.25. The impact of different group voxel radii on the actual leap values is visualized in Fig. 4.2. Fig. 4.3 shows that significant increases in performance can be achieved, even with the very noisy Zeiss data sets and conservative segmentation settings. It can also be seen that the segmentation range has a huge impact on the rendering performance as it influences how large regions grow, which again affects the leap values. However, an increase of segmentation range has a different effect for different volumes. This is due to the fact that the additional range needed for a substantial increase of the regions might be much larger in one data set than in another. For example, the engine data set consists naturally of very homogeneous regions and distinct transitions, so that the segmentation does not vary much with increasing value ranges and the maximum leap potential is almost reached already with a low range. Likewise, the visible difference between two renderings of the same volume with different range values depends on the data set. For the tested data sets, the results of the standard raycaster and the accelerated raycaster exhibit virtually no visual differences for small segmentation ranges. However, big ranges potentially lead to significant differences depending on the data set (Figure 4.4).



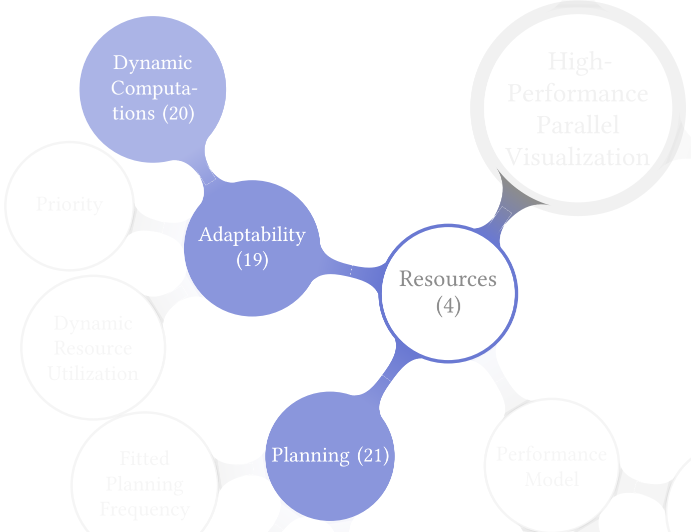
(a) Ellipse, 3% (b) Ellipse, 20% (c) Zeiss768, 3% (d) Zeiss768,20% (e) Engine, 3% (f) Engine, 20%

Figure 4.4: Comparison of renderings of data sets with a small and a big segmentation range.

4.1.4 Directions for Further Research

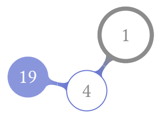
The integrated data structure only encodes unidirectional leap values. Taking the ray direction into account here, e.g., by classifying it into different direction categories, would allow to reduce the amount of samples required per ray even further. However, this would require implementing non-linear, or region-dependent leap values, as one 16 bit variable then needs to store multiple leap values. A small lookup in GPU shared memory could be utilized to execute this efficiently. The proposed acceleration algorithm could also be combined with out-of-core approaches and distributed volume rendering schemes to allow for the visualization of huge volumes exceeding the GPU's main memory. This would not require any changes to the data structure and only minor modifications to the raycasting kernel.

4.2 Spatio-Temporal Error-based Dynamic Progressive Rendering for Interactive Visualization

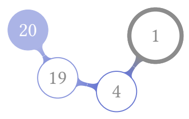

Dynamic Computa-tions (20)
Adaptability (19)
Resources (4)
Planning (21)
High-Performance Parallel Visualization

Despite all efforts toward optimization and growing computational power, volumetric raycasting is still very time-consuming in many scenarios. To alleviate this issue and still enable fluent interactive exploration in these cases, the computational cost can be reduced during interaction at the cost of reducing the rendering quality.

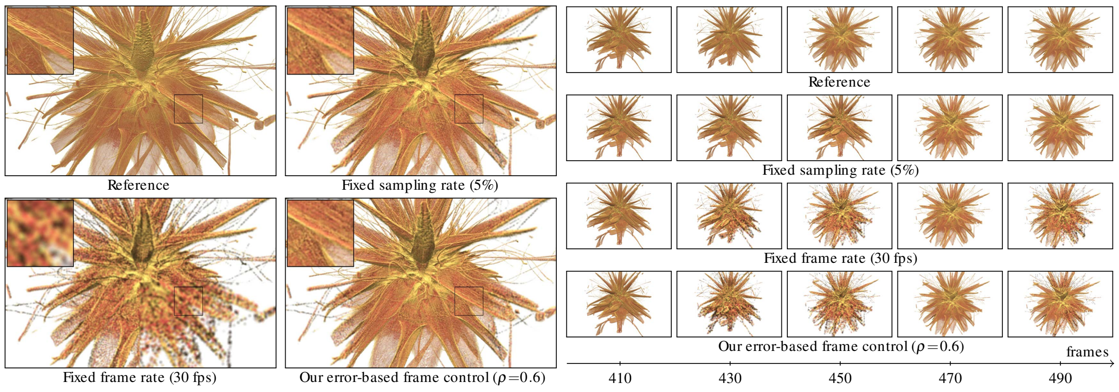
Strategy 19 ***Adaptability***
Provide the flexibility to allow for dynamic adjustment to changing input. In doing so, consider the trade-off between the flexibility and hand-tuned optimization for a fixed setup. Increased flexibility usually also results in higher implementation complexity.



Strategy 20 ***Dynamic Computations***
Allow dynamic adaptations regarding the amount and order of computational work.



Static settings toward a certain image quality or frame rate during interaction are popularly used, but are not flexible enough to avoid significant shortcomings regarding user utility in a variety of situations. We propose a dynamic model, in which we distinguish between spatial and temporal errors, that typically occur from insufficient image-space sampling and response delays, respectively (Fig. 4.5). The correlation of these errors is the basis of the dynamic steering of *interrupt*, *show*, and *power*, with sub-frame granularity. While spatial and temporal error are computed using fast heuristics

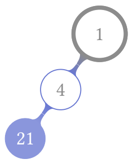

Reference
Fixed sampling rate (5%)
Fixed frame rate (30 fps)
Our error-based frame control ($\rho = 0.6$)
Reference
Fixed sampling rate (5%)
Fixed frame rate (30 fps)
Our error-based frame control ($\rho = 0.6$)
frames
410
430
450
470
490

(a) Slow movement: strong loss of detail for fixed frame rate (frame 1280 shown). (b) Rapid movement: severe lag for fixed sampling rate (e.g., at frame 450).

Figure 4.5: In interactive visualization, different sampling rate or frame settings are desirable for different user actions (see Sec. 4.2.3 for our definition of sampling rate). In contrast to static frame rate or sampling rate settings, our error-based approach adjusts dynamically to optimize the utility for a user. The frame numbers are given with respect to the video used for automatic evaluation in Sec. 4.2.4 (30 frames per second).

for on-the-fly usage, we also incorporate elaborate offline video quality analysis for optimizing the error correlation.

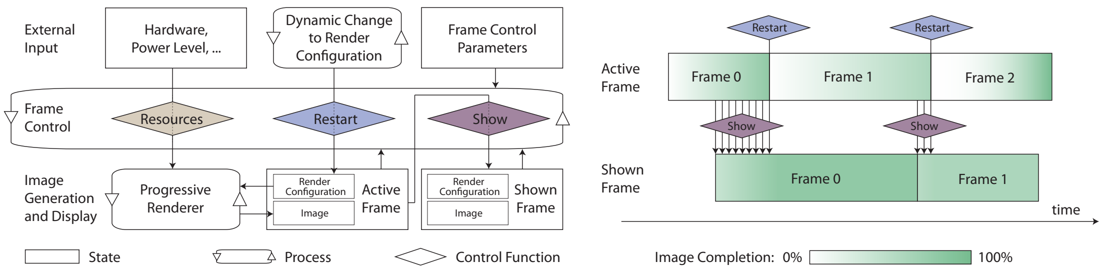
Strategy 21 ***Planning***
The planning process adjusts the spending of resources toward a certain goal, i.e., minimal computation time. The inputs to the planning are the identified key factors (Strat. 29, Performance Model) and the outputs are adjustments to the system (Strat. 19, Adaptability). This procedure is typically required to deliver good results at the cost of only minor computational overhead.



4.2.1 Progressive Visualization Model

Our technique is based on an idealized model of interactive progressive visualization. We derived it from a simple standard workstation setup with a single display. Its extension to setups with more than one camera like stereo rendering, more than one display like multiprojector configurations, or limited display throughput like in remote rendering, is beyond the scope of this paper and remains for future work.

Our model (Fig. 4.6(a)) consists of three basic processes: *dynamic change*, *progressive renderer*, and *frame control*. *Dynamic change* comprises the factors that alter a *render configuration* over time, like user interaction or change due to time-varying data. In this work, we address frame-based progressive visualization by means of a *progressive*



(a) Generic model of interactive progressive visualization.

(b) Exemplified process of double-buffered progressive visualization.

Figure 4.6: (a) Progressive visualization model and (b) an illustrating example. *Restart* triggers when to stop the refinement of the *active frame* (back buffer) and start computing a new one instead with the current *render configuration*. *Show* determines when to copy the *active frame* to the *shown frame* (front buffer) for display. *Resources* controls the share of the compute capacity that is consumed by the *progressive renderer*.

renderer. This is by far the most widespread variant, an alternative being frameless rendering techniques [Bishop et al., 1994; Dayal et al., 2005]. The *progressive renderer* continually refines the *active frame*, which reflects an individual *render configuration*. In this sense, a frame consists of both an (intermediate) rendering result of the *active frame* together with the *render configuration* of the *active frame*.

The model follows the principle of double buffering (Fig. 4.6(b)). In progressive visualization, a rendering result of the *active frame* is typically already shown while the *active frame* is further refined in the background. In our model, this is triggered by the *show* control function, which copies the *active frame* to the *shown frame* including the respective *render configuration*. If the *active frame* changes through an issued *restart* control function, i.e., the *active frame* is supplied with a new *render configuration*, the *progressive renderer* immediately starts to render a new image of the *active frame* from scratch. How fast the refinement advances can vary significantly, both depending on the *render configuration* (e.g., data set or camera position) as well as the compute resources allotted by the *resources* control function.

In total, our model requires three basic decisions: when to *restart* an *active frame*, when to *show* it, and what share of *resources* to consume in the *progressive renderer*. These three decisions are managed by *frame control*, using the available information about the (previous states of the) interactive system, and based on the *frame control parameters*. In these terms, (traditional) fixed-quality settings only consider the quality of the rendered image of the *active frame*, while (traditional) fixed-frame-rate settings take only into account the time stamp of the last rendered image. As long as there are no changes to the *render configuration*, i.e., *dynamic change* is idle, the image is progressively refined beyond the quality or frame rate limits.

4.2.2 Error-Based Frame Control

The fixed limits in typical progressive visualization systems cannot adapt flexibly to *dynamic change* due to camera rotations, transfer function changes, progressing time-dependent data, etc. As exemplified in Fig. 4.5, this can lead to significant shortcomings in compute-intense interactive visualization sessions. In this section, we introduce a new dynamic *frame control* approach, based on sampling error and response delay, to overcome these issues.

Motivation and Overview

The main goal of our error-based *frame control* is to minimize the perceived error in the *shown frame* over the course of an interactive visualization session. The error may additionally be balanced against the utilization of resources, which, however, heavily depends on the hardware-related setting, involving several questions. For instance, are the resources shared with other processes or users in the context of a remote visualization server? How are priorities determined? Is power consumption above a certain threshold problematic in the context of mobile devices? Under which circumstances may this threshold be exceeded temporally? While our progressive visualization model can cover these aspects, we use a simplified resource model in our exemplary implementation—a detailed consideration of these questions for different scenarios has to remain for future work.

In this paper, we focus on optimizing *frame control* by minimizing the error during interaction. Hence, offline optimization, i.e., a posteriori error evaluation by means of user studies or video quality metrics [Seshadrinathan and Bovik, 2010; Aydin et al., 2010; Sheikh and Bovik, 2006] is not practicable. Instead, error minimization by *frame control* has to take place online, during user interaction, and introduce very low overhead to avoid delay. This means that neither costly quality evaluation algorithms nor constant manual user adjustment are affordable during interactive visualization.

As a consequence, we split the quality evaluation into an online component (*spatial error estimator* and *temporal error estimator*) and an offline component (*frame control parameters* optimization). Akin to the notion of spatial and temporal errors from video encoders and quality metrics (e.g., [Seshadrinathan and Bovik, 2010]), our online component consists of a *spatial error estimator* and a *temporal error estimator* (Sec. 4.2.2), both providing error estimates at very low overhead. These estimates are then used by our heuristics to operate *restart*, *show*, and *resources* (Sec. 4.2.2). The heuristics itself is steered by the *frame control parameters*, which are determined and optimized by means of user adjustment, video quality analysis algorithms (Sec. 4.2.2), or user studies.

Space-Time Error Estimation

In an interactive visualization system, the temporal error τ shall reflect delayed response to changes, while the spatial error ς reflects compromises in image quality. Naturally, ς and τ are subject to a trade-off: while ς is typically monotonically decreasing with the time spent for rendering, τ is monotonically increasing. This partitioning into two types of error is a good fit for the frame generation pipeline in interactive visualization. The easiest way to implement the *spatial error estimator* would be to directly use the sampling density (e.g., [Woolley et al., 2003]). However, we take a more expressive, yet slightly more expensive content-aware approach that considers color variance (Sec. 4.2.3). The *temporal error estimator* can be derived directly from the difference between subsequent images, or indirectly by measures operating on subsequent render configurations. For instance, a simple indirect approach would be to project a precomputed set of vertices surrounding a geometric model to the screen, and then determine the largest difference to the previous projection [Woolley et al., 2003]. However, among other issues, this approach would ignore the degree of detail in the data. In contrast, we use a direct approach and assess the error analogously to the *spatial error estimator* directly from the images (Sec. 4.2.3). Inherently, this considers the spatial complexity of the data set and allows, e.g., to account for render configurations that are rich in detail by lowering the frame rate.

Control Heuristics

The temporal and spatial errors determined by the error estimators are used to control *restart*, *show*, and *resources*. For this, we employ the following heuristics. *Restart* strives to achieve the best-possible result for the current *active frame* with respect to errors in time τ and errors in space ς. This is based on the *active frame* and does not consider the *shown frame*. For *show*, both the errors of the current *shown frame* as well as those of the *active frame* are taken into account. The rationale behind this is that the *frame control parameters* are used to determine which combination of temporal and spatial error gives the lower overall error. For *resources*, as good settings highly depend on the use case, no clear, objective metric exists to balance the power or computation time consumed by the *progressive renderer* against the quality of the output. Here, we utilize the spatial error of the *active frame* to determine a maximum spatial error that is acceptable during interaction. In case the *progressive renderer* is capable of overachieving this value, resource usage can be reduced.

While errors could be related directly (e.g., [Woolley et al., 2003]), more flexible approaches are required to introduce a degree of freedom that allows the consideration of advanced aspects in dynamic frame control, like user preferences (Sec. 4.2.4). At the same time, parameters should be intuitively adjustable for a user and enable efficient automatic optimization. To achieve these goals, we introduce the *frame control parameters* consisting of a single parameter for each *restart*, *show*, and *resources*, denoted as $\rho, \vartheta, \chi \in [0,1]$, respectively. These parameters are defined by the user or by the automatic optimization process. The *frame control* can be represented by a function ϕ as follows, with multiple actions possible (note that *restart* automatically triggers *show* in our heuristics):

$$\phi = \begin{cases} \textit{restart}, & \text{if } \mu(\rho) \cdot \tau_t^a - \varsigma_t^a \geq 0 \\ \textit{show}, & \text{if } \mu(\vartheta) \cdot (\tau_t^a - \tau_t^s) + (\varsigma_t^a - \varsigma_t^s) \geq 0 \\ \textit{resources} & \text{if } \chi > \varsigma_t^a \vee \delta_t^\chi > \hat{\delta}^\chi \end{cases}, \tag{4.1}$$

where ς_t^a and ς_t^s denote the spatial error of the active and the shown frame at time t, respectively, τ_t^a and τ_t^s the temporal error of the active and the shown frame, and $\mu(\cdot)$ maps the parameters from their original range $[0,1]$ to $[0,\infty)$: $\mu(s) = \tan(s \cdot \pi/2)$, δ_t^χ gives the idle time of the resource at time t, and $\hat{\delta}^\chi$ denotes the respective threshold. The rationale behind the triggering of *restart*, *show*, and *resources* is discussed in detail next.

Restart. The temporal error τ_t^a of the *active frame* is weighted with ρ against the spatial error ς_t^a, thus defining the desired trade-off between these two quantities. Note that τ_t^a continuously increases with t, while ς_t^a is continuously decreasing.

Show. Similar to *restart*, we weight temporal against spatial error, but this time with respect to the differences between the *active frame* and the *shown frame*. Practically, the goal is to determine when the decrease in temporal error compensates for the higher spatial error. For all possible settings of ϑ, the active frame is shown as soon as $\varsigma_t^a > \varsigma_t^s$, because $(\tau_t^a - \tau_t^s)$ should always be nonnegative for reasonable implementations.

Resources. In our simple implementation, the resource usage is binary, i.e., either we are progressively refining a frame or idling. Here, the parameter χ defines a static target error value. If the spatial error estimation ς_t^a falls below χ during interaction, we pause rendering until the frame is restarted. We also limit the pause time δ_t^χ to a maximum value $\hat{\delta}^\chi$ (we used $\hat{\delta}^\chi = 1$ s), to ensure that the full (spatial) sampling rate is achieved eventually.

For the purpose of comparison, we also implement an alternative frame control $\overline{\phi}$ for the commonly used fixed image quality $\overline{\omega}$ settings and fixed render time per frame $\overline{\delta}$ settings.

$$\overline{\phi} = \begin{cases} \textit{restart}, & \text{if } (\omega_t^a \geq \overline{\omega} \vee \overline{\delta} \geq \delta_t^a) \wedge \tau^a > 0 \\ \textit{show}, & \text{if } (\omega_t^a \geq \overline{\omega} \vee \overline{\delta} \geq \delta_t^a) \wedge \tau^a = 0. \end{cases}, \tag{4.2}$$

where ω_t^a gives the current sampling rate, and δ_t^a denotes the time since the rendering of the *active frame* has been started. Practically, this means that for fixed settings, the

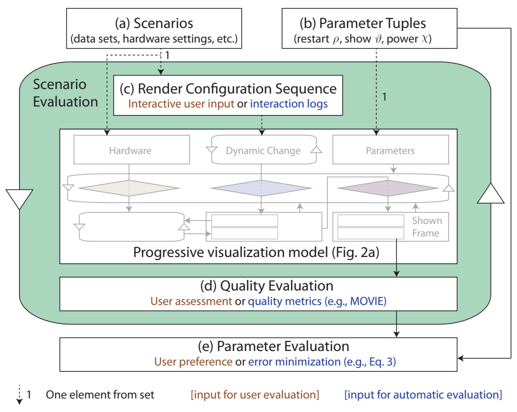

(a) Scenarios
(data sets, hardware settings, etc.)
(b) Parameter Tuples
(restart ρ, show ϑ, power χ)
1
Scenario Evaluation
(c) Render Configuration Sequence
Interactive user input or interaction logs
1
Hardware
Dynamic Change
Parameters
Shown Frame
Progressive visualization model (Fig. 2a)
(d) Quality Evaluation
User assessment or quality metrics (e.g., MOVIE)
(e) Parameter Evaluation
User preference or error minimization (e.g., Eq. 3)
1 One element from set
[input for user evaluation]
[input for automatic evaluation]

Figure 4.7: Our optimization scheme for *frame control parameters* with different variants for user-based (red) and automatic evaluation (blue).

progressive renderer proceeds until the target value is reached, and the *shown frame* is iteratively updated toward the full quality beyond these restrictions only when the *render configuration* is not modified (i.e., the temporal error τ^a is zero).

Parameter Optimization

The goal of the parameter optimization is to determine *frame control parameters* such that *frame control* delivers the best result according to a metric or user assessment. We distinguish between the following two basic variants: user evaluation and automatic evaluation by means of video quality metrics. In the former, a user simply assesses how well settings for ρ, ϑ, and χ are suited for the scenarios he works on (i.e., a data set, a task to accomplish, etc.) during an interactive session, and chooses the parameters accordingly. In the latter, automatic evaluation makes use of interaction logs recorded during previous interactive visualization sessions. Each interaction log defines a *render configuration sequence*, that along with different hardware settings define *scenarios*. Each *scenario* is rendered with a range of different *frame control parameters* and evaluated by means of a video quality metric. The output of the *quality evaluation* finally provides the basis for *parameter evaluation* to determine the best setting over all scenarios (Fig. 4.7). The respective components are implemented as follows.

(a) *Scenarios.* In our case, each scenario features a specific data set that is explored by means of a sequence of render configurations. We consider the exploration of different volumes in the following. A *scenario* may include computationally weaker systems, which might be simulated by virtually slowing down the test machine through different hardware settings. Optimally, *scenarios* should be representative in terms of the data sets, camera positioning, and so on, with respect to the target field of application.

(b) *Parameter Tuples.* Numerous parameter setting tuples (ρ, ϑ, χ) are considered for evaluation. Previous experience with the system or explicit pre-evaluation can help narrow down the considered range of parameters, thus pruning clearly undesirable settings early for a more efficient evaluation.

(c) *Render Configuration Sequence.* In the course of a user evaluation, the user may simply interact with the respective tool. For the automatic evaluation of our implementation, we use timelined series of changes to the camera setup and the transfer function. In our case, they come from recorded user interaction logs from previous sessions.

(d) *Quality Evaluation.* When a user is evaluating the interactive application, he or she may explicitly provide a score for the experience. Such a score could be determined automatically by measuring the user's performance for predefined tasks. In contrast, in our evaluation in this paper, we do not assess such explicit performance scores, but instead let the user provide his choice of parameters directly (Sec. 4.2.4). For automatic evaluation, algorithmic video metrics have been used for many years to assess the perceived quality of a video without the need for a user study. We use MOVIE [Seshadrinathan and Bovik, 2010] for the evaluation of interactive visualization, due to its high quality results and the availability of the source code. This gives us the possibility to seamlessly integrate it within our parameter optimization pipeline in a distributed environment. MOVIE is a full reference metric that compares a candidate video against a reference, and delivers a single error value as result. We generate these candidate videos by capturing the image from the display buffer 30 times per second. After completing a render configuration sequence for a certain parameter setting, we write the respective image files to disk, convert them to the required YUV video format using FFmpeg, and analyze it with the metric. The whole process runs automatically for given *parameter tuples* and *scenarios*. It also generates a script that can be used directly as input to the grid engine of our compute cluster. This script file defines a job array that invokes as many parallel instances of MOVIE as there are videos that need to be evaluated. As output, among others, MOVIE writes a text file containing a single overall scalar error value for each test, which we use for *parameter evaluation* in the following. Although these video metrics have been extensively researched and evaluated over the past 20 years, they can only be an approximation to the ground truth, the quality assessment by a human user. A detailed discussion of this aspect can be found at the end of Sec. 4.2.4.

(e) *Parameter Evaluation.* As indicated above, in the case of manual user evaluation, we simply let the user choose his favorite setting according to his interactive experience (Sec. 4.2.4). For the automatic evaluation, the error values are determined by the video quality metric for the m *scenarios* and the n *parameter tuples* to find the most favorable setting. This is accomplished by minimizing the least-squares relative error ϵ on the basis of the error measure $\gamma_{\cdot}^{a}(\cdot,\cdot,\cdot)$ from the video metric:

$$\epsilon = \min_{0<i\leq n}\left(\sum_{j=0}^{m-1}\left(\gamma_j^a(\rho_i,\vartheta_i,\chi_i)/\gamma_j^a(\rho_{b(m)},\vartheta_{b(m)},\chi_{b(m)}) - 1\right)^2\right), \tag{4.3}$$

with $b(m) \in \{0,\ldots,n\}$ denoting the index of the parameter setting giving the lowest error for scenario m. We use relative errors to account for the fact that the absolute error values given by a metric can vary significantly (e.g., with the length of an image sequence). In particular, this allows us to avoid any bias toward scenarios with higher error values overall, and to support the combined use of different error metrics.

4.2.3 Error-Based Progressive Volume Visualization

Despite significant improvements in algorithms and hardware, volume rendering is still computationally challenging for the data sets produced by up-to-date scanners and simulations. We implement a progressive, multiresolution GPU volume raycaster for illustrating and evaluating our error-based frame control. In particular, an efficient integrated implementation for the spatio-temporal error estimates is provided.

Progressive Volume Visualization

The main goals of our progressive volume rendering scheme are performance, flexibility, and simplicity, both with respect to implementation and integration with existing renderers. By flexibility, we mean both the capability to interrupt the renderer at virtually any time, and the absence of any kind of preprocessing or assumptions of coherence across frames. We utilize a multiresolution volume raycaster with optimized sample distribution in image space [Balzer et al., 2009].

Sampling. The renderer uses multiple resolution levels in image space, each of which is subdivided into tiles. The goal is to achieve both flexible interruptability of the *progressive renderer* as well as efficient usage of the hardware. Thus, the granularity (i.e., the tile size) needs to be chosen according to device characteristics. For our CUDA implementation, we determined by experiment that 16K samples per tile are sufficient to efficiently utilize the 512 stream processors of a NVIDIA GTX580 without introducing significant overhead. For the chosen aspect ratio, we generate sample points by means of capacity-constrained point distributions with periodic boundary conditions [Balzer et al., 2009] (Fig. 4.8). Next, following a quad-tree approach, we subdivide the image space into tiles of increasing resolution levels until there is more than one sample point per pixel. For each resolution level, starting from our original set of samples, we add the required periodic copies to the list of samples. In our experiments, we use WXGA+ (1440×900) as the screen resolution for our renderer, because this constitutes a good

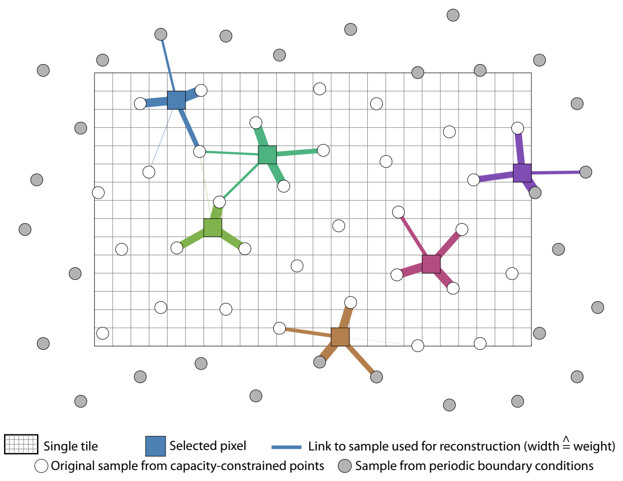
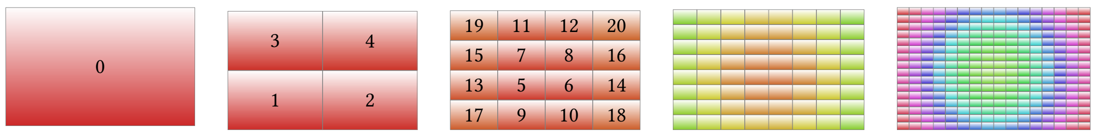

Single tile
Selected pixel
Link to sample used for reconstruction (width $\hat{=}$ weight)
Original sample from capacity-constrained points
Sample from periodic boundary conditions

Figure 4.8: Reconstruction of pixels from samples and their weight according to a Gaussian filter kernel (Eq. 4.4), illustrated for colored pixels. Periodic boundary conditions of the sample distribution are explicitly considered. In our implementation, sample weights are precomputed for each tile resolution and stored in sampling table S.



$l0$: 1440×900 (1×1), s : 0.0–0.3% $l1$: 720×450 (2×2), s : 0.3–1.5% $l2$: 360×225 (4×4), s : 1.5–6.2% $l3$: 180×113 (8×8), s : 6.2–24.9% $l4$: 90×57 (16×16), s : 24.9–100.0%

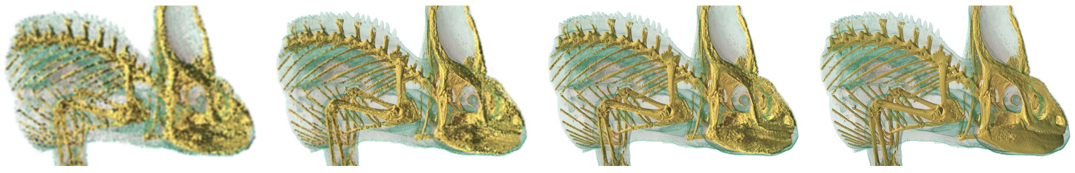
Figure 4.9: Tiles at different resolution levels l generated for a 1440×900 screen. Their sampling rate s is given and order of processing is depicted by numbers and colors.

trade-off between screen space and the induced cost for automatic video analysis. In total, this results in 5 resolution levels and 341 tiles in total (Fig. 4.9). In this paper, we measure the *sampling rate* by means of the ratio of completed to total tiles. Akin to the image-space sampling, the sampling distance along rays in object space is doubled, starting from the highest, most detailed resolution level. To render a frame, the tiles are processed from the lowest to highest resolution level. Within each resolution level, they are ordered from the screen center to the outside. Error estimation and *frame control* are carried out between the computation of tiles. The achieved high, sub-frame granularity is essential to be able to react quickly to sudden changes. Our volume raycaster uses a simple local lighting model, making use of gradients that are determined on-the-fly using central differences. The raycaster further makes use of early ray termination, and lighting computations are only executed in non-empty space.

Image Reconstruction and Blending. Pixel color values are reconstructed from the sample points using a Gaussian filter kernel

$$f(d) = \mathrm{e}^{-\alpha d^2} - \mathrm{e}^{-\alpha r^2}, \tag{4.4}$$

with d denoting the distance in image space, α controlling the falloff, and the radius r



(a) s = 0.3%, 103.3 fps (b) s = 3%, 6.4 fps (c) s = 10%, 1.5 fps (d) s = 100%, 0.2 fps

Figure 4.10: Different sampling rate settings s and achieved frame rates with our renderer for the Chameleon data set.

Algorithm 4 Integrated image reconstruction and error estimation. Spatial error estimation implementation is based on incremental variance computation and depicted in blue. Temporal error is simply computed from the difference to the previous color as shown in green.

```
1: procedure RECONSTRUCTION($x, y, l$)    ▷ pixel coordinates $(x, y)$ in tile with resolution level $l$
2:     $w_\Sigma, \boldsymbol{c}_\Sigma \leftarrow 0$    ▷ initialize sum of weights and sum of colors
3:     $\boldsymbol{m}, \boldsymbol{m}_2 \leftarrow 0$    ▷ initialize mean and squared differences from mean
4:     for all $(i, w) \in S(x, y, l)$ do    ▷ fetch sample index $i$ and weight $w$ from sampling table $S$
5:         $\boldsymbol{c_t} \leftarrow \boldsymbol{C}(i)$    ▷ lookup sample color from output of renderer
6:         $\boldsymbol{c}_\Sigma \leftarrow \boldsymbol{c}_\Sigma + w \cdot \boldsymbol{c}_t$    ▷ update weighted color sum
7:         $w'_\Sigma \leftarrow w_\Sigma$    ▷ back up sum of weights from previous iteration
8:         $w_\Sigma \leftarrow w_\Sigma + w$    ▷ update sum of weights
9:         $\boldsymbol{d} \leftarrow \boldsymbol{c}_t - \boldsymbol{m}$    ▷ deviation from current mean
10:        $\boldsymbol{d}_w \leftarrow \boldsymbol{d} \cdot \frac{w}{w_\Sigma}$    ▷ relative weighted deviation from current mean
11:        $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{d}_w$    ▷ update mean
12:        $\boldsymbol{m}_2 \leftarrow \boldsymbol{m}_2 + w'_\Sigma \cdot \boldsymbol{d} \cdot \boldsymbol{d}_w^T$    ▷ update sum of squared differences from current mean
13:    $\boldsymbol{c} \leftarrow \boldsymbol{c}_\Sigma / w_\Sigma$    ▷ output color for pixel $(x, y)$
14:    $\boldsymbol{\sigma}^2 \leftarrow (\boldsymbol{m}_2 / w_\Sigma) \cdot |S(x, y)| / (|S(x, y)| - 1)$    ▷ compute variance
15:    $\varsigma \leftarrow |\boldsymbol{\sigma}^2|$    ▷ spatial error $\varsigma$ is length of variance vector
16:    $\tau' \leftarrow |\boldsymbol{c} - \boldsymbol{c}_{\text{prev}}|$    ▷ $\tau'$ is difference between $\boldsymbol{c}$ and previous color $\boldsymbol{c}_{\text{prev}}$
17:    return $(\boldsymbol{c}, \varsigma)$ or $(\boldsymbol{c}, \tau')$    ▷ return resulting pixel and respective error
```

ensuring that the filter goes to zero at the boundary of its support [Pharr and Humphreys, 2010]. On this basis, in a precomputation step, we generate the sampling table S, that lists for each pixel (x, y) of a tile of sampling resolution l which samples are required for its reconstruction along with their respective weights (Fig. 4.8). S is then used at runtime in a CUDA kernel to compute the color value of each pixel of a tile (Alg. 4). Finally, for display, we use OpenGL to blend the boundary between tiles of different levels to reduce visual disturbances occurring from visible edges. This is implemented by employing precomputed opacity maps which are generated with respect to the distance of the pixels to the border between tiles of different resolution levels. Results for different sampling rates are shown in Fig. 4.10.

Error Estimation

As the basis for *frame control*, fast yet expressive metrics are required for the spatial and temporal error estimators to provide updated values after the rendering of each tile.

Spatial Error Estimation. For each pixel, the estimation of spatial error ς can be efficiently achieved along with image reconstruction such that it only induces marginal computational and no memory lookup overhead (Alg. 4, green). In our implementation, the spatial error is determined by the magnitude of the weighted RGB variance vector over the considered RGB sample color values $\boldsymbol{C}(\cdot)$ (Alg. 4, line 15). The underlying incremental algorithm to compute the weighted variance is due to West [West, 1979]. Pixel error values are then summed up using a hierarchical reduction scheme on the GPU that makes use of shared memory. Finally, the average pixel error value serves as spatial error estimation. Note that after the completion of each tile only a partial update to the previous error value is required, as only the image region covered by the respective tile needs to be considered. In our implementation, for each pixel covered by the tile, the difference between the previous and the current error value is added to the total value. When *restart* is issued, the spatial error of the active frame ς^a is simply carried over to the shown frame error value ς^s.

Temporal Error Estimation. For temporal error estimation, we first generate a quick approximation of the frame with the current *render configuration*, i.e., the latest camera position, transfer function, and data time step. In detail, we omit lighting, and sample the volume only sparsely during raycasting by using the lowest resolution tile and significantly increase the step size along rays (we use a factor of 50 with respect to the highest level). The respective temporal error is computed by means of per-pixel color differences of the current to the previous approximated rendering (Alg. 4, line 16). Next, the temporal error values τ' of all pixels are summed up by employing the hierarchical scheme that is also used for spatial error estimation. This value is added to the total temporal error value τ^a of the active frame and the shown frame τ^s, with $\tau^a = 0$ after each *restart*. As with the spatial error value, the temporal error value of the active frame τ^a is carried over to the error of the shown frame τ^s in case of a *show* or *restart*. Note that in contrast to the spatial error that is computed during the reconstruction of every tile for display, the temporal error is only evaluated in case of changes to the *render configuration* since the last assessment.

4.2.4 Application and Evaluation

For running our measurements, we use a NVIDIA GTX580 and an Intel Core i7 with 3.4 GHz. We employ four data sets from both CT scans and simulations in our evaluation (references to renderings and data set resolution are given in brackets): the Chameleon

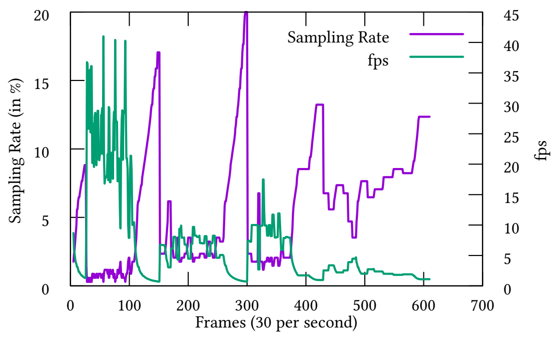


Figure 4.11: Sampling rate and frame rate for $\rho = 0.6$ for a camera path with the Chameleon data set.

data set (Fig. 4.10, 1024×1024×1080), the Jet data set (Figs. 4.12(a) and (b), 720×320×320), the Flower data set (Fig. 4.5, 1024 × 1024 × 1024, courtesy of the Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab at University of Zürich), and the time-dependent Vortex data set (Figs. 4.12(c)–(e), 529 × 529 × 529, with 60 time steps). All compute-intense steps required for interactive rendering are executed in parallel on the GPU using CUDA, particularly including the volume raycaster and the reconstruction along with the error estimation. For our 1440×900 screen resolution, the image reconstruction takes 5 ms for the lowest level, 1.5 ms for the second-lowest level, and below half a millisecond for all subsequent levels. This decrease in reconstruction time exhibits a roughly linear dependence on the decreasing number of pixels covered by a tile (Fig. 4.9). To reduce this cost for higher levels, a hybrid reconstruction and upscaling approach could be investigated for future work.

As discussed in Sec. 4.2.3, computing the spatial error through the variance of the samples can be done without significant overhead, basically requiring only a couple of additional floating point operations. Determining the temporal error basically consists of two steps, namely the approximate rendering and the integrated reconstruction and difference computation to the previous approximation. In our measurements, the approximate sampling was in the range of 1% to maximally 10% of the full render time of a tile, while the subsequent reconstruction and error computation always took around 0.5 ms (e.g., for reference, the sampling of tile 22 in Fig. 4.10 took 24 ms). This means that the computational overhead for assessing the temporal error is fairly low, particularly when considering that this is only done in case of changes to the *render configuration*.

Practical Example of Error-Based Frame Control

We showcase the characteristics of error-based frame control at the example of an interactive exploration session with the Chameleon data set (Fig. 4.11). In the first

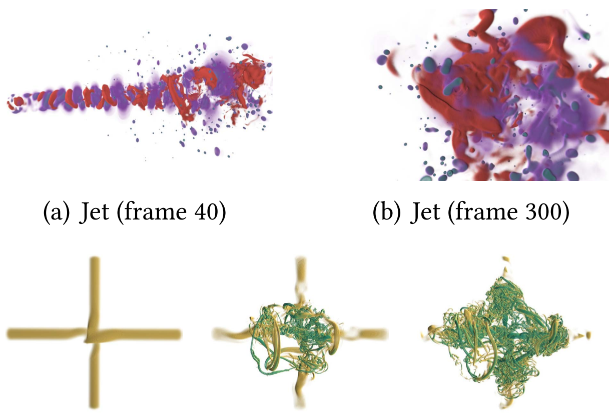


Figure 4.12: Key frames of the videos from the simulation data sets that were used for automatic evaluation. In the respective camera path, the Jet data set was quickly rotated from the side view (a) to the tip of the pressure advancement (b), which was then investigated in detail. The Vortex series (c)–(e) shows the temporal development of the vortex cascade, visualized with the λ_2 criterion [Jeong and Hussain, 1995].

(a) Jet (frame 40) (b) Jet (frame 300)

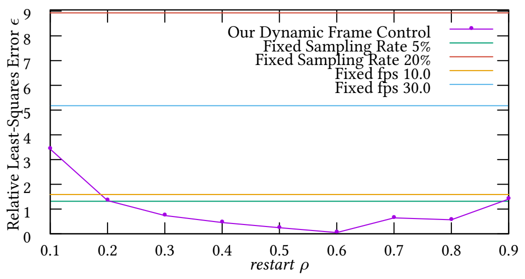
(c) Vortex (t = 10) (d) Vortex (t =70) (e) Vortex (t =100)

100 frames the data set is quickly rotated. This induces a high temporal error and accordingly frame rates around 25 fps for more fluent navigation. From frame 110 to 140, no changes occur for almost a second, which leads to a continuous refinement of the current frame, as can also be seen from the increasing sampling rate. Then, there is a sequence of slower camera movement (frames 150–250), followed by another phase of no change (frames 250–300). Then, after slower changes to the camera position (frames 300–380), a certain feature is investigated in detail. This requires a high sampling rate to enable the user to assess fine details. We refer the readers to the accompanying video for further details.

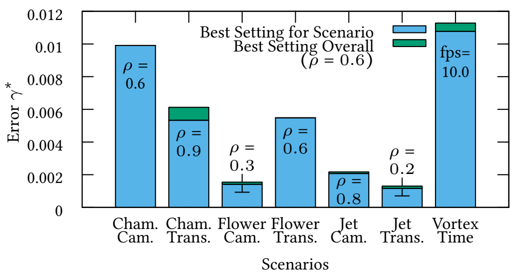
Automatic Evaluation Using a Video Quality Metric

By means of automatic evaluation, we aim to optimize the parameters ρ for *restart* and ϑ for *show*, and study the impact of the power reduction parameter χ. We considered seven scenarios in total: the Chameleon, Jet, and Flower data sets, each with one sequence adjusting the camera path and one adjusting the transfer function, and the Vortex data set receiving a new time step every 100 ms. As input for MOVIE, we compute a video for each parameter setting along with a reference video that batch renders a full-quality image for every frame. The videos are generated from transformations that were recorded from user interaction. To be representative for a variety of user actions, they contain both slow and fast-paced changes. These videos are only between six and twenty seconds long to keep computation time both for generation and subsequent evaluation within reasonable limits.

In our experiments, the evaluation of a video with a resolution of 1440×900 and 30 frames per second using MOVIE took twelve to sixteen hours on an Intel Xeon processor running at 2.4 GHz. By using a small cluster, we could process up to 64 videos in parallel. However, considering variations of multiple parameter values in one measurement series would still take a prohibitively long time. Thus, we use a multi-stage process to significantly cut down the number of evaluations. First, we optimize the restart parameter ρ, which has been shown by our previous experiments to be of predominant impact in comparison to the show parameter ϑ. Then, for the resulting optimal setting for ρ, we evaluate different parameter settings for ϑ. On this basis, we finally examine the impact of the resource parameter χ.



(a) Least-squares relative errors over all considered scenarios for different values for ρ.



(b) Error of the most optimal restart parameter setting for each scenario, plotted against the most optimal global setting of ρ = 0.6 from (a).

Figure 4.13: Automatic optimization of the restart parameter ρ using the MOVIE video metric and a variety of different scenarios.

For the restart parameter ρ, we consider values from 0.1 to 0.9 in steps of 0.1. We further include static frame rates with 10 fps and 30 fps as well as static sampling rates of 5% and 20% in our evaluation. Fig. 4.13(a) shows the resulting least-squares relative error over all scenarios (as discussed in Sec. 4.2.2). According to this, error-based frame control with ρ = 0.6 achieves the smallest error value overall. The relatively high static sampling rate (20%) that delivers high-quality renderings but frame rates below 1 fps delivers the worst result. Here, low quality settings (5%) favoring higher frame rates improve the results significantly. For static frame rates, 10 fps delivers significantly lower errors than 30 fps, meaning that it provides a better trade-off between sampling rate and responsiveness for the considered cases.

As indicated by the low slope of error values for ρ across a wide range of values in Fig. 4.13(a), a variety of different settings are potential candidates for delivering the best result in one specific scenario (Fig. 4.13(b)). However, it can be seen that the determined optimal setting ρ = 0.6 is only marginally worse in any scenario, with respect to the best result for each scenario individually. Our error-based control further yields the lowest error for each scenario individually, except for the Vortex series. Naturally, its discrete refreshes 10 times a second yields good results with the 10 fps setting. In total, it can be seen that error-based frame control with $\rho = 0.6$ can be used successfully across all considered use cases without requiring further adjustment.

For $\rho = 0.6$, the impact of the show parameter is investigated with the following settings: $\vartheta \in \{0, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.5\}$. Our results indicate that the lowest overall error is achieved with $\vartheta = 0$. This means that *show* is only issued when a frame is restarted or when the spatial error of the active frame is smaller than that of the shown frame. While a more early show operation can reduce the visible temporal error and allow the frame to still refine further, we think this result is due to the induced strong variation of displayed image quality, leading to significant flickering.

Finally, Fig. 4.14 shows the impact of the power parameter χ on the resource usage and the error determined by MOVIE. In our implementation, χ defines an error threshold with respect to our spatial error estimation. Most prominently, it can be seen that a quite significant reduction of resource usage can be achieved with only little increase in error. This is due to the fact that an increasing sample count has decreasing impact on the perceived image quality, i.e., the resource usage can be lowered for sampling rates beyond a certain threshold with merely sub-linear quality impact. The negative effect of trading resource usage against sampling rate is further significantly dampened by the adaptivity of our approach, i.e., resource utilization is only reduced in adequate situations as defined by the user. More elaborate schemes and a more detailed evaluation of its effects, e.g., on explicit power consumption, remain for future work.

Expert Review

Scope and Structure. The primary goal of our expert review was to evaluate the usefulness of our approach for volume rendering against fixed settings for frame rate and quality (denoted as *modes* below). Further objectives of the study were to identify preferred parameter settings, and to assess similarities and differences to the automatic evaluation from Sec. 4.2.4. Five visualization researchers evaluated our implementation by interactively exploring the Chameleon and the Jet data set. They are primarily concerned with the development of new visualization techniques, but also use their own and other interactive tools for analysis on a regular basis. Each participant spent 45 minutes to 1 hour interacting with our implementation.

The procedure was loosely structured into three phases. First, we asked the participants to make themselves familiar with the volume rendering tool and the different modes of the program. Second, we gave them a number of exploration tasks to accomplish and asked them to evaluate the usefulness of our tool along the way. In this phase, the parameter values of the different modes were fixed as follows: the static frame rate was set to 30 fps, while the sampling rate was at 20%, and the restart parameter ρ was set to 0.6. For simplicity and time constraints, the show and power parameters were set to 0 and ignored throughout the study. A task consisted either of matching a certain camera view or transfer function to a precomputed visualization result. Twenty randomly distributed and uniformly colored marker crosses were added to the volume to support the matching. Tasks were structured into groups of three (one for each mode), and the order in which the modes had to be used was shuffled quasi-randomly. Finally, there was a free exploration phase in which we asked the participants to explore different parameter settings, and compare the different modes freely to determine the one they like best. After phase two, the participants were handed a small questionnaire. They were asked to complete the part about their experience so far right away, and fill in the remaining parts during the final phase three. The questionnaire contained both multiple choice fields regarding the preferred modes as well as text fields for providing comments regarding different aspects.

User Comments. First, the participants were asked to assess the usefulness of the different modes with respect to the camera configuration and transfer function matching tasks. Overall, all participants found the fast response time of the static frame rate very helpful, particularly for quick camera rotations. However, Participants 2 and 4 noted that this also loses a significant amount of detail in the data set. With respect to the provided static sampling rate, all participants complained that while providing detailed renderings, its significant delays impede precise navigation. Our error-based frame control was rated as being a good compromise between responsiveness and detail (Participant 4), which adaptively allows for a high enough frame rate but also provides detail (Participants 0 and 3). Participants 0 and 1 additionally noted that they found it particularly helpful for matching transfer functions. However, also for transfer function matching, Participant 2 perceived the delays for small changes to be a little too long with our error-based control. In summary, error-based frame control was chosen over static frame rate or quality as the overall preferred method for the task phase. However, there were some remarks that the tasks were not emphasizing the properties and characteristics of the different modes strongly enough. For instance, Participant 0 noted that the disadvantage of low quality for static frame rate did not affect the tasks that much because the provided orientation markers were still visible. For future work, we would like to more closely emulate real world tasks in a more extensive study.

Next, in the exploration phase, the participants were asked to navigate freely, i.e., to explore the data set on their own and determine their personal preferences this way. We summarize their comments in the following. It was noted that the required sampling rate in general strongly depends on the data set and that it is thus hard to set for general purposes (Participants 0, 3, and 4). According to Participant 3, it also always bears the potential of sudden drops in frame rate should the rendering cost change quickly. Thus, the parameter setting highly depends on what the goal of the exploration is (Participant 0). As a result, Participant 2 found the static sampling rate unpleasant to use, particularly for longer sessions. Most of the time during interactive exploration, the

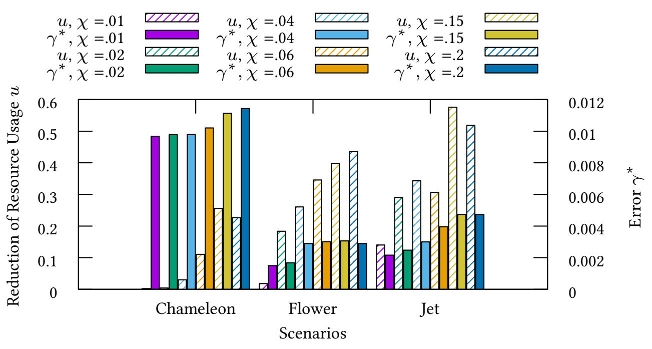


Figure 4.14: Impact of power parameter settings χ on resource usage u and error γ^* for the camera path scenarios with $\rho = 0.6$ and $\vartheta = 0$.

effects of a static sampling rate are either choppy movement or low rendering quality that misses important features (Participant 1). While static frame rate delivers better results more independently from the underlying data set (Participant 4), the strong loss of quality that occurs even for only slight changes was found unpleasant, particularly for cases in which details are of importance (Participant 3). Error-based frame control was generally found to be “a good compromise between static frame rate and static sampling rate” (Participant 3). Participant 1 stated that error-based frame control is most appropriate for adjusting transfer functions and the camera position, both when it comes to slow and fast movement. Participant 0 particularly preferred error-based frame control for detailed adjustments of the transfer function because more detailed and thus more helpful renderings were available during interaction.

Ratings and Parameter Settings. Like the participant’s comments discussed above, the order of suitability selected by the participants in the questionnaire clearly reflect a preference toward error-based frame control (Table 4.1). They favored it four to one for detailed investigation of the data set as well as for overall usage, with the other one being static frame rate. While a static frame rate delivering high quality would be great for detailed investigation once an interesting spot has been reached, getting there is cumbersome due to the involved sluggishness. For just getting a quick, rough overview of the data sets, comments and selected preferences suggest that both a high static frame rate and error-based frame control are well suited for this use case. This could be expected, as for faster movement, error-based frame control leads to high fps as well. However, the parameter settings vary in a certain range with specific settings depending on general user preferences toward high quality or responsiveness. For the static sampling rate, parameter settings in the low range between 2% and 7.5% were chosen, despite the significant visual disturbances associated with that (e.g., Fig. 4.10(b)). For static frame rate settings, preferred settings vary between 10 fps or 30 fps, with a slight preference overall toward the lower end for higher visual quality. Parameter settings for the restart parameter ρ range between 0.3 and 0.7, with a preference toward the higher end, i.e., toward higher frame rates. Furthermore, preferred settings may vary with the data set, with lower values for the both more expensive to render and complex structured Chameleon in comparison to the Jet data set. Relating to this, Participant 1 stated that the Jet data set contains less detail, and thus different settings seem adequate as compared to the Chameleon data set.

Comparison to Automatic Evaluation. In essence, the results from the user study confirm the basic trend from the automatic evaluation (Sec. 4.2.4 and Fig. 4.13). For the static sampling rate, relatively low quality settings are preferred by the users, as they allow for fluid interaction for a wide range of camera and transfer function configurations. In the automatic evaluation, this is reflected by the much lower relative least-squares error ϵ for the lower sampling rate setting in Fig. 4.13(a). For static frame rate, user preferences range approximately between 10 fps or 30 fps, i.e., varying in its trend toward image quality or responsiveness. Automatic evaluation exhibits basically the same trend, yet a lot more distinctively (Fig. 4.13(a)). For error-based frame control, the restart parameter $\rho = 0.6$ was determined as the best setting by the automatic evaluation, with lower values exhibiting only slightly, yet continuously worse results. Similar settings were also popular with the participants.

Regarding the chosen preferences with respect to the mode, both the user study and the automatic evaluation clearly favored our error-based frame control. However, while user preferences lean toward fixed frame rate when compared to fixed sampling rate, they perform about equally well with their optimal parameter setting according to automatic evaluation (Fig. 4.13(a)). Note that for the results in Sec. 4.2.4, four different data sets were used, while only two were part of the expert study here. More definite qualitative and quantitative statements would require an automatic evaluation with a wider range of data sets and performed interactions, a more extensive expert study with more participants, and possibly additional video metrics. In particular, while video metrics have the important advantage of allowing for automatic evaluation and optimization, they are optimized for determining the quality of videos, which might differ from the user experience of an interactive visualization tool. Here, we believe that more research is required to better quantify these differences.

4.2.5 Directions for Future Work

We plan to significantly expand the aspect of resource utilization and study the possibilities of intelligently decreasing power consumption in more detail. We would further like to extend our evaluation by means of other video metrics (like DRIVQM [Aydin et al., 2010]) and a more extensive user study. This user study should incorporate a more diverse group of users, particularly featuring application domain scientists, and consider the experiences of users utilizing it in their everyday work. In this context, taking the characteristics of human interaction into account more comprehensively

Table 4.1: Ratings given by the visualization experts in the following order: *fixed frame rate*, *fixed sampling rate*, *error-based frame control. O* stands for the order of preference, 1 being the most preferable to 3 being the least preferable. *P* stands for the selection of the best-suited parameter value for each of these. *Tasks* denotes the rating of the task phase of our evaluation. *Overv.* denotes the suitability for getting a quick overview of the data set, while *Det.* stands for the suitability for the detailed analysis of a certain feature. *Tot.* gives the overall rating.

Id	*O* Tasks	*P* Cham.	*P* Jet	*O* Overv.	*O* Det.	*O* Tot.
0	2 3 **1**	5 5 0.2	5 6 0.07	**1** 3 2	2 3 **1**	2 3 **1**
1	**1** 3 2	28 2 0.3	25 2 0.7	**1** 3 2	2 3 **1**	2 3 **1**
2	**1** 3 2	9 2 0.7	9 4 0.6	2 3 **1**	**1** 3 2	**1** 3 2
3	2 3 **1**	10 3 0.3	25 5 0.6	2 3 **1**	2 3 **1**	2 3 **1**
4	2 3 **1**	10 2 0.15	10 7.5 0.2	**1** 3 2	2 3 **1**	2 3 **1**

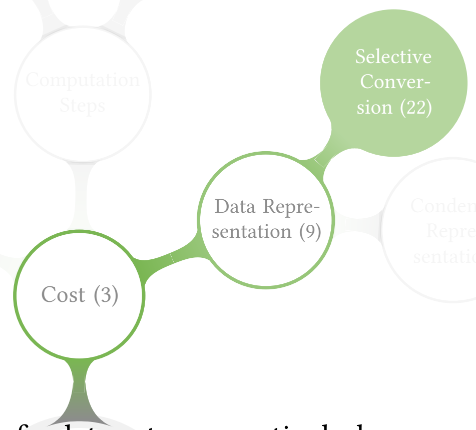
could also be a promising direction. We would further like to implement and evaluate our error-based control scheme for an extended raycaster (featuring other data types and out-of-core rendering), as well as other visualization techniques beyond volume rendering. In addition, limiting ourselves to one technique for generating frames, like to raycasting in this paper, cannot avoid significant (temporal and/or spatial) artifacts in cases in which the gap is too large between the cost of this technique and the power of the available compute resources. The extension to hybrid approaches that switch to other, computationally cheaper techniques (like warping [Qu et al., 2000; Shen and Johnson, 1994]) when required could help handling these cases in a more suitable way for the user. Finally, considering additional information like the rate of incoming data (e.g., with simulations running in parallel), or outgoing images over the network in remote rendering could allow for more efficient frame control in such scenarios, too.

Chapter 5

View-Dependent Representations

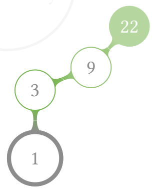
Scientists must be able to analyze the full extent of the simulation output at high resolution. However, transferring data at large scale for post-processing visualization and analysis is not feasable in many situations. Even if the data files can be moved, desktop data analysis and visualization tools struggle to handle such large-scale data. To alleviate this issue, view-dependent image-based rendering techniques have become increasingly popular in recent years as they combine the high quality of images with the explorability of interactive techniques. These approaches make use of an intermediate representation, often referred to as explorable image, that allows for deferred interaction. Explorable images have many different applications in a variety of areas. They can be used to generate frame previews when rendering is expensive, or provide a compact in-situ representation for results from large-scale, high-fidelity simulations.

5.1 Volumetric Depth Images



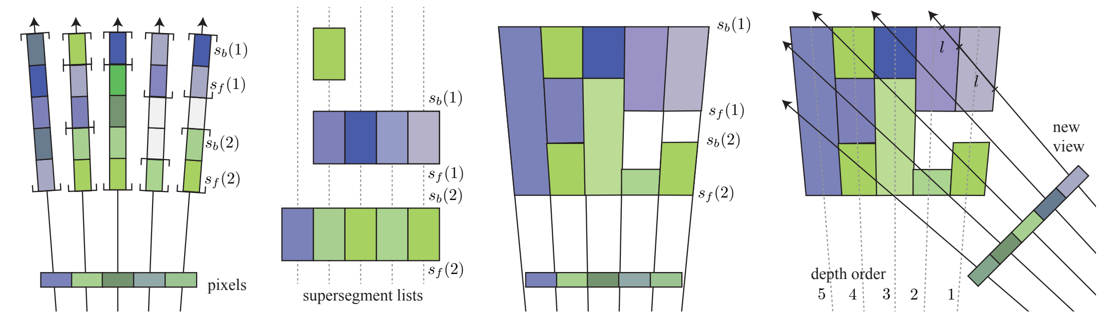
The interactive analysis of a data set may particularly concentrate on certain features and areas. Here, we aim at preserving full quality for a certain parameter configuration of interest, while still maintaining some flexibility to allow for user exploration by means of arbitrary changes to the camera. For this, Volumetric Depth Images (VDIs) are introduced, a condensed representation for volume data. Instead of only saving one color value for each view ray as in standard images, VDIs store a set of so-called supersegments, each consisting of a depth range as well as composited color and opacity. This compact representation is independent from the structure of the original data and can quickly be generated by a slight modification of raycasters. VDIs can be rendered efficiently at high quality with arbitrary camera configurations.

Strategy 22 ***Selective Conversion***
Convert the data set to a different, more compact representation. For this, additional knowledge may be exploited concerning regions or characteristics of interest.



5.1.1 Generation from Raycasting

As a generalization of Layered Depth Images (LDIs), VDIs allows to capture not just a surface, but a volume from a certain camera configuration in a fast and efficient way for subsequent rendering (Fig. 5.1). It can easily be generated during volumetric raycasting by partitioning the samples along rays (Fig. 5.1(a)) according to their similarity. Empty regions are skipped completely. These partitions can then be stored as lists of so-called supersegments containing the bounding depth pair (s_f, s_b) and partial color accumulation values (Fig. 5.1(b)). Each pixel in the image plane forms a square pyramid with the camera position at its apex. Accordingly, supersegments are rendered as



(a) Supersegments (b) VDI storage (c) Frustums (d) Rendering

Figure 5.1: VDI generation and rendering. (a) Supersegments are generated during raycasting (b) and stored as tuples that are organized in a 2D array of lists (Sec. 5.1.1). (c) For rendering, frustums constitute the 3D representations of supersegments (d) that are accumulated to generate a new image (Sec. 5.1.2).

frustums of these pyramids (Fig. 5.1(c)). Frustums generated by the same ray are conceptually grouped into a frustum list. Color and opacity are modeled constant within a frustum, and determined during VDI-generation in a raycaster. During rendering, frustum lists are depth-ordered and composited (Fig. 5.1(d)). For this, the length l a ray passes through each frustum is determined to correctly adjust a frustum's opacity contribution.

A modified front-to-back raycasting procedure for creating supersegments is depicted in Alg. 5. The main difference to standard raycasting is that color and opacity are not composited over the whole ray but only within supersegments. The segmentation criterion Γ (Line 4, Eq. 5.1) returns true if the old supersegment needs to be terminated (Lines 8–10) and a new one has to be started (Lines 12–13). The current supersegment is also finalized when entering a transparent segment (Line 5), and a new supersegment is started when leaving such a transparent segment (Line 11). Finally, a supersegment is closed before terminating the ray (Line 6). When a new supersegment is started, the color and opacity values are reset (Line 13). As c represents a premultiplied color value after compositing, it is divided by the opacity of the supersegment to make it non-premultiplied (Line 10) as a preparation for the subsequent over-operator type blending in the rendering stage (Sec. 5.1.2).

Naturally, mostly homogeneous supersegments are desirable to achieve good results, particularly for large view parameter changes in VDI rendering. There is a trade-off between quality and both storage requirements and rendering speed: the more supersegments, the higher the quality but also the higher the memory usage and render cost. Our criterion Γ is based on premultiplied color values and the correction of opacity

Algorithm 5 Generation of supersegments with color $C^{s_f}_{s_b}$ and opacity $\alpha^{s_f}_{s_b}$ using raycasting. Modified version of standard raycasting routine (Alg. 3).

```
1:  function SupersegmentGeneration
2:      $c \leftarrow (0,0,0), T \leftarrow 1, p \leftarrow -1$  ▷ initialize color $c$, opacity $T$, and supersegment count $p$
3:      for $i = 1 \rightarrow N$ do  ▷ step along ray (iterating over segments)
4:          $g_{\mathbb{B}} \leftarrow \Gamma(\gamma, C, \alpha, c(i), \alpha(i))$  ▷ new supersegment? (Eq. 5.1)
5:          $e_{\mathbb{B}} \leftarrow (\alpha(i) = 0 \wedge \alpha(i-1) \neq 0)$  ▷ transition into an empty region?
6:          $f_{\mathbb{B}} \leftarrow i = N$  ▷ last segment?
7:          if $p \neq -1 \wedge (g_{\mathbb{B}} \vee e_{\mathbb{B}} \vee f_{\mathbb{B}})$ then  ▷ close old supersegment
8:              $s_b(p) \leftarrow \min(i, N)$  ▷ write back depth
9:              $\alpha^{s_f(p)}_{s_b(p)} \leftarrow 1 - T$  ▷ write opacity
10:             $C^{s_f(p)}_{s_b(p)} \leftarrow c/\alpha^{s_f(p)}_{s_b(p)}$  ▷ factor out opacity and write color
11:         if $(g_{\mathbb{B}} \vee \alpha(i-1) = 0) \wedge \neg f_{\mathbb{B}} \wedge \alpha(i) > 0$ then  ▷ start new supersegment
12:             $p \leftarrow p + 1, s_f(p) \leftarrow i$  ▷ increment count and write front depth
13:             $c \leftarrow (0,0,0), T \leftarrow 1$  ▷ reset color and opacity
14:         $c \leftarrow T \cdot \alpha(i)c(i), T \leftarrow T \cdot (1 - \alpha(i))$  ▷ standard front-to-back compositing
```

with respect to integration lengths:

$$\Gamma : \gamma > |(c^{i-1}_{s_f}, \alpha^{i-1}_{s_f}) - (\hat{\alpha}(s_f, i)c(i), \hat{\alpha}(s_f, i))| \tag{5.1}$$

with γ being the sensitivity parameter, s_f is the starting index of the supersegment, $(c^{i-1}_{s_f}, \alpha^{i-1}_{s_f})$ represent color and opacity of the supersegment respectively, and i depicts the current segment. The opacity is length-corrected

$$\hat{\alpha}(s_f, i) = \omega(\alpha(i), \mu(s_f, i-1)),$$

with μ being the length of the supersegment (i.e., the sum of lengths of the contained segments):

$$\mu(f, b) = \sum_{j=f}^{b} \delta(j). \tag{5.2}$$

In total, this means that a raycaster segment is merged into the supersegment if the adjusted color and opacity difference is below the user-provided sensitivity parameter γ. Otherwise, a new supersegment is started. This greedy criterion is fast and simple to compute, can easily be integrated with existing raycasting codes, and proved to deliver good results during our experiments. Many other (more complex) schemes are possible and could easily be used with our flexible approach to meet different demands of a specific application at hand.

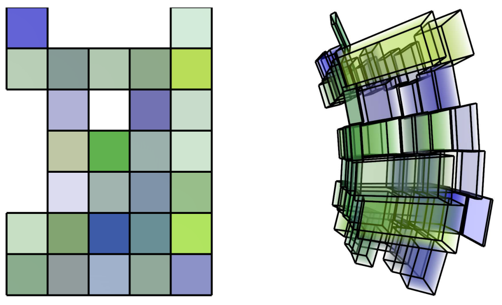

(a) VDI original view
(b) Moved camera

Figure 5.2: The engine data set (see Sec. 5.1.3 for details) with very low resolution settings demonstrates the geometry of frustums, and the results of depth sorting and opacity determination.

5.1.2 Rendering

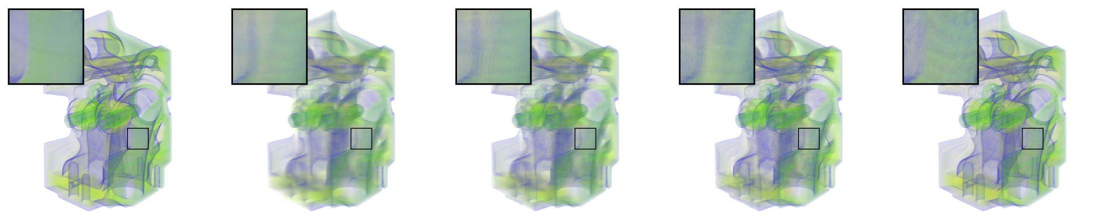
Each supersegment is represented by a frustum for rendering. For every supersegment list of a VDI, four rays are cast from the initial camera position through the corners of the respective pixel. For this purpose, the original modelview and projection matrices are used. The "bottom left" ray defines a perpendicular front and a back plane by using its direction vector as well as the two depth values marking the beginning and end of the respective supersegment. The front and back planes are then intersected by the other three rays, thus defining a quadrilateral frustum belonging to each supersegment.

For rendering our frustums, we employ alpha compositing with the over operator-according to Eq. 5.3: a over b computes the resulting color C as follows Max [1995]:

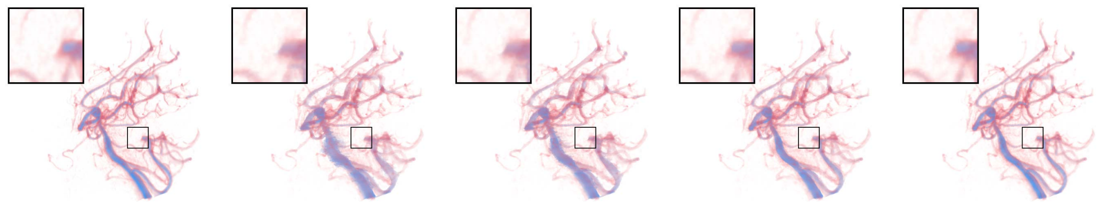
$$C = C_a\alpha_a + C_b(1 - \alpha_b), \tag{5.3}$$

where C_a and C_b are the colors, and α_a and α_b are the opacity values belonging to a and b, respectively. This requires depth ordering for correct results. Due to the way our frustums are constructed, we can perform sorting with respect to their Euclidean distance to the new camera position in $O(L\log(L))$ in contrast to $O(S\log(S))$ that would be required when sorting every polygon or frustum on its own (L being the number of supersegment lists while S is the number of supersegments) (Fig. 5.1(d)). For this, we exploit that frustums of the same supersegment list are implicitly sorted already. This significantly decreases the sorting cost and makes it invariant with respect to γ. Note that the ordering within a frustum list needs to be inverted if a supersegment list is viewed in opposite direction with respect to its generating view ray.

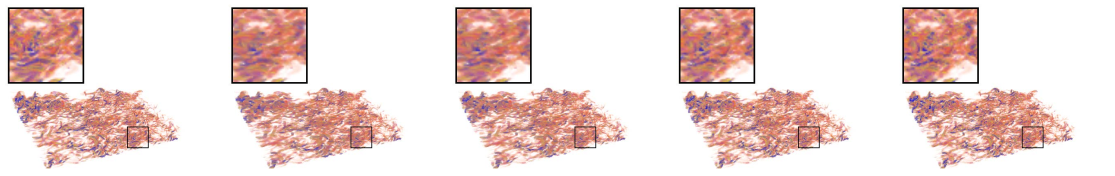
The length l of a new view ray in a frustum (Fig. 5.1(d)) is used to adjust the opacity contribution of the respective part of a supersegment. The opacity contribution of supersegments can be adjusted in the same way as for segments with respect to the step size δ. This can be computed for an arbitrary length l in relation to the original length $\mu(s_b, s_f)$ of the supersegment (Eq. 5.2), where l is determined by intersecting the view ray belonging to the current pixel with the frustum belonging to the supersegment.



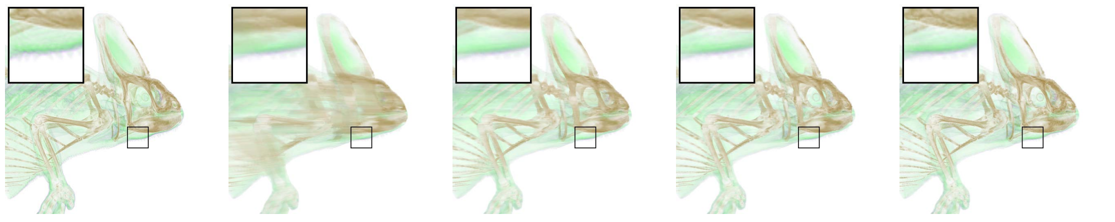
(a) Engine Ref. (b) $\gamma : 1.5, 33ms, 231k$ (c) $\gamma : 1, 33ms, 269k$ (d) $\gamma : 0.5, 34ms, 444k$ (e) $\gamma : .15, 35ms, 965k$



(f) Vertebra Ref. (g) $\gamma : 1.5, 34ms, 366k$ (h) $\gamma : 1, 34ms, 375k$ (i) $\gamma : 0.5, 34ms, 397k$ (j) $\gamma : .15, 35ms, 486k$



(k) Flow Ref. (l) $\gamma : 1.5, 21ms, 63k$ (m) $\gamma : 1, 22ms, 186k$ (n) $\gamma : 0.5, 22ms, 242k$ (o) $\gamma : .15, 22ms, 408k$



(p) Chameleon Ref. (q) $\gamma : 1.5, 33ms, 259k$ (r) $\gamma : 1, 33ms, 415k$ (s) $\gamma : 0.5, 34ms, 617k$ (t) $\gamma : .15, 35ms, 915k$

Figure 5.3: Comparison of ground-truth raycasting (leftmost) and the VDI renderings with decreasing γ from left to right and $50°$ rotation with respect to the original camera configuration. Captions give render time and supersegment count. Closeups are given for the black rectangle.

5.1.3 Results

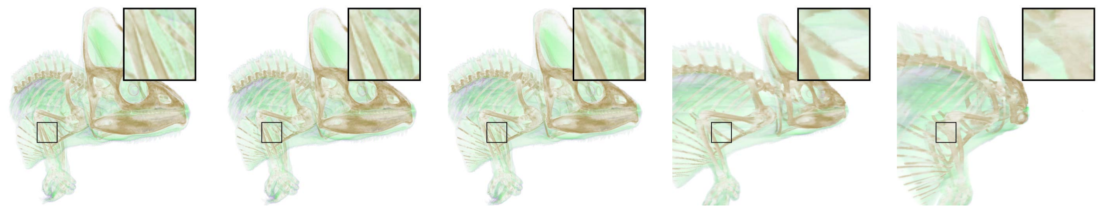
The evaluation was performed by using an Intel Core i7-2600k and a NVIDIA GTX 580 with 3GB of video memory, and an image resolution of 512×512. The list of data sets we used in our experiments, including their standard raycasting timings, is given in Table 5.1. All data sets are given in 16-bit. The timing overhead for generating a VDI

Data set	Resolution	Size	Raycast.	Gen.
Engine	256 × 256 × 256	32MB	23ms	11ms
Chameleon	1024 × 1024 × 1080	2160MB	560ms	54ms
Vertebra	512 × 512 × 512	256MB	48ms	18ms
Flow	2018 × 220 × 1085	919MB	37ms	5ms

Table 5.1: Data sets and their raycasting time for results of Fig. 5.3 with $\gamma = 1$.

during raycasting (Table 5.1, Gen.) is generally low, but varies depending on various factors like the data set and transfer function complexity.

Fig. 5.3 shows VDIs rotated by 50° with respect to their direction of generation, for different settings of γ. In general, the rendering gets crisper and closer to the raycasting reference the lower the value for γ is. This is due to the fact that the supersegments exhibit a lower deviation in color and opacity from the original underlying data, which leads to less blur during rendering. The loss of detail due to too large segments can clearly be seen by means of the almost opaque bone structure of the chameleon data set (q). A similar effect can be observed by the example of the vertebra in (g). (j), (o), and (t) show that particularly for low values of γ, the VDIs provide a good approximation of both the structure and value with respect to the reference volume raycasting.


(a) VDI, 0°, 42ms (b) VDI, 5°, 42ms (c) VDI, 10°, 40ms (d) VDI, 45°, 39ms (e) VDI, 90°, 39ms

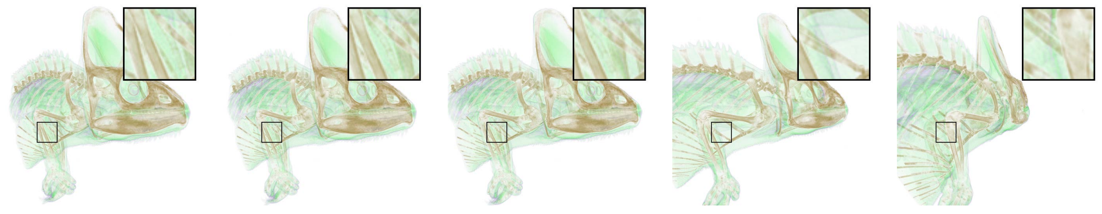

(f) Ref., 0°, 560ms (g) Ref., 5°, 592ms (h) Ref., 10°, 604ms (i) Ref., 45°, 558ms (j) Ref., 90°, 542ms

Figure 5.4: Comparison of ground-truth renderings (a)–(e) with the raycaster and VDI-renderings with $\gamma = 0.8$ and 450k supersegments (f)–(j). The geometry was generated at 0°. Timings for VDI both include sorting and rendering.

Table 5.2 shows that the number of supersegments increases approximately linearly with the number of supersegment lists across different resolutions, and so does the time for geometry creation and rendering. Sorting times are consistently below rendering

Res.	Lists	SupSegs	Geometry	Sorting	Rendering
Engine					
256^2	29k	67k	36ms	5ms	11ms
512^2	118k	269k	142ms	6ms	33ms
768^2	267k	606k	323ms	21ms	71ms
Chameleon					
256^2	29k	103k	48ms	5ms	10ms
512^2	119k	415k	189ms	6ms	33ms
768^2	268k	934k	428ms	20ms	67ms
Vertebra					
256^2	29k	94k	45ms	5ms	10ms
512^2	118k	375k	175ms	8ms	33ms
768^2	267k	846k	398ms	22ms	68ms
Flow					
256^2	15k	46k	28ms	4ms	11ms
512^2	63k	186k	111ms	6ms	22ms
768^2	113k	419k	250ms	8ms	40ms

Table 5.2: Supersegment numbers and timings for $\gamma = 1$. Respective raycasting times are given in Table 5.1.

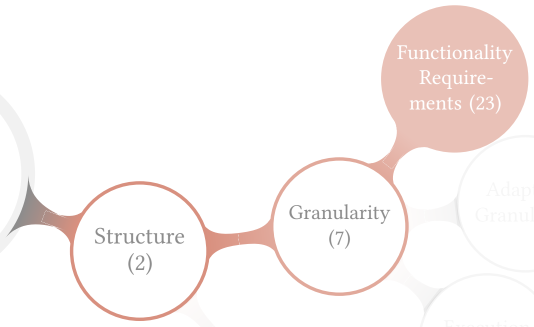
times and only depend on the number of supersegment lists as expected. In our implementation, geometry generation only needs to be carried out once as a preprocessing step and is only executed on a single CPU core with large potential for improvement.

Fig. 5.4 shows the influence of the rotation angle away from the original camera view on quality, as determined by means of the images from VDI rendering and respective reference images from raycasting. The rendered VDI match the raycasting image initially (Figs. 5.4(a) and 5.4(f)), and the visual difference is minor for small rotation angles. It only becomes significant for large angles (e.g., Figs. 5.4(d) and 5.4(e)). Timings for VDI rendering are about one order of magnitude faster in comparison to raycasting, leading to a significant gain from less than 2 fps to over 20 fps (Fig. 5.4). This goes along with a much lower graphics memory usage, enabling the rendering on more limited hardware.

5.1.4 Directions for Further Research

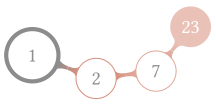
In addition to investigating specific application scenarios in detail, the technique could be extended to further degrees of freedom for interactive exploration as proposed in related research work (e.g., transfer function modification). Additionally, techniques suggesting informative views in volume visualization could be used to select the initial camera configuration (e.g., Zheng et al. [2011]). Further, more elaborate supersegment partitioning criteria could yield higher rendering quality. The extension of VDIs to space-time VDIs that consolidate supersegments across neighboring pixels and time steps for better compression has been performed by Fernandes et al. [2014] and is briefly outlined in Sec. 8.3.1.

5.2 Preview Geometry



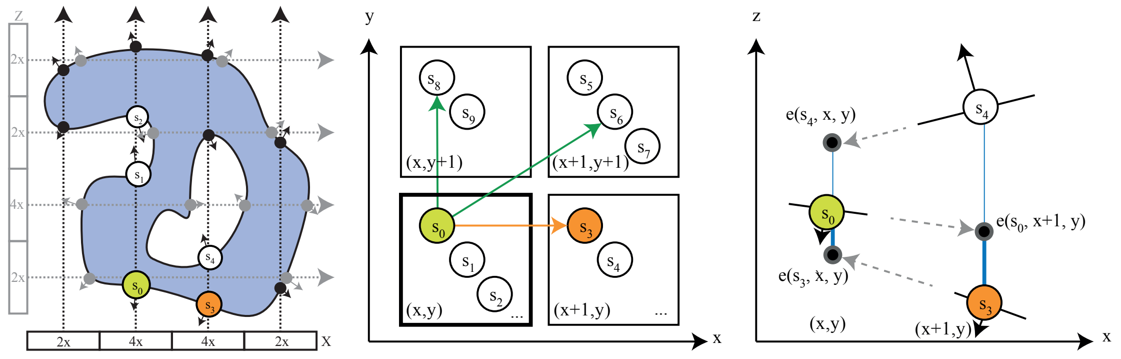
Raycasting data directly can be impracticable due to high storage and computation costs, in particular for more complex representations like radial basis functions from smoothed-particle hydrodynamics, and cell-based fields featuring piecewise polynomial representation from discontinuous Galerkin simulations [Üffinger et al., 2010]. LDIs [Shade et al., 1998] of isosurfaces can quickly be generated and used as a replacement for the actual data to drastically lower hardware requirements, e.g., for preview rendering. However, common LDI rendering methods (warping or splatting) suffer from quality or performance issues in a number of scenarios, and many analysis operations (e.g., distance measurement) are not applicable. A technique is discussed in the following that generates LDIs and subsequently extracts a mesh from these (Strat. 22, *Selective Conversion*). Starting from a quickly generated stub for preview rendering, different methods to improve the mesh quality may flexibly be selected and applied depending on the imposed requirements on the results.

Strategy 23 ***Functionality Requirements***
Different scenarios can have different requirements with respect to the outcome of an application. A program structure that allows to flexibly remove expendable functionality can help to significantly reduce the overall cost.



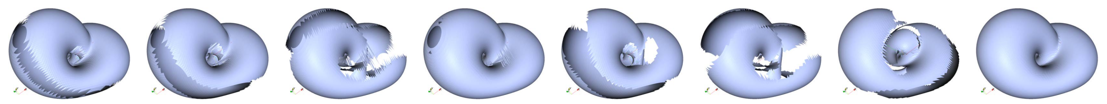
5.2.1 Mesh Generation and Trimming

For generating LDIs with a raycaster, the only modification required is to store depth and gradient information for all hits occurring along a ray (Fig. 5.5(a)). The following discussion limits itself to parallel projection for the sake of simplicity (perspective projection works accordingly with slight adjustments). To form a quad from this, every sample (s_0, located at (x, y)) selects one sample (its best match) each from the right $(x+1, y)$, top $(x, y+1)$ and the top right $(x+1, y+1)$ image position (Fig. 5.5(b)). The



(a) LDIs from two directions (b) Quad Generation (c) Normal-based depth prediction

Figure 5.5: Extraction of isosurfaces on the basis of LDIs from different directions (a). (b) For each sample s_0 at (x, y) of a LDI, quads are potentially generated with samples of the same LDI from $(x + 1, y)$, $(x, y + 1)$, and $(x + 1, y + 1)$. (c) Normal-based depth prediction is used to identify the best match (s_3 in this case, normals are depicted by black arrows).

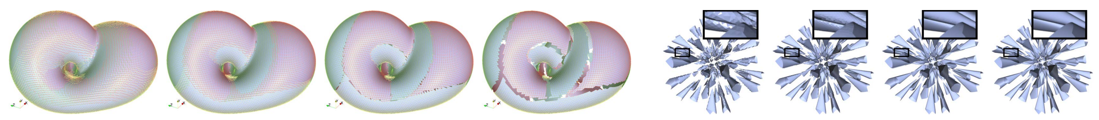


(a) 1d:x (b) 2d:x (c) 2d:y (d) 2d:x,y (e) 3d:x (f) 3d:y (g) 3d:x,y (h) 3d:x,y,z

Figure 5.6: Preview meshes of the KleinBottle with a resolution of 128 × 128 per direction (Tab. 5.3), showing the differences in trimming for the same $r = 0.5$ but a varying set of directions (subcaption depicts the total number of directions as well as the ones that are rendered).

best match is determined by depth predictions based on normals. In detail, s_0 estimates the depth value $e(s_0, x + a, y + b)$ at the image position of the neighbor set ($a, b \in \{0, 1\}$, $(x + 1, y)$ in Fig. 5.5(c)). Likewise, all candidate samples s (s_3 and s_4 in Fig. 5.5(c)) estimate a depth value $e(s, x, y)$ at image position (x, y). The best match for s_0 is then the sample s with the smallest sum of distances $|e(s, x, y) - d(s_0)| + |e(d(s_0), a, b) - d(s)|$, where $d(\cdot)$ returns a sample's depth.

Using a single mesh can deliver a good approximation for slightly varying camera positions, but meshes from multiple directions substantially improve the result for larger surface variations (Figs. 5.6 and 5.7). While our approach works with an arbitrary number of directions, we restrict ourselves to three as a reasonable trade-off between cost and quality (refer to [Frey et al., 2013a] for a more detailed discussion). Meshes from multiple directions cover some surface areas multiple times. Overlaying meshes from different directions for rendering simply requires enabled depth testing. For merging meshes, distinguished boundaries are advantageous which are achieved by trimming the meshes according to their quality of coverage, as defined by means of the view



(a) $r = 0.5$,51k (b) $r = 0.9$, 38k (c) $r = 1$,34k (d) $r = 1$b,31k (e) 64 (f) 128 (g) 256 (h) 512

Figure 5.7: Preview meshes with different parameter settings. (a)–(d) Varying the trimming parameter r with meshes from x, y and z-direction colored in red, yellow and green respectively for the KleinBottle (subcaption depicts r, b in case of additional boundary trimming, and the number of triangles). (e)–(h) Barth data set with $r = 0.5$ for x, y, and z-directions and an increasing number of samples per direction, resulting in higher quality with a higher number of triangles: 64^2 : 24k, 128^2 : 136k, 256^2 : 625k, 512^2 : 2646k.

direction and the surface normal. For every quadrilateral primitive $q \in Q_d$ from view direction $d \in D$, we compute the normal vector n of each of its four vertices $v \in q$ as the cross product with its two neighboring vertices: $n_v = (v - v_{prev}) \times (v - v_{next})$. Whether a vertex v is valid or not is determined by testing for $|n_v \cdot d_q| / \max(|n_v \cdot d|, \forall d \in D) > r$, with d_q being the view direction from which q was generated. The user-adjustable trimming parameter r specifies the extent of surface reduction with respect to its normal and view direction as well as all other view directions. The larger r, the more vertices are classified as invalid and the more primitives are eventually discarded (Fig. 5.7). The choice of r is application-dependent, and should thus be chosen such that there are neither low quality primitives occluding fine details, nor holes in the geometry. Finally, only quads with no invalid vertex remain.

5.2.2 Mesh Refinement

Templates can be employed for refining the meshes to better represent areas of high curvature without generating hanging vertices (i). This requires new samples for the LDI (ii). Refinement not only happens within meshes, but also at boundaries for growing the mesh toward silhouettes (iii).

(i) Refinement Templates. Templates differ for triangle and quad output. For triangles, a classical quadtree approach is used (Fig. 5.8(a)–(c)) with the maximum subdivision level difference between neighboring cells being restricted to one to achieve good quality triangles. After cells are marked for subdivision, an additional iterative *1-level difference* pass is employed that marks cells that have to be subdivided additionally to assure the constraint before generating triangles.

For quad mesh output the 1-level difference pass is substituted with 2-refinement quad templates (Fig. 5.8(d)–(e)) [Schneiders, 1996; Ebeida et al., 2011]. Two templates featuring

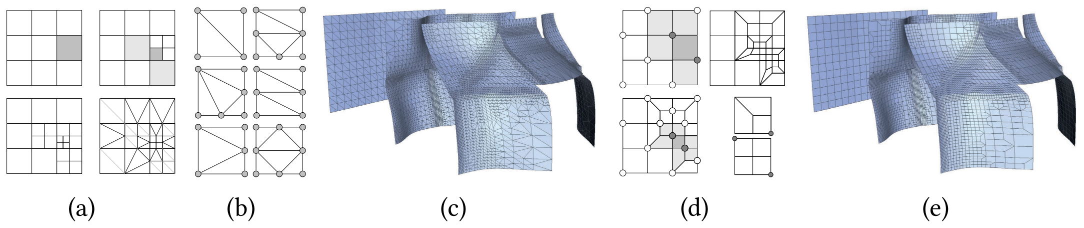


Figure 5.8: Mesh refinement for triangles and quads demonstrated with the higher-order Shock Channel data [Üffinger et al., 2010] (t = 3.0, isovalue 4.6) showing the density of the flow around a box obstacle (c), (e). *Triangles:* (a) The subdivision of a cell (dark gray) forces further subdivisions (light gray) according to the 1-level difference rule. (b) Triangles are generated using six triangle templates (up to rotation and inversion). *Quads:* (d) Subdivision of a cell (dark gray) marks two respective edge nodes (dark circles) and thus cells (light gray). New edge nodes are introduced and previously marked edge nodes are removed. Another cell is chosen for subdivision (dark gray), respective edge nodes are marked and transition cells selected (light gray). Finally, quads are created using two templates.

"reference nodes" (dark circles in Fig. 5.8(d)) have to match so-called *edge nodes* inside the mesh. During subdivision, an edge node is created each time an edge is subdivided. For example, classical quadtree subdivision produces 5 new nodes, one in the center and one on each of the edges of the original cell. Ebeida et al. [2011] propose to subdivide every cell of the initial mesh for producing an initial set of edge nodes. Instead, we identify edge nodes for the initial (unrefined) mesh from the pixel coordinates (x, y) of a vertex by testing if $x + y \bmod 2 = 0$, thus producing a "chessboard pattern" of initial edge nodes. First, edge nodes are identified that belong to cells marked for subdivision (*active edge nodes*, gray circles). Subsequently, all cells sharing such an active edge node are identified (*transition cells* marked by light gray quads) and templates are applied to all identified cells. Finally, previously active edge nodes are removed from the edge node set. A quad is marked for refinement, if any of its vertices is invalid, or the smallest dot product of a vertex normal with all neighboring vertex normals is below a certain value (0.99 proved successful in our experiments).

(ii) LDI Extension Sampling. Requests for additional LDI samples using the raycaster contain the direction and the new image position. All requests of one refinement pass are collected and processed at one go. Quad generation then works analogous to the initial mesh creation process (Section 5.2.1). If no new quad can be generated with the samples, an "invalid" vertex is used instead whose depth and normal are generated by interpolation from surrounding vertices. Invalid vertices are used for refining toward boundaries and silhouettes (iii). Primitives containing invalid vertices at the end of the refinement phase are removed.

(iii) Growing toward Boundaries and Silhouettes. The initial quads might not suf-

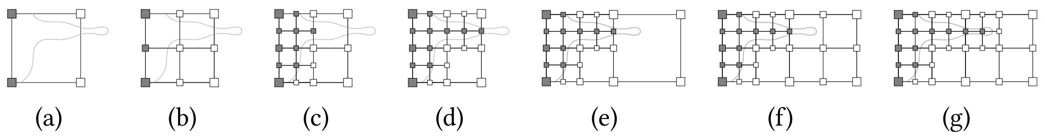


Figure 5.9: Refinement at silhouettes. (a) Initial quad and true silhouette (gray curve). Cells containing true (gray) and invalid (white) samples are refined. (b) After two levels of refinement (c), the edge growing rule marks the right upper quad because it contains a valid sample on its left edge. (d) To resolve the hanging node at the right edge of the original quad, a new quad to its right is generated (e) and subdivided (f)–(g) to the respective level.

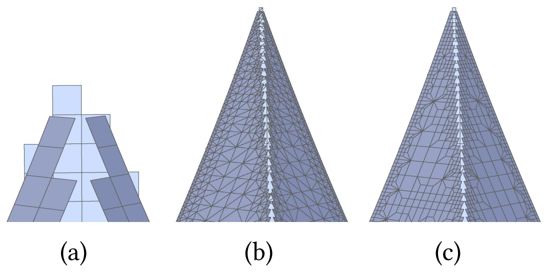


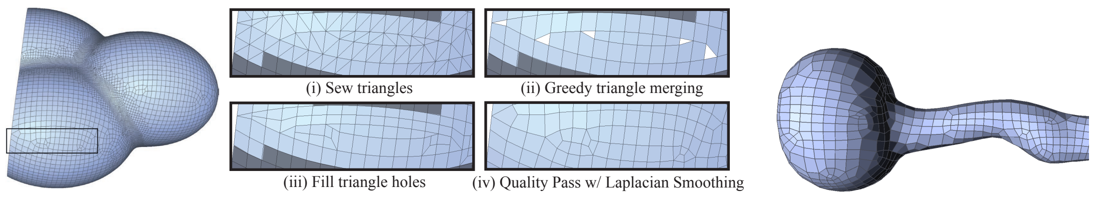
Figure 5.10: Exemplary refinement of (a) toward edges (and corners) with triangles (b) and quads (c).

fice as features can be missed that are smaller than the initial sampling distance. Thus boundaries and sharp tips of the isosurface might not be represented appropriately (Figs. 5.9 and 5.10). To refine toward these (Fig. 5.9(a)), a new top level quad is added if one valid vertex exists on an open edge of an existing top level quad (quad before subdivision) (Figs. 5.9(b)–(d)). Such a vertex represents a hanging node that is resolved by quad subdivision (Figs. 5.9(f)–(g)). Adding a top level quad instead of an already refined one allows efficient recognition of quads growing from different boundaries to a sharing edge, thus preventing mesh overlaps. Furthermore, the refinement templates with their 1-level-difference criterion or marked vertices can be handled more consistently.

5.2.3 Combining Meshes

First, we trim overlapping parts (Section 5.2.1) using $r = 1$ (i.e., keeping only vertices whose normal best matches its original view direction) and then remove the "boundary" layer of cells featuring an edge with no neighbor (e.g., Fig. 5.7(d)). The resulting parts are combined by inserting so-called bridges (i) that connect close meshes. The remaining holes in the connected meshes are filled with triangles (ii) which can be quality-improved (iii), and finally converted to quads *(iv)*.

(i) Bridge Generation. Bridges are quads with one edge e_a being connected to mesh a, one edge e_b connected to another mesh b, and two connecting open edges e_{ab} and


(i) Sew triangles
(ii) Greedy triangle merging
(iii) Fill triangle holes
(iv) Quality Pass w/ Laplacian Smoothing

(a) Triangle to quad conversion for sewing regions (light (i)–(iii)) (b) Quad mesh from CFD flow

Figure 5.11: Merged and refined quadrilateral isosurface meshes from three LDIs.

e_{ba}. Bounding boxes are used to determine the distances of all mesh patch pairs a and b in order to identify the bridge to be inserted with the smallest edge lengths $|e_{ab}|$ and $|e_{ba}|$. The bridge is kept and thus the meshes are merged if neither $|e_{ab}|$ nor $|e_{ba}|$ exceed $l = |e_a| + |e_b|$. In our experiments, the search was stopped early if $|e_{ab}| + |e_{ba}| < l/2$.

(ii) Hole Filling. Bridges reduce mesh combination to a hole filling problem. An ear-cutting algorithm is employed for this due to its simplicity and low computational complexity: iteratively the shortest possible edge is introduced that forms a triangle with two existing open edges until there are no open edges left. While these approaches are “heuristics” that typically cannot be proven to provide the correct result [Held, 2000], refinement and trimming typically provide good-natured problems.

(iii) Triangle Mesh Enhancement. First, all quads that are directly adjacent to hole-filling triangles are split into triangles (resulting in the sewing region Fig. 5.11(a)(i) (light)). Next, edge flips based on edge length are performed. 3-to-1 triangle merges then resolve configurations of three adjacent triangles that are almost coplanar, which is detected by a vanishing determinant of the spanned tetrahedron.

***(iv)* Quad Mesh Generation.** The Catmull-Clark algorithm would generate three quads for each sew triangle, resulting in a high primitive count. Instead, we iteratively combine the pair of triangles that leads to the best quad according to a simple quality metric (ratio of minimum to maximum edge length) (Fig. 5.11(a)(ii)). Typically a small number of triangles remains, for which iteratively the triangle pair with the shortest connecting path is determined using Dijkstra’s algorithm. On this edge-connected path, we go from one triangle to the other, splitting traversed edges by introducing edge vertices (Fig. 5.11(a)(iii)). Passing a quad straight splits the quad in two, while passing adjacent edges splits the quad in three (using the quad refinement template from Fig. 5.8(d)). Finally, quads sharing two edges are merged, the vertex valence is improved via quad edge flips toward four, and Laplacian smoothing is applied(Fig. 5.11(a)(iv)).

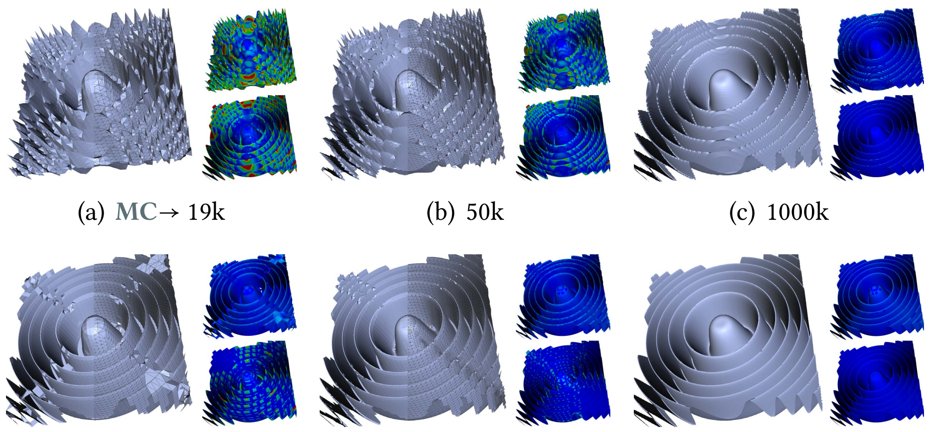
Table 5.3: Timing results in seconds for different data sets and steps of our approach (if executed) on a single core of an Intel Core i7 with 3.4 GHz. Timings do not include raycasting times to generate the underlying LDI.

Fig.	Mesh.	Ref.	Bridge	Sew	Qual.	Tot.
Klein Bottle [Knoll et al., 2009a] (128^2)						
5.6, 5.7	0.04	–	–	–	–	0.04
Barth [Knoll et al., 2009a] ($64^2, 128^2, 256^2, 512^2$)						
5.7(e)	0.02	–	–	–	–	0.02
5.7(f)	0.11	–	–	–	–	0.11
5.7(g)	0.43	–	–	–	–	0.43
5.7(h)	1.84	–	–	–	–	1.84
Marschner-Lobb [Marschner and Lobb, 1994] ($64^2, 512^2$)						
5.12(d)	0.08	–	0.11	0.04	0.23	0.46
5.12(e)	0.08	0.73	0.62	0.1	0.79	2.32
5.12(f)	7.19	–	0.82	3.38	22.07	33.46
Coulomb [Knoll et al., 2009a] 64^2						
5.11(a)	< 0.01	< 0.01	0.01	0.01	1.84	1.86
Sphere [Üffinger et al., 2010] 64^2						
5.11(b)	< 0.01	–	< 0.01	0.01	0.40	0.40
Shock Channel [Üffinger et al., 2010] 64^2						
5.8	< 0.01	0.01	–	–	–	0.01
Slices $16^2, 24^2, 32^2$ $x^2 + y^2 + z^2 + \sin(5x + 15y + 6z) - 1$						
5.13(b)	< 0.01	–	< 0.01	< 0.01	< 0.01	< 0.01
5.13(d)	< 0.01	–	0.01	0.01	< 0.01	0.02
5.13(f)	0.02	–	0.02	0.01	0.04	0.09

5.2.4 Results

For evaluation, we use the higher-order unstructured grid raycaster by Üffinger et al. [2010] using CUDA, and the implicit surface raycaster by Knoll et al. [2009a] implemented in Cg. Data sets and timings are listed in Tab. 5.3. Results were obtained using three viewing directions along the x, y, and z-axis unless otherwise noted. Fig. 5.7(a)–(c) show that the larger r, the larger the mutually covered regions, thus reducing the risk of holes, but also potentially leading to invalid coverings. In our experience, $r = 0.5$ provides a good trade-off overall. The timings (Tab. 5.3) also suggest that r could be interactively adjusted to best fit the data set and the requirements of the user. Fig. 5.7(d) shows $r = 1$ with the additional boundary trimming prior to sewing. Preview meshes generated from different LDIs resolutions (Fig. 5.7(e)–(h)) provide more details for higher resolutions with better coverage of thin features and strong curvature with an approximately linear relation between primitive numbers and generation time. Even the generation of high-resolution meshes at interactive rates would be possible, considering the large improvement potential through parallelization. An LDI from only one direction may already suffice depending on the nature of the isosurface (Fig. 5.8).

In Fig. 5.12, we demonstrate the accuracy of our approach using geometric distances in comparison to meshes from MC [Cignoni et al., 1998] with an approximately equal triangle count. We compare all vertices of the candidate mesh to a reference (MC with 5M triangles) (forward) and all vertices of the reference to the candidate (backward). In contrast to MC with a similar triangle count, our meshes deliver close to perfect results with forward comparison, and are also consistently better for backward comparison


(a) **MC**→ 19k (b) 50k (c) 1000k

(d) **Our**→ 64^2 (e) 64^2, ref. (f) 512^2

Figure 5.12: Comparison of our approach (bottom row with LDI resolution, “ref.” denotes refinement) to MC (top row with triangle count) using the Marschner-Lobb signal, with vertical image pairs having approximately the same triangle count. Forward and backward geometrical distances are additionally depicted (top and bottom right) using a rainbow color map from 0 to 0.005 and 0.03, respectively.

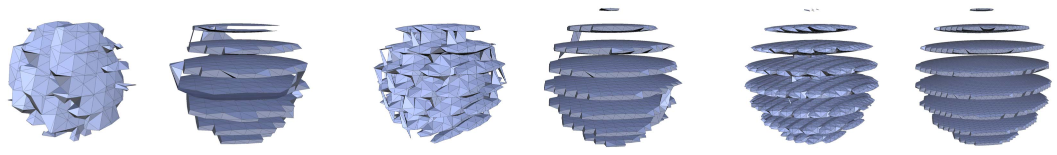

(a) **MC**, 1.6k (b) **Our**, 16^2 (c) **MC**, 3.8k (d) **Our**, 24^2 (e) **MC**, 15.1k (f) **Our**, 48^2

Figure 5.13: Comparison of results from our approach (LDI resolution given) to meshes from MC (triangle count given) with a similar triangle count using the Slices data set.

(particularly at signal peaks in comparison to the jagged coverage of MC). Much better results than MC (Fig. 5.12(a)) are also achieved for very low resolutions, despite some artifacts due to insufficient sampling (Fig. 5.12(d), incorrect bridges, and hole filling as both steps rely on sampling distance for finding correct matches). Refinement leads to smaller gaps between mesh patches that belong together topologically and thus reduces artifacts (Fig. 5.12(e)). However, although significantly decreasing, small holes in the peaks of the signal in boundary regions even persist with fine sampling (Fig. 5.12(f)) as connections across the peak are shorter than along the peak. This could be fixed by advanced bridging and hole filling heuristics considering normal variation in addition to distance. Nevertheless, as also shown by Fig. 5.13, from low to high resolution, our approach was able to generate a more detailed approximation for roughly equal triangle counts.

5.2.5 Directions for Further Research

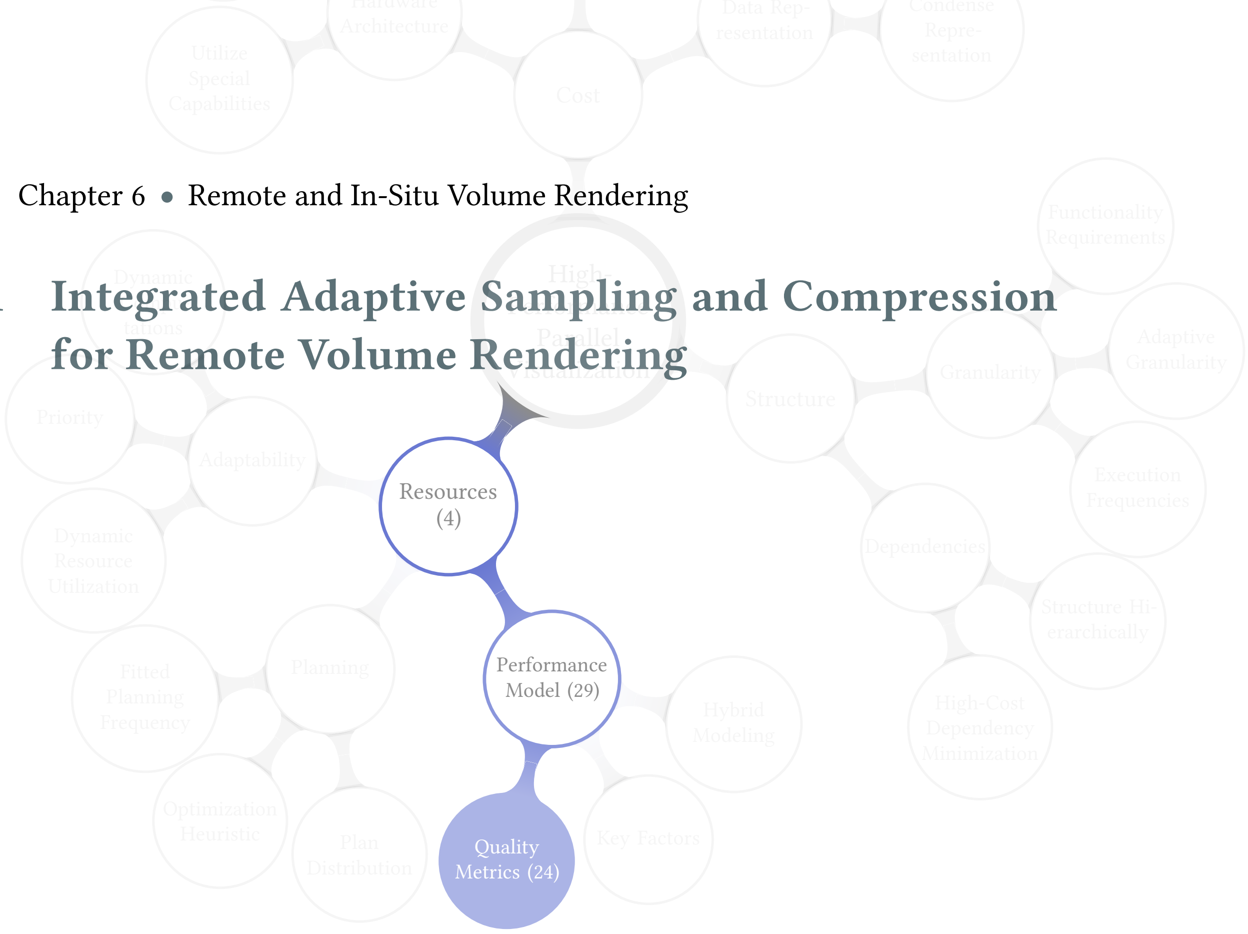
A data-dependent LDI view selection would allow to accurately and efficiently capture the views that are of interest. Additionally, a more in-depth evaluation of the technique in comparison to other meshing techniques beyond classic MC is required, e.g., by using topology verification techniques [Etiene et al., 2012]. Further work might also include considering different mesh combination strategies, testing our algorithm with other type of higher order data as well as parallelizing expensive stages of the pipeline.

Chapter 6

Remote and In-Situ Volume Rendering

Storage and network bandwidth constraints more and more become the limiting factor for overall compute performance. This has significant impact on almost all applications run on larger scale machines, as they typically not only need to process a large number of operations, but also produce and/or require significant amounts of data. One general approach to alleviate this issue is to directly process the data as locally as possible where it has been generated by simulation or reconstruction from sensor data. For instance, data size can be significantly reduced by extracting certain structures or transforming it to any kind of different representation (like the view-dependent representations discussed in Ch. 5). A slightly different approach is to not generate and transmit any interaction-capable representation at all, but to visualize data remotely on-demand on the machine that originally generated it.

6.1 Integrated Adaptive Sampling and Compression for Remote Volume Rendering



In interactive remote rendering, there are basically two major components causing the response delay to user input: the generation of the visualization and the transfer of the resulting image. In practice, the time required in each one is influenced by a variety of factors, including the power of the hardware used for rendering, the complexity of the data set, the viewpoint, the quality of the network connection of the client to the server, and so on. As a result, in different scenarios, the achieved latency varies hugely. Typically, the image generation and transfer parts are controlled independently by specifying the quality of the visualization and the quality of the lossy compression used for transfer. As a consequence, different parameter settings need to be chosen in each scenario to harmonize image generation and transfer and yield the most optimal result with respect to user-defined constraints.

In this project, we attempt to optimize remote rendering with the constraint of a fixed setting regarding the response latency to a user request. For this, we use a local heuristic that aims to achieve the best quality for each individual frame. While the cost for rendering and transfer need to be reduced individually, they need to be related in a meaningful way. To achieve this, our approach for remote visualization shares concepts, criteria, and computation steps for integrated lossy frame compression and adaptive volume raycasting. Without a tight integration, high effort for rendering would eventually be wasted by subsequent compression, or data transfer size would be unnecessarily large with respect to the details provided in the frame. For this tight integration, we not only do compression but also adaptive sampling in the frequency domain. This allows us to directly relate the loss that stems from undersampling and

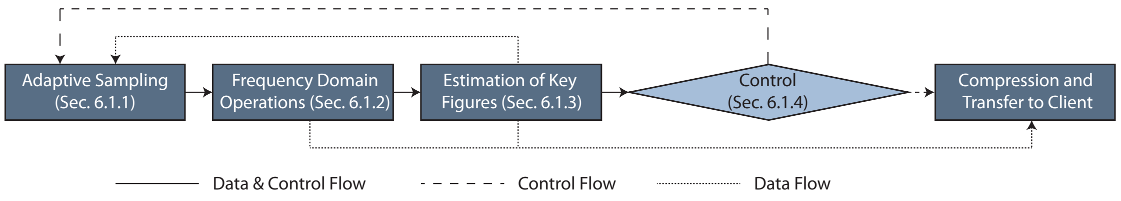
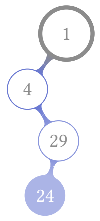

Adaptive Sampling (Sec. 6.1.1)
Frequency Domain Operations (Sec. 6.1.2)
Estimation of Key Figures (Sec. 6.1.3)
Control (Sec. 6.1.4)
Compression and Transfer to Client
Data & Control Flow
Control Flow
Data Flow

Figure 6.1: Processing a frame on the server.

lossy compression, and map it to a combined image quality metric. For this purpose, we use multi-scale structural similarity (MSSSIM) [Wang et al., 2003]. MSSSIM is a full-reference metric (i.e., it is computed with respect to the perfect image), thus this mapping needs to be generated offline. For this, we use a series of representative camera paths.

Strategy 24 ***Quality Metrics***
Quality metrics quantifying the achieved outcome from a users perspective can provide more meaningful target values for optimization.

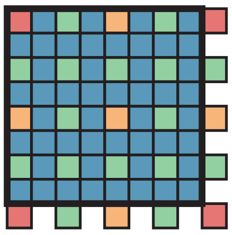


Our approach follows an extended progressive visualization procedure on the server (Fig. 6.1). After refining the current frame by means of adaptive sampling (Sec. 6.1.1), we update the frequency representation of the modified parts (Sec. 6.1.2). The frequency representation is then used to estimate a number of key figures required in the following, including the compression quality that yields the requested latency based on the time spent for sampling already, and an estimation of the overall frame quality (Sec. 6.1.3). Based on this, it is decided whether the current frame is further refined in another iteration, or whether it is compressed and transferred to the client (Sec. 6.1.4).

6.1.1 Adaptive Sampling

Blocks. The image space is partitioned into blocks. While we restrict ourselves to 8×8 blocks in the following, basically any block size could be used. Whenever a block is processed, the additionally required samples for the next resolution level l need to be generated by the renderer. For this, predefined sampling patterns are used with respect to l (Fig. 6.2).

Since determining deviation values δ that are required for control and adaptive sampling (discussed in more detail below) are computed using subsequent resolution levels, resolution levels 0 and 1 are computed in the same step. Here, resolution level 1 does



Resolution Level	#Additional Samples (S)	#Frequency Values (N)
0	🟥 1	1
1	0	4
2	🟧 3 to 5	9
3	🟩 12 to 16	25
4	🟦 48	64

Figure 6.2: Image-space sampling patterns for different resolution levels.

not require any additional samples, as values from other blocks are used (Fig. 6.2). Similarly, resolution levels 2 and 3 also use adjacent pixels from the neighboring right, bottom, and bottom right blocks, respectively. This sharing of samples requires each block to check whether the neighboring right, bottom and bottom right blocks are also updated to the next level. If they are not, their shared samples at the border need to be added extra.

In case of a change to the render configuration between frames, all blocks are reset to level $l = -1$ and refined to $l = 1$ in the first iteration. Otherwise, refinement just continues where it left off from the computation of the previous frame.

Refinement Iterations. For adaptive sampling, the blocks are sorted with respect to their deviation value δ (Sec. 6.1.3). Blocks that have reached their full resolution level are always set to $\delta = 0$ and removed from the list. To be able to leverage massively parallel hardware like GPUs, samples for several blocks need to be generated in parallel in each sampling iteration. Starting from the block with the highest δ, we select blocks for refinement with decreasing δ, until a predefined chunk size is reached. Similar to Sec. 4.2, this chunk size should be chosen such that it is big enough to fully occupy the target hardware, but also as small as possible for sufficient granularity.

6.1.2 Frequency Domain Operations

Frequency Domain Transform. Transform and quantization work akin to common image and video compression techniques. In particular, we orient ourselves at the procedure of JPEG [JPEG, 1992]. First, the image is transformed into YCbCr color space. Each (8×8) block is then transformed individually to the frequency domain using the discrete cosine transform (DCT).

In order to obtain a frequency representation for the incomplete sampling patterns from Fig. 6.2, i.e., where not all 8×8 pixels of a block are given, we use the approach due to Bolin and Meyer [Bolin and Meyer, 1995] to estimate two-dimensional type-II DCT coefficients $G_{u,v}$ from arbitrary samplings. Their approach employs least-squares fitting and obtains S frequency coefficients from S samples. The least squares solution for $G^c_{u_i,v_i}$ with $0 \le i < S$ from the S monochromatic samples $x^c_{x_s,y_s}$, located at positions (x_s, y_s) within the block, is given by

$$\underset{G_{u_i,v_i},\, 0 \le i < S}{\arg\min} \left(\sum_{s=0}^{S-1} (g^c_{x_s,y_s} - r^c_{x_s,y_s})^2 \right), \tag{6.1}$$

with $r^c_{x_s,y_s}$ representing the sampled values for either of the Y, C_b or C_r color channels. Eq. 6.1 is then solved, with X being the transformed set of frequencies and Y denoting the transformed set of samples:

$$X = J^{-1}Y, \tag{6.2}$$

with

$$J_{i,j} = \sum_{s=0}^{S-1} C_{x_s,u_i} C_{y_s,v_i} C_{x_s,u_j} C_{y_s,v_j} \tag{6.3}$$

$$Y_i = \sum_{s=0}^{S-1} r^c_{x_s,y_s} C_{x_s,u_i} C_{y_s,v_i} \;, \qquad X_i = \frac{1}{4}\kappa(u_i)\kappa(v_i) G^c_{u_i,v_i}. \tag{6.4}$$

In our implementation, we solve for $G^c_{u,v}$ using a precomputed $N \times S$ matrix for each resolution level. This can be done by a simple matrix multiplication with Y, the transformed set of samples of the block. We implemented this on the GPU such that one frequency value $G^c_{u,v}$ is computed per thread.

Frequency Weighting. The human eye is geared toward detecting small brightness variations over comparably large areas, but is not so good at distinguishing the exact strength of a high frequency brightness variation. To account for this, frequencies are weighted with factors depending on their position within a block and their channel c (luminance or chrominance). For this, we use the JPEG quantization matrices Q^c (see JPEG Standard [JPEG, 1992], Annex K, Luminance and Chrominance). In detail, the weighted components $\bar{G}^c_{u,v}$ for channel c of $G_{u,v}$ are then simply determined as follows:

$$\bar{G}^c_{u,v} = \frac{G^c_{u,v}}{Q^c_{u,v}}. \tag{6.5}$$

Frequency Deviation. For estimating the overall frame quality as well as for ordering the blocks for adaptive sampling, we determine the deviation δ for each block. In this process, we distinguish between spatial deviation δ_s and temporal deviation δ_t. Spatial deviation quantifies the difference between the current resolution level l of a block, and its previous resolution level $l-1$ for the currently computed frame f by determining the maximum deviation of any weighted frequency component $\bar{G}_{u_i,v_i}$:

$$\delta_s = \max_{0 \le i < |\bar{G}(l,f)|} \left(|\bar{G}_{u_i,v_i}(l,f) - \bar{G}_{u_i,v_i}(l-1,f)| \right). \tag{6.6}$$

Temporal deviation δ_t is computed similarly, yet it quantifies the difference between the current frame f and the previous frame $f-1$:

$$\delta_t = \max_{0 \le i < |\bar{G}(l,f)|} \left(|\bar{G}_{u_i,v_i}(l,f) - \bar{G}_{u_i,v_i}(l_{\max}, f-1)| \right). \tag{6.7}$$

In the end, the total deviation δ is determined as the minimum of the spatial and the temporal deviation:

$$\delta = \min(\delta_s, \delta_t). \tag{6.8}$$

The minimum is taken because ultimately only one of the measures is adequate, as discussed in more detail in the following paragraph.

Reusing Previous Frequencies. For each block individually, it is decided whether frequencies from the previous frame should be reused. This is done with respect to spatial and temporal deviation values. If the temporal deviation is smaller, frequencies are taken from the previous frame for the positions for which no frequencies have been computed yet:

$$\bar{G}(l_{\max}, f-1) + (\bar{G}(l,f) - \bar{G}(l,f-1)) \tag{6.9}$$

The heuristics behind this is that if the difference to the previous frame for a block is low, then it is assumed that the block remained largely the same, and if it has been previously refined to a high resolution, taking its frequencies is more adequate than using a low-resolution version of the current frame.

6.1.3 Estimation of Key Figures

The main constraint in our approach is the specified latency l from a user request to the response. As indicated above, in the following discussion, we restrict ourselves to the two major cost factors frame generation and transfer. Accordingly, depending on the amount of time t_f spent for processing a frame already, $l - t_f$ time remains for further sampling and data transfer. Several measures are computed in this step to determine how latency l can be achieved, and to supply the control component with the basis to decide how the remaining processing time $l - t_f$ of the frame is spent most efficiently.

First, we aim to determine the compression quality that would lead to data size τ and eventually require $l - t_f$ to transfer the compressed frame to the client. Below, we first quickly outline JPEG's quantization process that we use in the following (a), and how we can quickly estimate the resulting data size after entropy encoding (b). Based on this, the compression quality is estimated that yields the target data size τ (c). With the current state of sampling and the determined compression quality, we estimate the resulting overall quality of the frame (d). Finally, estimations regarding the achieved overall quality after one more sampling iteration are carried out (e).

(a) Compression. In JPEG, quantization of the frequency terms is steered by the user-defined quality term q with $0 < q < 1$:

$$\hat{G}^c_{u,v} = \left\lfloor \frac{\bar{G}^c_{u,v}}{\alpha} \right\rceil, \ \alpha = \begin{cases} 1/(2q), & \text{if } q \leq 1/2 \\ 2 - 2q, & \text{otherwise.} \end{cases} \tag{6.10}$$

Hence, the lower q, the stronger the quantization, yielding a higher loss in quality but also allowing for stronger compression. For further increasing the efficiency of the compression, we use the transferred frequencies from the previous frame for determining a residual. For this, as the current compression quality q_f may differ from the compression quality of the previous frame q_{f-1}, the previous (quantized) frequencies are scaled accordingly by a factor of $\alpha(q_{f-1})/\alpha(q_f)$ to account for this. Finally, a combination of LZ4 and Huffman compression is used for entropy encoding.

(b) Data Size Estimation. Compression Θ finally reduces the size of the quantized frequencies. As full compression is relatively expensive, we use an estimation $\tilde{\Theta}$ to quickly approximate the resulting data size. This is required for determining q in the subsequent step (c). Our experiments showed that just 10% of the original data is already enough to allow for a sufficiently exact estimation in almost all cases. As entropy encoding works more efficiently on larger data sizes, the outcome of the partial compression cannot just be scaled up by a factor of ten. To determine the factor that should be used, we carry out simple linear fitting without a constant term with partial and full compression value pairs over the last 10 frames. This can be done easily, as the ground truth Θ is evaluated every time the full frame is compressed for transfer to the client. In our experiments, this such factors typically ranged from 6–8.

(c) Compression Quality Determination. This fast way to estimate the size of the compressed data now allows us to optimize the compression quality q_τ such that the target data size τ is met:

$$q_\tau = \underset{q,\ 0<q<1}{\arg\min} \left(\tilde{\Theta}(\hat{G}, q) - \tau \right). \tag{6.11}$$

In our implementation, we use a simple bisection approach to solve this optimization problem.

(d) Frame Quality Estimation. To estimate the overall frame quality, we use a pre-computed mapping function from the quantification factor from image compression and the root-mean-square deviation (RMSD) δ_Ω of deviations δ over all blocks from sampling:

$$\delta_\Omega = \sqrt{\frac{\sum_{0 \leq i < |B|} \delta(i)^2}{|B|}}. \tag{6.12}$$

As indicated previously, we use MSSSIM (Ξ) to quantify frame quality in the range $[0, 1]$, with 0 denoting the maximal difference between two images, and 1 denoting the

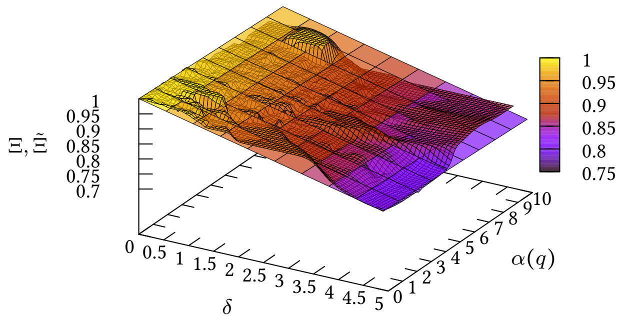


Figure 6.3: Measured MSSSIM data with Ξ (grid from measured data) and its estimation $\tilde{\Xi}$ (planar fit with Eq. 6.14) generated by Levenberg-Marquardt fitting.

identity of two images (i.e., full quality in our context) [Wang et al., 2003]. Evaluating Ξ on-the-fly is impractical, as it is both computationally expensive and requires a full sampling (with frequency representation Γ) for reference. Thus, we estimate Ξ with $\tilde{\Xi}$ on the basis of δ_Ω and q_τ:

$$\Xi(G, \Gamma) \approx \tilde{\Xi}(\delta_\Omega, q_\tau). \tag{6.13}$$

In detail, we compute $\tilde{\Xi}$ as follows:

$$\tilde{\Xi}(\delta, q) = a \cdot \delta + b \cdot \alpha(q) + 1. \tag{6.14}$$

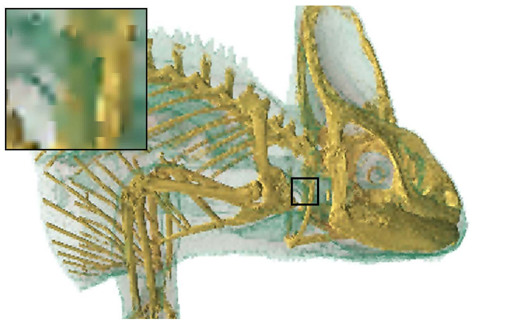
The matching of the parameters a and b is carried out offline using a data base of precomputed values of Ξ and their respective δ_Ω and q_τ by means of the nonlinear least-squares Levenberg-Marquardt algorithm (Fig. 6.3). Note that the fit works well in practice because both δ and $\alpha(q)$ represent values in the same domain (deviations of frequency terms).

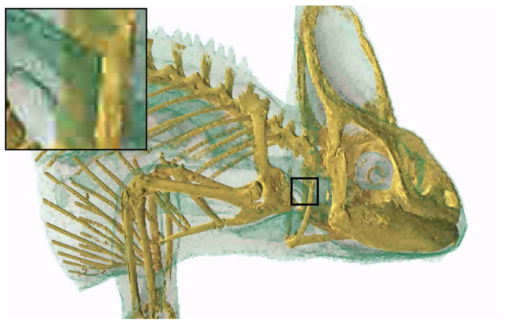
(e) Prediction of Frame Quality after the Next Iteration. From the previous and current iteration, the frequency deviation value of the next frame needs to be predicted for the control stage (Sec. 6.1.4). It is computed as the difference between the RMSD for the maximum and minimum of the deviation values of each block i, with j denoting the current sampling iteration:

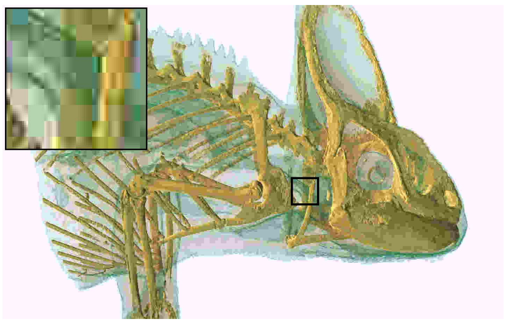
$$\Delta\delta_\Omega = \sqrt{\frac{\sum_{0 \le i < |B|} \max(\delta(i,j), \delta(i,j-1))^2}{|B|}} - \sqrt{\frac{\sum_{0 \le i < |B|} \min(\delta(i,j), \delta(i,j-1))^2}{|B|}}. \tag{6.15}$$

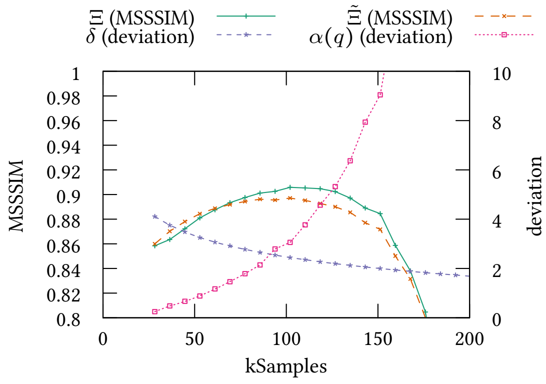
6.1.4 Control

Local Control Heuristic. In the control stage, we aim to optimize the quality of each frame as specified by quality metric Ξ. This should be achieved by steering the size of


(a) kSamples=28, $q = 0.87$, $\Xi = 0.85$


(b) kSamples=102, $q = 0.16$, $\Xi = 0.91$


(c) kSamples=151, $q = 0.05$, $\Xi = 0.88$



(d) Development of key figures within a frame.

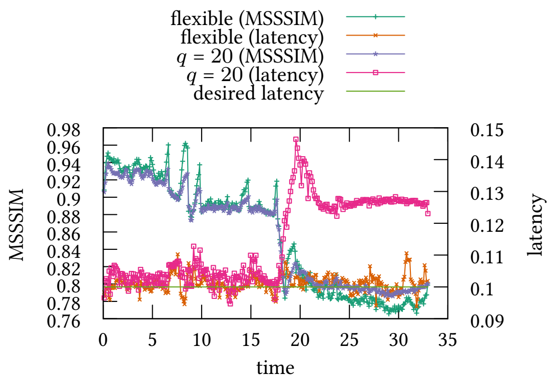
Figure 6.4: (a)–(c) Selected frames within the refinement of a frame and (d) representative development of differences and measures. This was taken from the first frame of the Chameleon data set, without interrupting the frame when control detected the optimum.

the allocated time slots for sampling and transfer, respectively. Our local heuristic can be seen as a kind of (limited) hill climbing optimization. In detail, sampling is stopped and the frame is transferred to the client if the current frame quality is higher than the predicted frame quality after the next iteration:

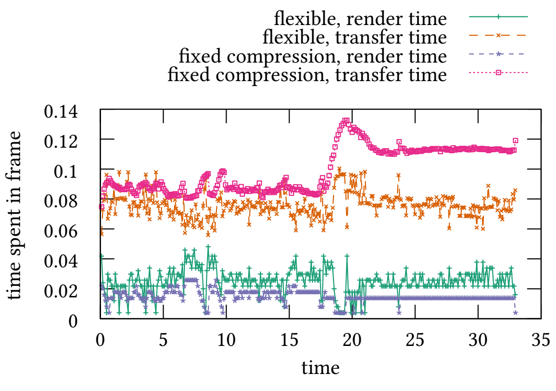
$$\tilde{\Xi}(\delta_\Omega, q_\tau) > \tilde{\Xi}(\delta_\Omega + \Delta\delta_\Omega, q_{\tau'}). \tag{6.16}$$

Here, τ' is the determined target data size, based on the assumption that the next sampling iteration takes as long as the current sampling iteration.

Local Versus Global Optimization. Note that this is a local heuristic targeted at optimizing each frame individually instead of the whole sequence of frames in an interactive rendering session. An optimal choice over a whole sequence of frames would require knowing (or estimates regarding) future events. For instance, spending more time on sampling can be advantageous overall if the render configuration does



(a) Image quality and latency.



(b) Time usage.

Figure 6.5: Flexible settings against a fixed quality setting of $q = 0.2$, a bandwidth of 1MB/s and a rendering performance of 2M samples per second.

not (or only marginally) change for the next rendered frame, as samples can be reused directly. In such a case, quality of the current frame might be sacrificed due to stronger compression as less time is available for transfer, but the more refined image provides a better basis for continued sampling in upcoming frames. As exact knowledge about future events is intrinsically not available in interactive applications, assumptions are required to still employ a global optimization approach (e.g., Sec. 4.2).

6.1.5 Results

For the purposes of detailed evaluation, the comparability of different variants, and flexible development, we utilize a simple simulation of an interactive remote rendering application on the basis of precomputed reference camera paths. For this, we used the Chameleon and Flower camera paths that were also used in Sec. 4.2. Regarding the cost of fundamental components of or approach, frequency transforms can easily be executed in parallel, and basically come down to floating point MAD (multiply-add) operations that are very efficient on GPUs. In our experiments, these operations took around 2-10 ms for transforming a full frame on a NVIDIA GTX580. A major cost factor is the full compression of a frame using a combination of Huffman and LZ4 compression which takes around 30-40 ms on an Intel Core i7-2600k.

Fig. 6.4 shows selected snapshots during the computation of an exemplary frame (a)–(c) as well as a chart of the development of key figures (d). It can be seen that the deviation due to sampling δ and the deviation due to compression $\alpha(q)$ monotonously decrease and increase, respectively, with the number of samples taken. In the case of compression, this is due to the fact that less time is available for transfer and thus a

	kSamples/s	kB/s	data	q%	min(Ξ)	max(Ξ)	rmsd(Ξ)	max(t)	rmsd($t - l$)
a	2048	1024	Chameleon	0	0.846	0.977	0.114	0.127	0.003
b	2048	1024	Chameleon	20	0.849	0.972	0.122	0.145	0.021
c	2048	1024	Flower	0	0.817	0.964	0.130	0.111	0.003
d	2048	1024	Flower	20	0.819	0.938	0.135	0.146	0.018
e	2048	8192	Chameleon	0	0.931	1.000	0.034	0.115	0.002
f	2048	8192	Chameleon	20	0.923	0.983	0.042	0.105	0.002
g	2048	8192	Flower	0	0.915	1.000	0.053	0.103	0.001
h	2048	8192	Flower	20	0.909	0.983	0.059	0.105	0.002
i	8192	1024	Chameleon	0	0.882	0.982	0.085	0.136	0.004
j	8192	1024	Chameleon	20	0.850	0.980	0.120	0.134	0.016
k	8192	1024	Flower	0	0.850	0.982	0.091	0.112	0.003
l	8192	1024	Flower	20	0.818	0.955	0.126	0.136	0.012
m	8192	8192	Chameleon	0	0.979	1.000	0.008	0.122	0.002
n	8192	8192	Chameleon	20	0.973	0.983	0.019	0.103	0.001
o	8192	8192	Flower	0	0.958	1.000	0.016	0.106	0.001
p	8192	8192	Flower	20	0.967	0.983	0.022	0.102	0.001

Table 6.1: Results for different balancing settings and input scenarios.

lower value for q needs to be taken to achieve the target latency. This reverse behavior of the two functions typically results in one maximum for the overall quality measure Ξ, as well as its estimated counterpart $\tilde{\Xi}$.

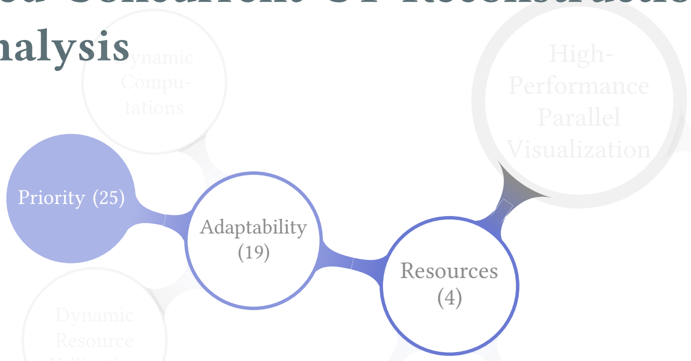
Fig. 6.5 depicts the performance of flexible balancing against a hand-tuned, yet fixed compression quality setting $q = 0.2$. Fig. 6.5(a) shows major drawbacks of a fixed compression quality setting. From 0 s to 18 s, the content of the frame changes only slowly, and higher compression quality is required to reach higher quality overall. From 1 8s, frames change more strongly between time steps, and the given latency constraint of 0.1 s is significantly exceeded by the fixed quality setting. Fig. 6.5(b) shows that, with respect to the fixed compression setting $q = 0.2$, transfer time is traded for rendering time in the flexible approach, allowing for higher quality overall.

Tab. 6.1 shows measures for different data sets, rendering speeds, bandwidths, and compression quality settings. Overall, it can be seen that our approach flexibly adapts to systems with different performance characteristics, in terms of quality values Ξ performs better than the tuned fixed setting ($q = 20\%$). Furthermore, for the low bandwidth settings (1024 kB/s), the fixed quality setting often significantly exceeds the target latency time (Tab. 6.1b, d, j, and l). This may also happen in the variable setting (a), yet the RMSD is still much lower than that of $q = 0.2$ (b). Here, it is not directly a consequence of the steering heuristic but rather due to mispredictions of the data size of the full compressed frame.

6.1.6 Directions for Further Research

The evaluation of the approach was carried out using a simulated interactive remote rendering system as this allows for a simpler, cleaner evaluation of the core concepts, minimizing side effects that occasionally occur when doing timing measurements. However, as a next step, the full system needs to be evaluated in practice, and compared against other remote rendering systems. Furthermore, with the discussed approach as a basis, more flexible, adaptive adjustments could also be applied directly (like the one discussed in Sec. 4.2). In particular, it would be interesting to compare the local approach of optimization taken here to global optimization approaches like in Sec. 4.2. Furthermore, as entropy encoding consumes a significant share of processing time, we would like to consider recently developed GPU entropy encoding methods as drop-in replacements for our current technique (e.g. [Patel et al., 2012]). This could further help with the issue of the occasional mispredictions of compression data size that may lead to violation of the latency restrictions when underestimating the resulting data size. In addition, further means of estimation should be evaluated. Finally, we aim at integrating more advanced sampling patterns to potentially improve the rendering results particularly in low-quality settings.

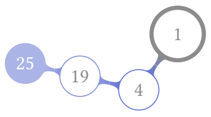
6.2 Distributed Concurrent CT Reconstruction and Visual Analysis



Modern CT scanners acquire high-resolution 2D X-ray images from a huge number of different angles at rapid pace. Engineers and scientists require a high-resolution reconstruction and segmentation, while it is often also critical to have the results, e.g., of a non-destructive quality test, at hand as early as possible. However, volumetric reconstruction from X-ray images is very time consuming, even with fast GPU reconstruction algorithms (the most popular method due to Feldkamp et al. [1984] has a runtime complexity of $O(N^4)$).

Here, to alleviate this issue, the reconstruction and segmentation processes of volumetric bricks are distributed to a GPU cluster. By employing a hybrid mult-resolution renderer, finished full-resolution parts are successively displayed in the context of a low-resolution volume. The low-resolution data set can be created quickly by a single GPU within several seconds on a single node. The full-resolution volume is progressively created by the cluster nodes, that also segment and render their respective blocks. As in practice some bricks are of higher interest to the user than others, bricks can be prioritized graphically in a low-resolution preview volume.

Strategy 25 ***Priority***
Different areas or parts of the result might be more important to the user than others. Computing them with higher priority to make them available earlier can help the user to reach conclusions more quickly.



6.2.1 Architecture Overview

The integrated reconstruction and visualization system consists of two classes of nodes: a single front-end node and one or more back-end nodes. The front-end node displays the volume rendering and exposes the interface that allows the user to influence the order of reconstruction. For this, it generates a low-resolution preview reconstruction of the volume at startup, that is subsequently used for rendering the parts of the volume

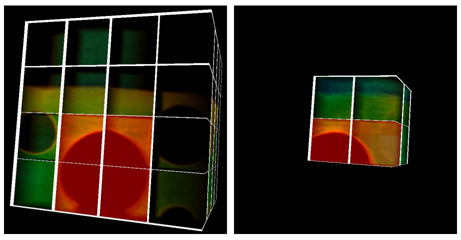


Figure 6.6: The user can find interesting regions by stepping through the layers and simply click on a sub-volume that should be prioritized in reconstruction or changed in rendering mode (selected sub-volumes are highlighted red).

that are not yet available at full resolution. At the beginning, this low-resolution preview reconstruction volume data is exclusively used for rendering. As a detailed sub-volume block has been reconstructed on a back-end node, it is rendered on the same node and included in the final rendering on the front-end node as a kind of pre-integrated sub-volume block using our hybrid raycasting and compositing approach. Additionally, the sub-volume is queued for region segmentation on the multicore CPU of the respective node. The segmentation module is a fully automatic 3D flood-fill variant designed for distributed operation on multicore CPUs (see [Frey et al., 2009] for a a detailed discussion). It uses multiple CPU threads while reconstruction and rendering are done on the GPU concurrently. At the front-end node, the preview rendering can further be superimposed by a grid showing the sub-volume blocks that are reconstructed on the back-end nodes (Fig. 6.6), which allows the user to specify the order of reconstruction that is then communicated from the front-end node to the back-end nodes. It displays the coarse volume and allows the precise selection of sub-volume blocks by mouse click for prioritization.

Reconstruction of Sub-volume Blocks

The Feldkamp cone beam reconstruction algorithm works for industrial CT scanners which move the source in a circular trajectory shooting rays diverging as a cone through the object of interest on a detector [Feldkamp et al., 1984; Turbell, 2001]. Several GPU implementations have been presented in recent years [Xu and Mueller, 2005; Scherl et al., 2007]. In general, the Feldkamp algorithm can be subdivided into two phases: the preparation of the projection images and their subsequent depth-weighted backprojection into the volume. The preparation consists of weighting and filtering each image with a filter kernel derived from a ramp filter. The computationally most expensive part of the reconstruction is the backprojection, on which we will concentrate in the following. It is commonly implemented by determining for each volume element which projection image value it corresponds to by projecting it along the X-ray from the source to the detector. The depth-weighted sum of the respective pixels from all projection images yields the reconstructed voxel value.

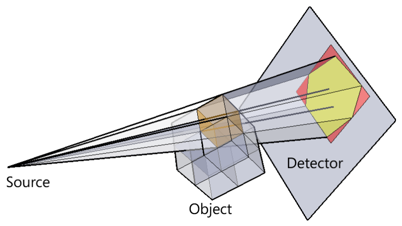


Figure 6.7: For the reconstruction of a sub-volume (orange), only a part of the original projection image is needed (yellow), and in the implementation only the smallest axis-aligned rectangle (red) containing it is loaded.

In order to completely reconstruct a group of voxels in parallel by backprojection in a single CUDA kernel pass, all required projection image data must reside in graphics memory. This can be accomplished – even for large data sets as we focus on – by only considering one sub-volume block for reconstruction at a time such that just subsets of the projection images are needed (Fig. 6.7). The dimensions of the sub-volumes are determined in a preprocessing step to cover the volume with a minimal amount of blocks considering the graphics memory available. All projection images are cropped and stored in a single container texture, similar to the storage of renderings for the front-end raycaster. Please refer to Frey et al. [2009] for further details regarding the reconstruction process.

Hybrid Multiresolution Volume Rendering Incorporating Sub-volume Blocks

Whenever requested by the front-end node, a back-end node renders images of the high-resolution sub-volume blocks it has reconstructed and segmented. The renderer only raycasts the pixels that lie within the image-space footprint of the current sub-volume with respect to the camera parameters transmitted by the head node. As the reconstruction of a subvolume can be interrupted by rendering requests, the renderer must be able to handle sub-volumes for which high-resolution images are only available up to a certain slice b. For this, the renderer substitutes the missing high-resolution slices with coarse volume data that has been reconstructed by the front-end node during the initialisation phase. In order to avoid dynamic branching on the GPU and to achieve more efficient texture fetching, the coarse volume data of resolution l is appended to the texture of high resolution h. Thus, the sampling coordinates s on the ray must be scaled to yield the texture coordinates c for the coarse volume past the boundary b in z-direction to low-resolution data: $c = (s_x \cdot \tau_x, s_y \cdot \tau_y, (s_z - b) \cdot \tau_z + b)$ with $\tau = (\frac{l_x}{h_x}, \frac{l_y}{h_y}, \frac{l_z}{h_z})$ for $s_z > b$ and $\tau = (1, 1, 1)$ otherwise (note that $b = h_z$ for a complete block).

Eventually, the renderer on the front-end node integrates the high-resolution imagery from the back-end nodes with the coarse volume data in a raycasting process (Fig. 6.8). High-resolution images are raycasted by the front-end node as a kind of pre-integrated voxels. All pre-rendered images are stored in one color texture on the graphics card, similar to the container texture of the reconstruction algorithm. Images are placed next

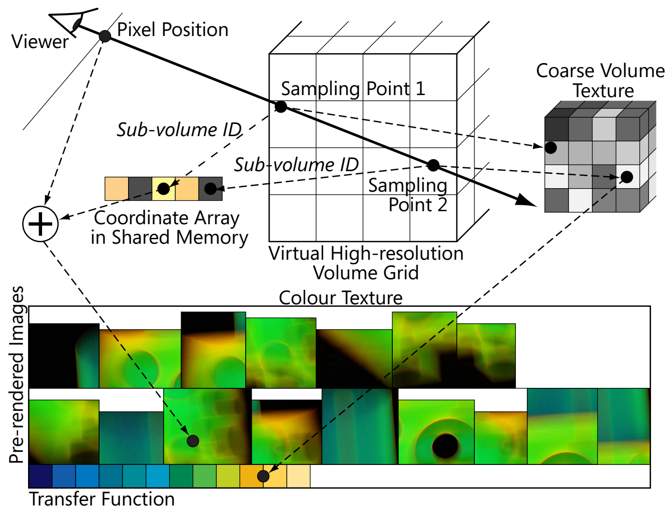


Figure 6.8: When rendering the final image, it is determined for each sampling point whether there is pre-rendered high-resolution data available (sampling point 1) or not (sampling point 2). As the case may be, different coordinates are used to access the color map.

to each other until the end of the texture is reached and then a new row of images is started at the base level of the tallest image of the previous row (Fig. 6.8).

The information on whether a high-resolution rendering for a sub-volume exists, respectively the coordinates to access it, are uploaded to the graphics card's shared memory for efficient access. Two 16 bit integers per sub-volume are utilised to determine the texture coordinates of a pre-rendered pixel for the sample point of a ray. These coordinates have already been pre-modified such that the pixel position of a ray only has to be added to fetch a pre-rendered pixel. Due to the use of shared memory and the overloading of the color map access, no actual branching is required and the amount of texture memory accesses in the sampling loop is the same as of a standard single pass raycaster: one volume texture fetch requesting a scalar density value and one color transfer texture lookup retrieving a 4D vector. Yet here the color texture lookup is also used for accessing a rendered high-resolution image, depending on the sub-volume the respective sample is in.

6.2.2 Results

We tested our system on an eight node GPU cluster with an InfiniBand interconnect. Each node was equipped with an Intel Core2 Quad CPU, 4GB of RAM, an NVIDIA GeForce 8800GTX and a commodity SATA hard-disk. One node acted as front-end creating a 256^3 voxel preview volume, while the remaining ones reconstructed a 1024^3 volume from 720 32-bit X-ray images with a resolution of 1024^2 pixels (Fig. 6.9 shows the volumes). The calculated sub-volume size was 352^3 resulting in a total of 27 sub-volumes. The time from program start until the preview volume is reconstructed on the frontend-

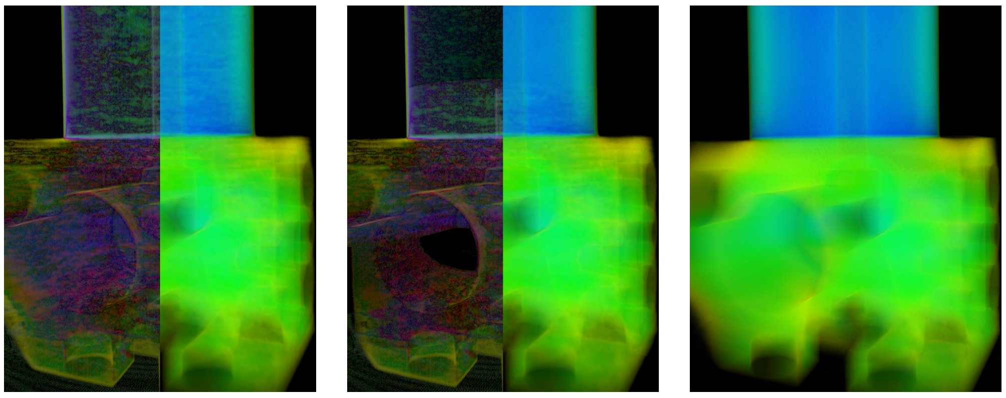


Figure 6.9: Renderings for different states of reconstruction with a coarse volume resolution of 256^3 and a high-resolution of 1024^3. The left half of figures (a) and (b) depicts the difference to (c) (level adjusted with gamma value 2.5), while the right half shows the original rendering.

node and rendered is around 29s. Most of this time is required for the I/O caused by projection image downsampling that runs in parallel with the data distribution, while the actual reconstruction on the GPU takes only 1.3s. The determination of the sub-volume dimension only takes a few seconds and runs concurrently.

Fig. 6.10 shows the data distribution and reconstruction times measured on the front-end and the back-end nodes. The times for the front-end node also include communication overhead and show the time span between program start and the availability of the complete high-resolution volume. In contrast, the numbers for the back-end nodes comprise only the longest computation. So although the average reconstruction time on the back-end nodes quickly decreases with an increasing number of nodes, the observed time on the front-end node declines more slowly, because this timing includes the input distribution and other communication overhead. The rendering times indicate the time span between the moment the front-end node requests new sub-volume images and the moment the first respectively the last remotely generated image is used in the visualization. Images from the back-end nodes can either be sent in batches or as separate messages. In our measurements, we let the batch requests—in contrast to the separate requests—interrupt the reconstruction not only after a sub-volume block but already after a sub-volume slice has been completed. For the reconstruction, this means that after rendering projection images have to be re-uploaded to the graphics card, resulting in a slightly worse rendering performance than when interrupts are prohibited. For the separate transfer of sub-images there is only little latency between the request and display of an image on the front-end node which gives the impression of a more fluid interaction, while in our test setup the time until the last image is received on the

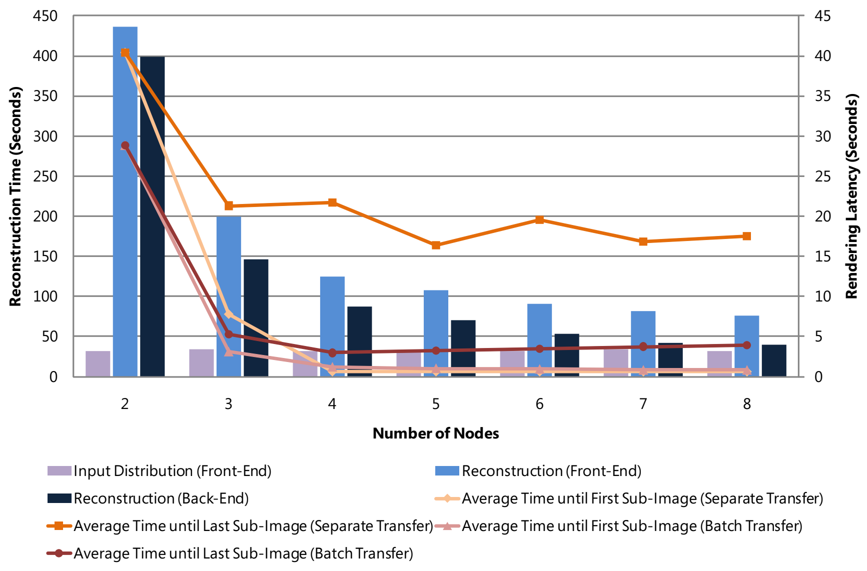


Figure 6.10: Timing results for several cluster configurations. The bar chart shows setup and reconstruction times on the front-end and back-end nodes (units on the left), while the lines show the average time from a remote rendering request until the reception of the first/last sub-volume image on the front-end node (units on the right).

head node is much longer. This is partly due to the decrease in network throughput and the potential interruption of the image transfer by higher priority messages. But more significantly, a node potentially has to wait the reconstruction time of up to 14s for a sub-volume block to be completed by the reconstruction until rendering can be started. Naturally, the likelihood of such a delay occurring on any node increases with the number of involved nodes. This hinders scaling with the cluster size of the average latency until the last image has been received. Please note, however, that the system always remains responsive as it can use the coarse volume data that is available from the beginning.

Fig. 6.11 shows the mutual influence of the reconstruction, the segmentation and user interaction during the reconstruction phase. Frequently interrupting the reconstruction by manipulating the scene increases overall reconstruction time and subsequently also segmentation time. The latter is caused by the fact that all sub-volumes have to be read for rendering from disk resulting in reduced I/O performance of the segmentation threads. The same holds true for reconstruction performance, but in this case both computations additionally conflict in the usage of the GPU.

When reconstructing a 2048^3 volume from 1440×2048^2 projection images on eight nodes, data distribution takes ~13.5min. It is limited by disk I/O as nodes need to read and write data simultaneously. The reconstruction needs ~103min, which is about 84

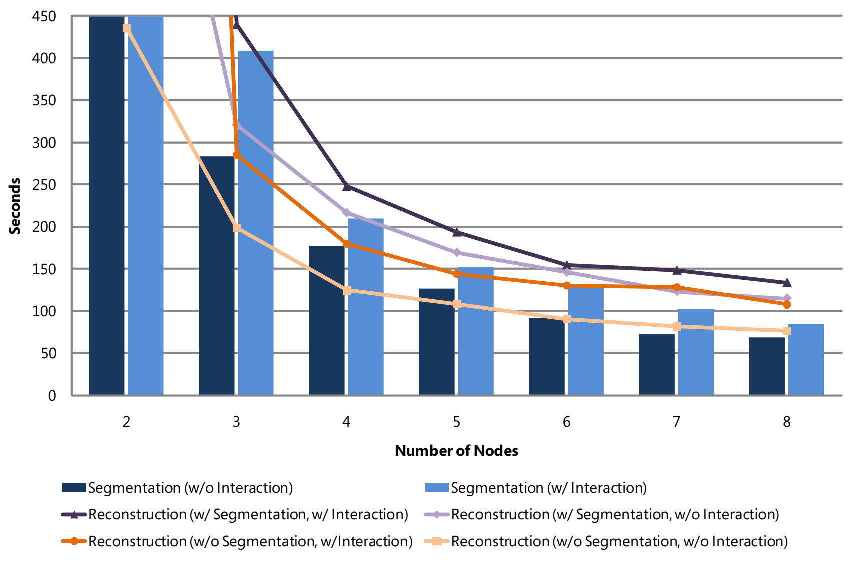


Figure 6.11: Mutual influence of reconstruction, segmentation and user interaction during the segmentation.

times longer than reconstructing the 1024^3 volume. The scaling behaviour is severely hindered by I/O performance in this case due to the excessive need for data swapping, which affects primarily the storage/access of the input data (23GB versus 4GB of RAM). Sub-volume sizes are further limited to 256^3 due to GPU memory restrictions, which requires a total of 512 sub-volumes to cover the complete volume – resulting in 19 times more accesses to projection images and sub-volumes. This significantly adds to the permanent memory pressure as the ratio of projection images and sub-volumes that can be stored in memory is already eight times worse in comparison to the 1024^3 case.

6.2.3 Directions for Further Research

The size and resolution of industrial CT scanners has further increased since the publication of this work in 2009. Accordingly, the system should be evaluated with the resulting bigger data sizes and current hardware. On top of that, a larger cluster could also be used to study the the scalability properties in more detail. Overall, the reconstruction process limited by lacking I/O performance and the capacity of the GPUs main memory. Thus, also using a fast distributed filesystem could be promising. Our architecture could be further extended by (semi-)automatically assisting the user during the sub-volume selection by pointing out regions that could be of high interest, e.g., by identifying areas with large gradients in the coarse volume. Finally, the architecture could be used to implement an interleaved data generation and visualization for other applications beyond CT reconstruction like simulations.

Chapter 7

Load-Balancing in Parallel Volume Rendering

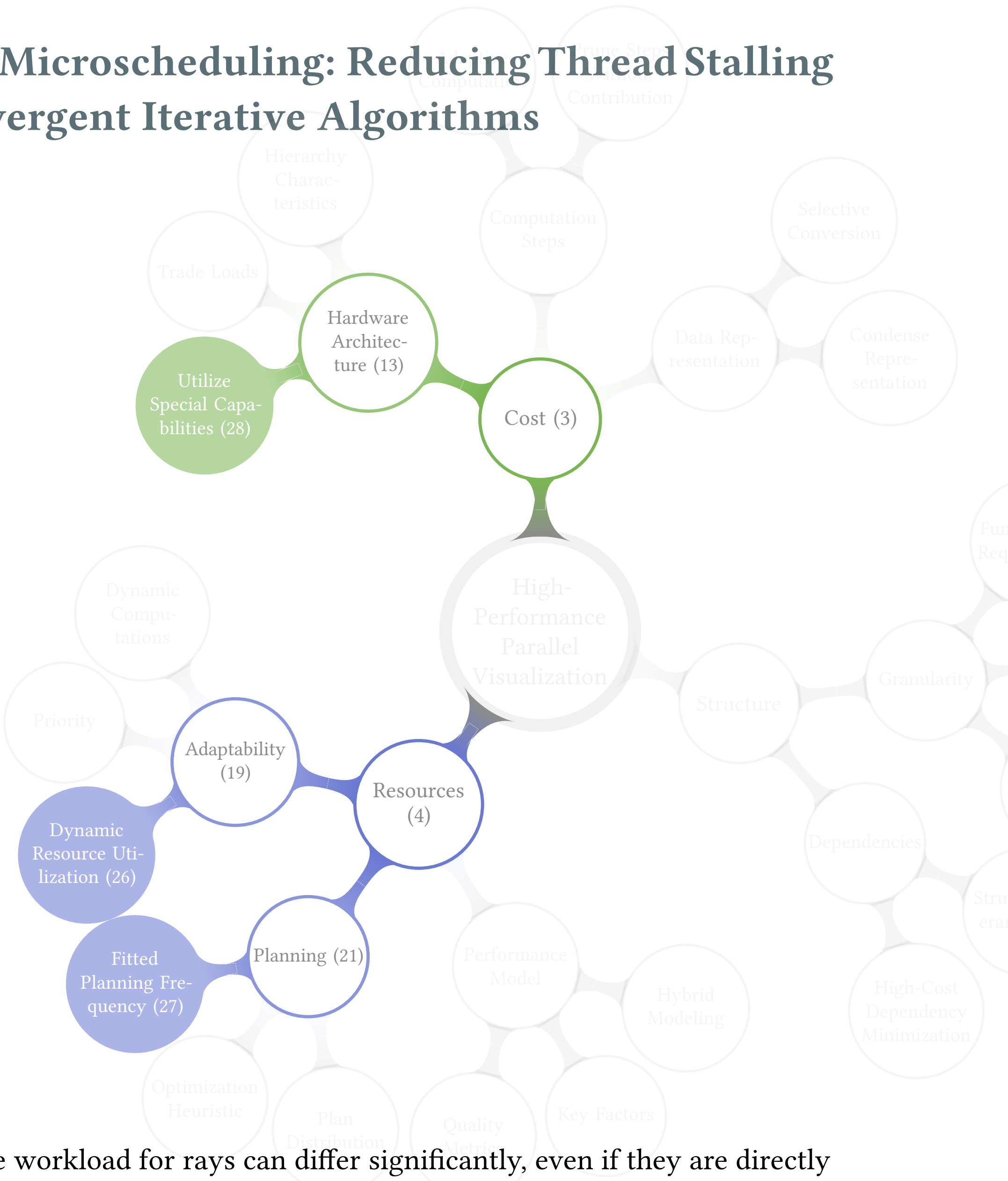
Rendering large volume data is very costly, both with respect to storage and computation. Distributing the computation over several devices is a common and powerful approach to alleviate this issue, both providing more processing and memory resources. However, significant load imbalance may be caused by the inherent heterogeneity, like the inhomogeneous distribution of costs or varying processor processor speeds. In effect, this means that resources are not utilized to their full potential.

Sec. 7.1 Alleviate the warp divergence issues on GPUs by means of an efficient work fetching scheme. . . . . [Frey et al., 2012a]

Sec. 7.2 Redundantly distributed volume data to GPUs gives the scheduler the flexibility to balance the load without inducing data transfers. . . . . . . [Frey and Ertl, 2011]

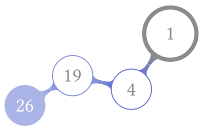
Sec. 7.3 Explicit handling of heterogeneous applications and hardware environments using graph-based algorithms with the goal to achieve the most efficient device usage. . . . . [Frey and Ertl, 2010]

7.1 SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms



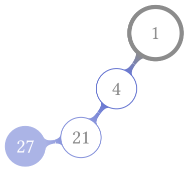
In raycasting, the workload for rays can differ significantly, even if they are directly adjacent. On GPUs, this introduces divergence issues stemming from their underlying SIMT architecture (Sec. 2.1.1). Relaxing divergence on the fly within a computation kernel by assigning new work to otherwise idle GPU threads can achieve a much higher total utilization of processing cores.

Strategy 26 ***Dynamic Resource Utilization***
Allow the flexible adaptation of resource allocation to achieve efficient device utilization, i.e., the adjustment to changing load and other factors that influence device utilization during runtime.



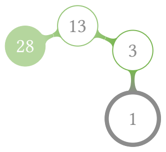
The number of new task items that are fetched during a kernel run can flexibly be adjusted—and with that implicitly how often this fetching needs to take place—to optimize for different scenarios.

Strategy 27 ***Fitted Planning Frequency***
Changes in plans or schedules can occur at different rates (corresponding to static, semi-static and dynamic scheduling, Section 2.1.1). This typically comes down to a trade-off between the overhead cost for balancing and the degree of adaptivity.



In order to avoid race conditions, task items from global task pools are fetched using atomic operations on global memory. Local task pools can either be accessed similarly using atomic operations on shared memory or using warp vote functions depending on the device capability. Exploring the different possibilities and optimizing accordingly can lead to significant speedups.

Strategy 28 ***Utilize Special Capabilities***
Utilize special hardware capabilities in order to more efficiently implement performance-critical algorithms.

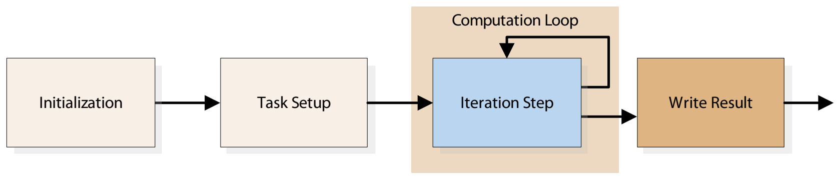


7.1.1 Tackling Iteration and Branch Divergence

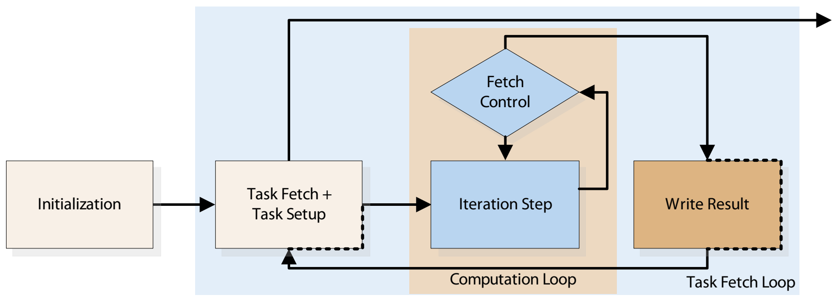
To alleviate termination divergence, an ordinary iterative application (Fig. 7.1(a)) is modified to check after an iteration step whether their associated task item is finished, and to fetch a new one if this is the case (Fig. 7.1(b)). The basic idea to avoid branch divergence and the resulting serialization of execution paths (Fig. 7.2(a)) is to choose only one (active) branch for execution and to switch task item contexts accordingly such that preferably all threads are active (Fig. 7.2(b)). The active branch is determined by the largest amount of task contexts that need to execute either of the branches. We limit ourselves to `if-then-else` statements in the following discussion without loss of generality

7.1.2 Task Pool Hierarchy

In interactive applications, a low overhead of fetching task items from a task pool during the execution of kernels is crucial. To achieve this, we design our scheme to most efficiently utilize the memory hierarchy of a GPU by employing different levels of task pools in global and shared GPU memory (Strat. 14, *Hierarchy Characteristics*). We explore different possible techniques utilizing the hierarchical memory organization of



(a) Ordinary iterative application



(b) Modified (a) for tackling termination divergence

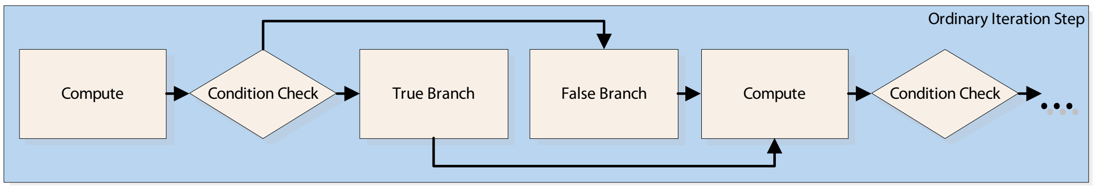
Figure 7.1: Tackling iteration divergence by altering the structure of iterative applications.

the GPU in various ways. Task pools are organized hierarchically, and distinguished in the following with respect to the group of threads which have collaborative access to it (Fig. 7.3). All threads have access to the *Global Task Pool* (in global memory), while only threads of the same thread block or warp (depending on the technique) have access to the same *Local Task Pool* (in shared memory). The *Private Task Pool* (in register space) may be used only by one thread. Task items are embodied by monotonically increasing, continuous integers. Accordingly, all task pools are represented by two counters, one for the current task and one for the last available task items. Task items are cached from the global task pool to the local or the private task pool in *chunks* to reduce costly global memory accesses, with the *chunk size* being a user-defined parameter.

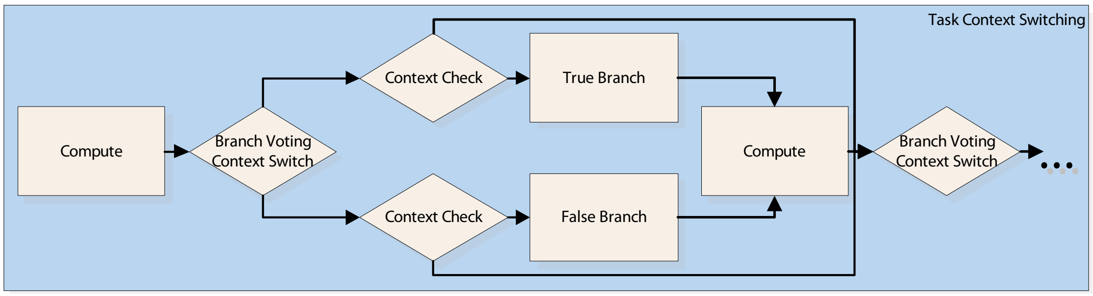
7.1.3 Task Item Fetching

While task items from global task pools are fetched using `atomicAdd()` in global memory, local task pools can either be accessed using `atomicAdd()` in shared memory or a combination of `ballot()` and `popc()` (refer to NVIDIA Corporation [2013a] for details on these functions). Three basic work distribution techniques (Local, Global and Hybrid) are introduced and evaluated in this project. They can be distinguished by the employed task pool hierarchy (Fig. 7.3). Properties of different work distribution techniques are summarized in Tab. 7.1.

Local (Fig. 7.3(a)). The local work acquisition strategy only uses local task pools with one being assigned to each thread block. Task pools are statically initialized before



(a) Standard Iteration Step



(b) Modified (a) for tackling branch divergence

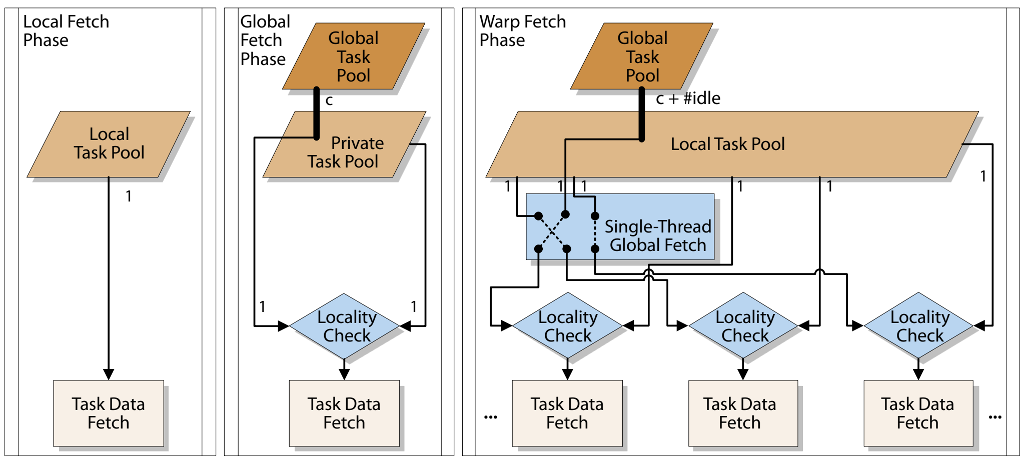
Figure 7.2: Modify the handling of branches to tackle branch divergence.

	none	local	global	hybrid	persistent
collaboration	thread	block	thread	warp	block
atomics	none	shared	global	both	global
coherency	high	high	low	medium	high
grid size	task	task	gpu	gpu	gpu

Table 7.1: Feature overview of task fetching strategies.

device kernel invocation with a certain number of task items c. On top of that, there is no transfer of other task items from or to other task pools. As the local task pool only contains a small subset of all task items, high divergence cannot be cushioned as smoothly as with a global pool, but task fetching is cheap as no operations on global memory are required.

Global (Fig. 7.3(b)). The global task fetching strategy uses one global task pool (containing all task items) and one private task pool per thread. When its private task pool is empty (*Locality Check*), a thread attempts to refill it with task items from the global task pool. The number of transferred task items depends on the user-defined chunk size c. If the private task pool is still empty after the global task fetch (because of a lack of task items in the global task pool) the thread exits the task fetch loop. Especially for small chunk sizes, this strategy allows a very fine-granular distribution of task items leading to very good iteration length divergence compensation. However, this also introduces high costs due to the large amount of global memory accesses required to refill the private task pools.



(a) Local (b) Global (c) Hybrid

Figure 7.3: Work acquisition for the local, global and hybrid strategies, with parameter c denoting the chunk size of task items that are fetched at once.

Hybrid (Fig. 7.3(c)). The hybrid strategy uses both local and global task pools. Threads which are out of work attempt to fetch new task items from the local task pool. If threads cannot get a new task due to the local task pool being empty (*Locality Check*), the thread that gets the smallest task item id beyond the valid range transfers tasks from the global to the local task pool (*Single-Thread Global Fetch*). The amount of fetched task items is determined by the number of threads which are currently out of work plus the chunk size c. The local task pool is shared by a warp, and all involved threads run in lockstep which avoids expensive thread block level synchronization calls after updating the local task pool. Initially, the local task pool is empty.

When threads exit the computation loop to fetch new task items is determined by *Fetch Control* (Fig. 7.1(b)). We experimented with the following variants:

Any Leave the loop as soon as any thread is out of work

All Only leave the computation loop when all threads are finished with their task items (similar to the persistent threads technique [Aila and Laine, 2009]).

Ballot/Atomic Exit the computation loop when a user-defined number of threads is finished. (`Ballot` uses warp voting, while atomic uses `atomicAdd` to determine the amount of idle threads).

Leaving the computation loop early (*Any*) leads to the masking of result writing and task item fetching. In contrast, exiting the computation loop late (*All*) masks iterations of the computation loop, and as a consequence does not alleviate divergence.

7.1.4 Task Item Context Switching

Branch Divergence is tackled by attempting to achieve that all threads need to take the same path, e.g., that the evaluation of the conditions of the respective task items yields the same results. This is achieved by allowing threads to switch to a fitting task item context (TIC) on the fly to minimize wasted clock cycles. We choose only one (active) branch for execution and subsequently switching TICs such that preferably all threads are active (i.e., the evaluation of the conditions of the respective task items yields the same results, and no clock cycles are wasted). The active branch is determinined by the largest amount of TICs that need to execute either of the branches (similar to [Han and Abdelrahman, 2011]). Subsequently, we increase this amount by swapping TICs. While a task item contains all information necessary for its initialization – which can be as little as one integer –, a TIC contains the complete description of the state of a computation, which is necessary for storing and resuming jobs at certain positions in the code. Branches to which we apply our technique are denoted as *managed branches* in the following.

Task Item Contexts (TICs). TICs can be attached to and detached from threads dynamically, with exactly one TIC being active for a thread at any time. TICs that are currently not attached are shared amongst threads of a warp through the *TIC pool* residing in shared memory. The *branch map* contains a list of references to the TIC pool for each managed branch and each possible condition leading to a different branch there (typically *true* and *false* for an *if*-statement). The branch map is used when switching TICs to provide the information which contexts can be loaded that fit the upcoming branch path. Besides the task item-specific information contained in a TIC, it also features an integer giving information about the current state of the context (0: Active; 1-254: Temporarily invalidated before a managed branch with the respective number; 255: Permanently invalidated). TICs are invalidated temporarily if the branch path it actually needs to execute does not match with the executed branch path. Temporarily invalidated TICs are not allowed to vote on any upcoming managed branches until the initially diverging branch is reached again in order not to corrupt code semantics. TICs are invalidated permanently when the respective task item is completed. A certain number of TICs (typically two to four times the warp size) is initialized at the very start of a kernel with distinct task items.

Task Item Context Switching. To determine the branch path with the highest saturation, votes on the upcoming branch consider both currently attached and detached TICs. Threads not agreeing with the upcoming branch path attempt to switch their current TIC with one that fits the upcoming path. The references to these detached TICs are looked up from the branch map. Whether a TIC for a certain thread is available (and which) is determined efficiently with respect to the thread id within its warp and the branch map (refer to Frey et al. [2012a] for details). In case no detached TIC is

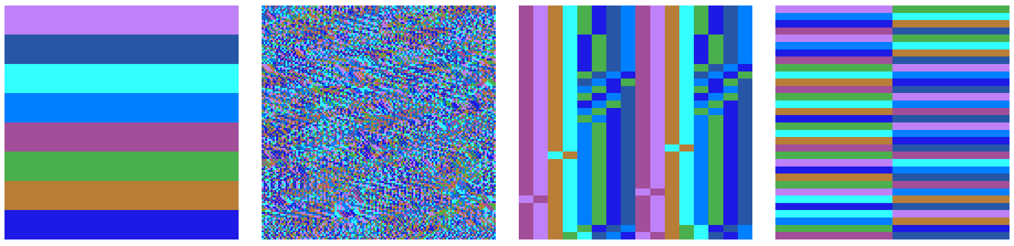

(a) Local, *Blocks* (b) Global, *Warps* (c) Hybrid, *Warps* (d) Persistent, *Blocks*

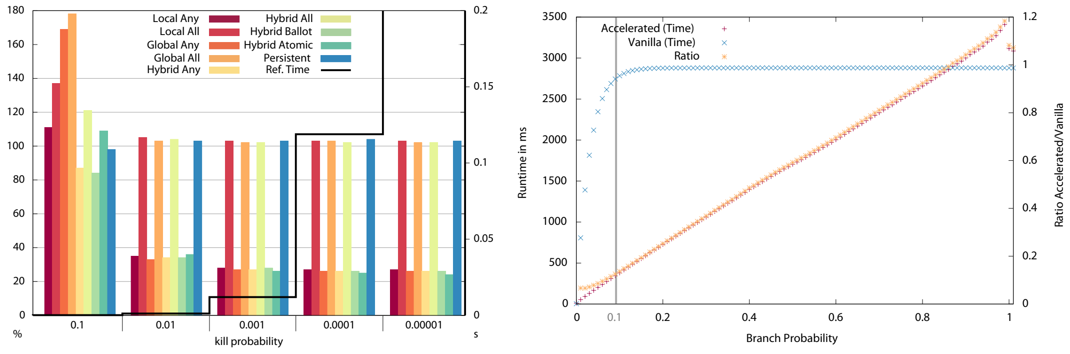
Figure 7.4: Color depicts warp or block membership of a thread per pixel of a 128x128 image with different work distribution techniques (block consists of eight warps).

available for a thread for the upcoming branch when required, the thread cannot switch its context and the current context is temporarily invalidated until this managed branch is reached again. The kernel is exited when no more contexts are referenced in the branch map and all attached contexts are completed (i.e., permanently invalid).

Deferred Context Switching. In cases in which a single iteration step is cheap, the TIC switching procedure might introduce significant overhead. *Deferred context switching* can circumvent this issue by carrying out the task switching procedure every *n*th iteration only and using a pre-defined voting outcome otherwise. TICs not complying with the pre-defined outcome are temporarily invalidated. The default vote and the *n* need to be adapted by the programmer to the problem at hand.

7.1.5 Results

Our techniques are evaluated by means of a synthetic Monte Carlo program (termination and branch divergence) and two real world applications: Fractals (termination divergence) and isosurface ray casting (branch divergence). In our evaluation, we further incorporate another variant called persistent threads by Aila and Laine [2009]. It launches just enough threads to occupy the hardware and allows thread blocks to fetch new task items from a task pool in global memory when all of its threads are idle. We concentrate on removing computational divergence only and try to avoid other effects influencing the timing results like caching in memory accesses as much as possible. All tests were run on an NVIDIA GeForce GTX580 using CUDA. Reasonable chunk sizes for tackling termination divergence were determined empirically, ranging between 256 and 12000 for *Local*, 1 and 8 for *Global* and 0 to 128 for *Hybrid*. The number of acceptable idle threads for *Ballot* and *Atomic* techniques was varied between 1 and 32. Variants of techniques resolving iteration divergence are denoted in the form [Task Item Fetching Technique] [Task Item Fetch Control Technique]. For resolving branch



(a) Monte Carlo termination divergence (b) Monte Carlo branch divergence

Figure 7.5: The Monte Carlo testing scenario for termination and branch divergence. Termination divergence results are given relative to the vanilla variant (=100) whose absolute values are depicted in seconds by the black line (right y-axis). Measurements were performed over a variety of chunk sizes, only the results for the best test case are depicted. Branch divergence was measured with varying probabilities for entering the branch.

divergence, the most beneficial amount of TICs per warp was empirically determined to be 160 for Monte Carlo and 64 for ray casting.

Task Item Fetching Properties. The distribution of task items logged from an actual run can be seen in Fig. 7.4. The global task pool approach may lead to a strong scattering of task items that are processed by threads of the same warp. This might have a negative impact if this results in scattered memory accesses (Fig. 7.4(b)). Unlike with the local strategy, the number of thread blocks which are created at the beginning of the kernel invocation is independent of the actual problem size but is chosen such that the device can be fully occupied. In the hybrid strategy, the task locality within a warp can be preserved to a large extent (Fig. 7.4(c)). Similar to the global method, the amount of thread blocks generated only depends on the targeted hardware. Only the persistent technique starts task items simultaneously for all threads of a block (Fig. 7.4(d)).

Monte Carlo. In each iteration of this generic, synthetic test case, a random number is generated using the Sobol32 generator of NVIDIA's CURAND Library. Depending on the testing scenario, this number either determines when to exit the computation loop (for termination divergence) or when a branch is entered that sums up several random numbers within an inner loop (for branch divergence). In both cases, a single iteration is cheap, but there can be significant overhead for stepping through a masked branch path (branch divergence) or having a large iteration count difference between threads of a warp respectively (termination divergence). Fig. 7.5(a) shows the results for different scenarios with increasing termination divergence through a decreasing kill probability. Techniques that do not waste computation iterations by leaving the computation loop early (*Any* and *Ballot*/*Atomic* with a low amount of idle threads)

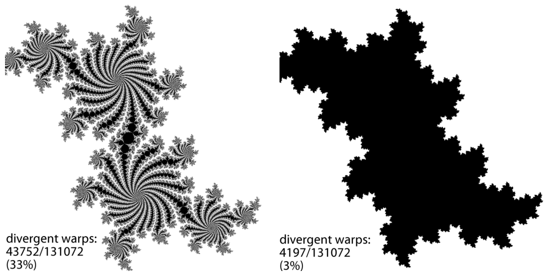


Figure 7.6: The amount of iterations (black = high) necessary for computing a pixel in the Julia set with different offsets (*left:* offset 0, *right:* offset 1). The divergent warp counts are based on a vanilla kernel with 8×4 warp tiles.

perform best with speedups up to factors of 4 to 5. In general, speedups increase with divergence. With a kill ratio of 0.1 (meaning low divergence and few iterations) it shows that task item fetch overhead can also be an important factor, especially when the computation steps are cheap like in this scenario. In particular *Global Any*—which for higher load divergences is very close to our best result—suffers from the overhead of frequently accessing the global task pool, especially because many accesses to the pool happen simultaneously. Conversely, *Hybrid Any* and *Hybrid Ballot* are even able to achieve a speedup of almost 1.2 in this scenario due to their small overhead but nonetheless good scheduling quality. Techniques that do not address thread divergence (*All* and *Persistent*) do not achieve any speedup.

Fig. 7.5(b) shows the performance of the branch divergence test case with increasing probability to enter the computation branch. Both the loop surrounding and the loop inside of the branch run 1000 iterations. The effect of branch serialization and the resulting negative performance impact show clearly in the vanilla case: there is no performance difference between one thread entering the branch or all threads entering the branch. It can be expected that at least one thread enters the branch in each iteration for a branch probability of ≈ 0.1. With our technique, in contrast, the runtime almost linearly scales with the actual work that needs to be done. Speedups of up to 15 were achieved with our technique for low branch probabilities. However, if the branch probability is higher than 84% (i.e. the efficiency loss due to divergence is low), the introduced overhead causes longer runtimes.

Fractals. The reduction of termination divergence is further evaluated at the example of Julia set fractals at a resolution of 2048×2048. Iteratively, a formula is evaluated for every pixel until it either converges or the maximum amount of iterations (specified by the *crunch factor*) is exceeded. Besides the crunch factor, we also modify the precision (single or double) and offset parameters that influence the distribution of necessary iteration counts from strongly divergent (offset 0) to constant (offset 2) (Fig. 7.6). This allows us to cover the most important execution characteristics of parallel programs: iteration cost is steered by the precision (computations with double precision take significantly longer), iteration count depends on the crunch factor and iteration count distribution is a function of the offset parameter.

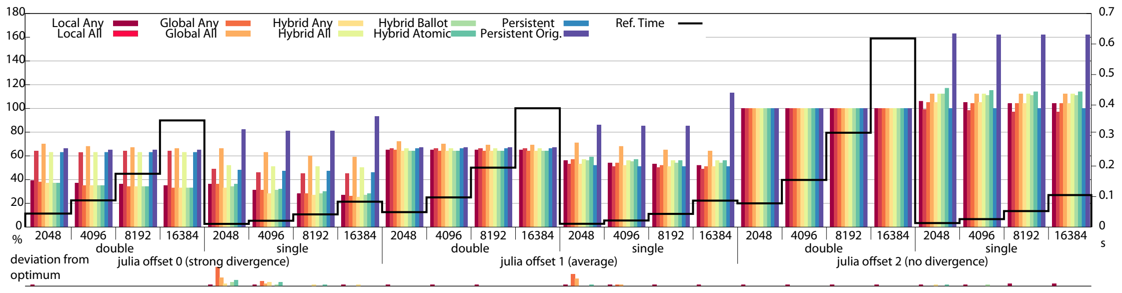
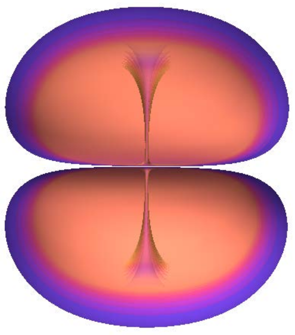


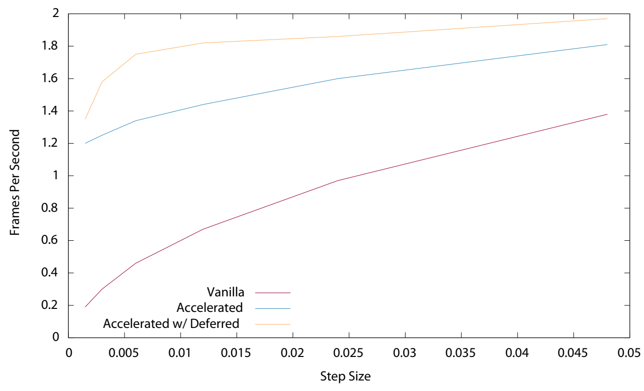
Figure 7.7: Timing results in relation to vanilla technique (=100, absolute values depicted in seconds by the black line (right y-axis)) the Julia set test scenarios. Persistent Orig. refers to the unmodified implementation of the CUDA SDK, while Persistent refers to our implementation using more thread blocks. The top bars depict the best results for a given scenario across a wide range of chunk size and idle thread settings while the bottom bars show the difference between these and the best-performing constant parameter settings across all test scenarios.

Our measurements show that fetching new task items immediately when a task is finished (Any and Ballot/Atomic with minimal idle thread threshold) is most beneficial in almost all scenarios (Fig. 7.7, *top*). When divergence is low, thread level work distribution is largely reduced to warp level work distribution (like *Persistent* and *All*) leading to similar speedups across all techniques doing task item fetching. Our *Persistent Threads* implementation in the framework and the original implementation actually perform equally well when starting them with the same amount of blocks. The difference in the diagram results from the fact that we spawn three blocks per SM, while we left the original implementation completely untouched (one block per SM). We believe that this effect is due to a better occupancy with more active warps per SM that can be used for latency hiding. Naturally, the speedup potential is higher for more divergent cases (julia offset 0 allows for higher speedups than julia offset 1). In the constant case (julia offset 2), there are no speedups since no cycles are wasted by the vanilla variant. The difference bars (Fig. 7.7, *bottom*) show that good settings for chunk size and idle threads can be determined once and remain close-to-optimal across a wide range of scenarios. Across all our tests (including Monte Carlo), chunk sizes favoring the most flexible task distribution (at the expense of higher task item fetching costs) proved to be most beneficial (small chunk sizes for *Global* and *Hybrid* as well as large chunk sizes for the static task pool of *Local*). Similarly, the best idle thread granularity setting is 1 across all scenarios for *Atomic* and *Ballot*, basically reducing both to *Any*.

Raycasting Isosurfaces. Finally, we study branch divergence for the example of multi-layer isosurface raycasting of scalar 3D functions. Rays are cast from the view point into the function domain. When a ray intersects one of eight isosurfaces, it determines an accurate intersection position using bisection. For rendering an isosurface, the first and second order derivative are evaluated numerically at this position using the SOBEL operator. Blinn-Phong shading is employed for lighting using the first order derivative



(a) Isosurface raycasting of $\psi_{3,2,1}$



(b) Timing results with various step sizes

Figure 7.8: Isosurface raycasting of the hydrogen atomic wave function $\psi_{3,2,1}$ with a resolution of 512×512 and on-the-fly evaluation and respective timing results with varying step sizes (sampling distances). In the rendering, each isosurface is mapped to a distinct color.

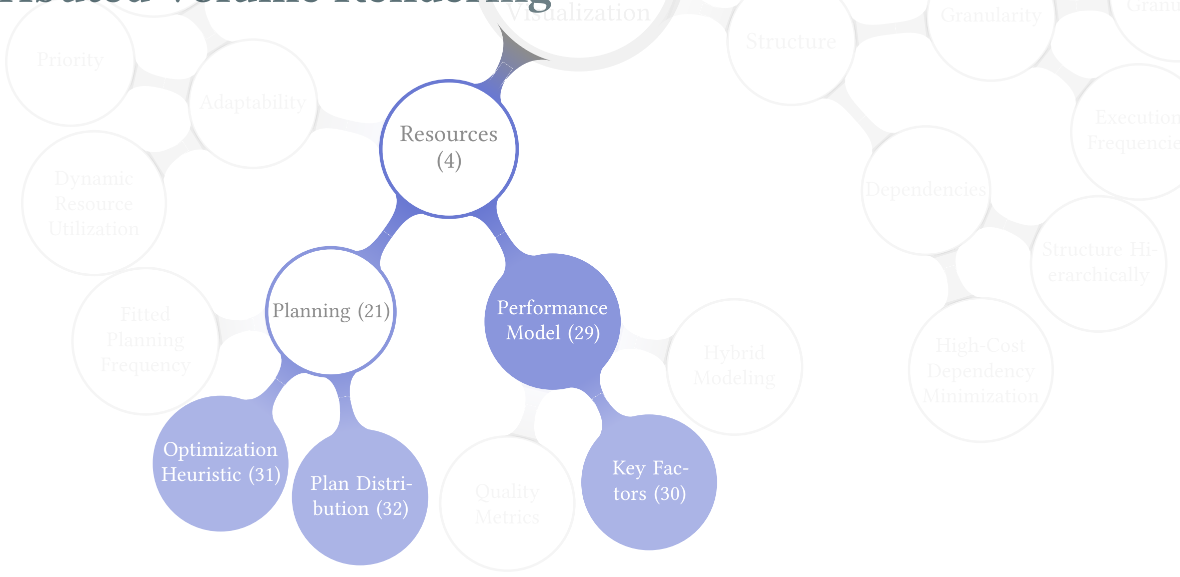
(i.e. the gradient), while the opacity of the isosurface is calculated using the magnitude of the second order derivative (Fig. 7.8(a), similar to Parker et al. [1998] and Hadwiger et al. [2005] among others). Branch divergence incurs from rays hitting isosurfaces after a different number of volume sampling iterations, which forces threads to step through the isosurface rendering procedure significantly more often than actually needed. We measured speedups of up to 7 using our proposed technique (Fig. 7.8(b)). Overall, higher performance is achieved for an increasing sampling distance (or step size), yet at the cost of lower precision. More divergence and thus higher speedups were measured with smaller step sizes because fewer rays of a warp hit an isosurface at the exact same step count. In contrast to the Monte Carlo test case, using deferred context switching (with $n = 32$) achieved speedups of up to 30% towards the standard branch divergence alleviation technique. As expected, our approach achieves higher speedups for small step sizes as isosurface hits of neighboring rays are more likely to occur in different iterations. It further shows that when there are many iterations without an isosurface hit, it is beneficial to defer the execution of the voting and switching procedure to reduce its overhead.

7.1.6 Directions for Further Research

Task item fetching can not only be used for balancing the workload but also for implementing additional features. For instance, new task items could be generated by pushing them onto a dynamic task stack at any point in the code. Elements of such a dynamic task stack could also be prioritized when executing a task item fetch to keep the task stack level as low as possible. A user could further be allowed to specify the exact order in which the task items should be executed, e.g, by means of a priority map. This would enable task items to be processed earlier if they are more relevant. For example, in combination with page-locked memory, this would allow to render and display the most important parts (e.g., the focus region) of an image first while the overall computation is still running.

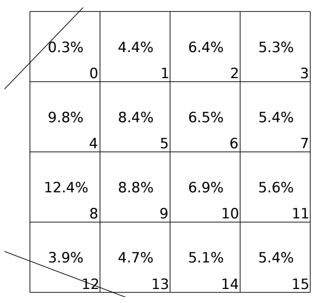
A more detailed analysis of our techniques using different types of applications could be the basis for customized improvements. In particular, closely investigating memory coherency effects could be interesting as they are strongly influenced by dynamic task fetching. Based on this, a general scheme could be developed that provides a guideline as to which applications benefit most from which technique. Based on this, the framework could also automatically determine the best approach at application startup via a series of measurements. Finally, to be applicable to a wider set of techniques, the presented approach could be extended to additionally consider dependencies between task items.

7.2 Load Balancing Utilizing Data Redundancy in Distributed Volume Rendering

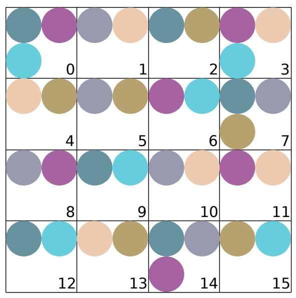


Volume data can easily exceed the storage capabilities of a single GPU. Thus, particularly for large data sets, sort-last distributed volume rendering (Sec. 2.2.2) is advantageous as not all nodes of a cluster individually need to have enough memory, but merely all nodes combined. For this, data is partitioned into volume blocks (or bricks) and distributed across different nodes. However, the rendering cost of bricks can vary significantly and change quickly with interactively changing parameters like the camera configuration (Fig. 7.9(a)). Typically, one brick is assigned to one device, and bricks are resized to adapt to the changing load. However, moving the rendering of certain volume parts from one node to another induces time consuming data transfers (Sec. 2.2.2).

In this project, to avoid the cost involved with resizing, the volume data is partitioned into many equally sized bricks. Bricks assigned to a device are widely scattered throughout the volume (Fig. 7.9(b)). This minimizes the dependency on the view parameters, as the distribution of relatively cheap and expensive bricks stays roughly the same for most camera configurations. Most importantly, bricks are saved redundantly on different compute devices to achieve evenly balanced load without any data transfers. This redundancy allows to change the work assignment without data transfers. Further, when the rendering of only one or very few bricks constitutes the major share of the overall cost, bricks can also be split logically to render them on multiple nodes (Strat. 16, *Adaptive Granularity*). As a basis for assigning bricks to devices for each frame, the render time of a brick from the previous frame is taken into account as well as the brick distribution.



(a) Percentage of ray samples per brick for a view frustum


(b) Distribution of bricks to devices (colored differently)

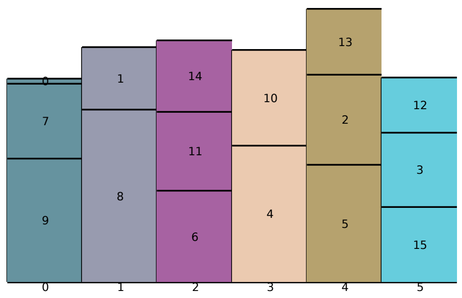
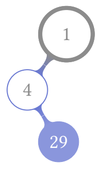
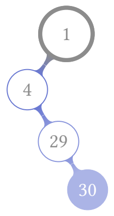

(c) Workload balanced over devices

Figure 7.9: (a) The cost of volume bricks highly depends on dynamically adjustable parameters, like camera position and orientation. (b) Distributing bricks redundantly to compute devices (c) allows our scheduler to evenly spread the load across devices by reassigning brick render task items without inducing costly data transfers.

Strategy 29 ***Performance Model***
Optimize the model determining which input values have a significant impact on the outcome, and how they can be put into relation.

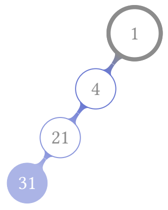


Strategy 30 ***Key Factors***
In the selection of key factors two main aspects need to be considered: expressiveness and the cost of collection. While expressiveness denotes how valuable it is to model or predict the performance behavior of the application, the cost of collection comprises the expenses required to obtain and communicate this information.



To evenly balance the load (Fig. 7.9(c)), a best-fit decreasing heuristic is employed that is commonly used to tackle the class of bin packing problems.

Strategy 31 ***Optimization Heuristic***
Typically, runtime minimization problems could be solved optimally, yet appropriate heuristics from mathematics and theoretical computer science need to be used in practice to meet time constraints.



The scheduling process is run locally on every host node with the same input data yielding the same results. This avoids transferring the schedule to all nodes, but naturally requires the timing results to be broadcast to all nodes after rendering. However,

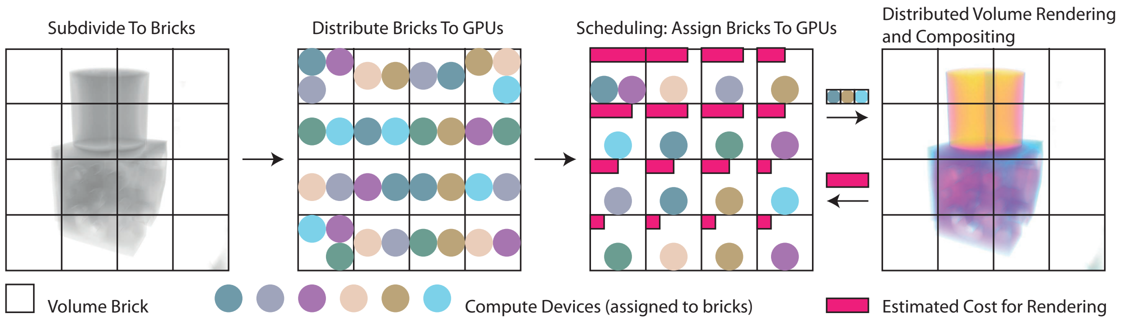
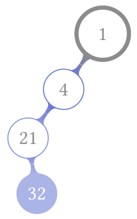

Subdivide To Bricks
Distribute Bricks To GPUs
Scheduling: Assign Bricks To GPUs
Distributed Volume Rendering and Compositing
Volume Brick
Compute Devices (assigned to bricks)
Estimated Cost for Rendering

Figure 7.10: Initially, the volume is subdivided into bricks and distributed redundantly to compute devices. For every frame, the scheduler determines which device should be used to render a specific (sub-)brick.

transmitting the timing results can be done in parallel to the rendering and compositing computations.

Strategy 32 ***Plan Distribution***
Minimize computational overhead of generating a plan and distributing it by customizing it with respect to the overall application.



7.2.1 Initialization

The brick size is influenced by several factors like device memory or device architecture and the number of devices. In particular, there must be enough work to do in parallel for good device occupancy on each graphics card, but also across all nodes in a cluster (Fig. 7.10). Distributing these bricks to devices sensibly is also important to enable good load balancing properties. For this, desirable properties are defined as follows:

1. Each device holds the maximum amount of bricks depending on its memory capacity.
2. Bricks should be distributed about the same number of times whenever possible.
3. Any two devices do not share a large amount of associated bricks.
4. The minimum distance of the bricks assigned to a device is as big as possible to achieve a wide load diversity for all camera configurations.

To achieve this, firstly, the bricks are distributed randomly, taking care of conditions 1, 2 and 3 (Fig. 7.11, Brick Distribution Optimization). Secondly, the brick distribution is

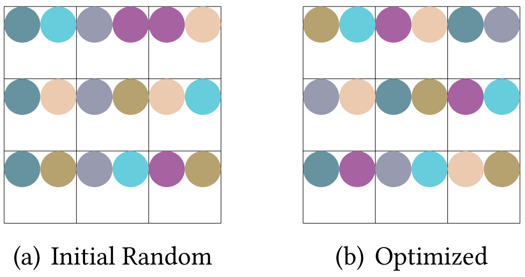


Figure 7.11: Distribution of bricks (squares) to devices (colored circles). The optimized version is reorganized to increase the euclidean distance between bricks.

optimized regarding condition 4 by swapping bricks between devices (Brick Distribution Optimization).

Brick Distribution Initialization. First of all, the list of devices and the list of bricks are shuffled randomly. Then, a brick and a device index are used to iterate over the respective lists. In every iteration, it is attempted to add the current brick to the current device. If it was successful, i.e., the brick has not already been assigned to the device previously, both indices are incremented. In case of an index reaching the end of a list, the respective list is shuffled and the index is set to the beginning of that list. If a brick cannot be added to device, the index of the brick stays the same and only the device index is incremented. A brick is deleted from the list if it could not be added to any device (anymore). Devices are erased from the device list when they reach their maximum brick capacity. The initialization is complete when either the brick list or the device list is empty.

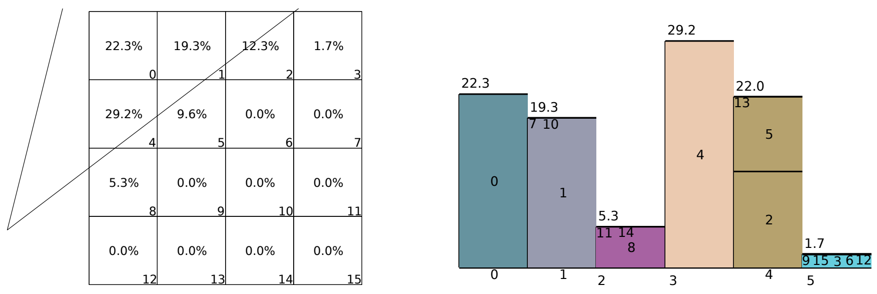
Brick Distribution Optimization. In the second step, devices swap bricks to optimize the brick distribution (the process bears some similarity to swapping points between sites for LCCVD as discussed in Sec. 3.2). For each pair of devices, the most beneficial swap of bricks occurs if it marks an improvement. The quality q_d of the distribution of bricks $b \in B_d$ for a device d is measured based on the pairwise distance between bricks as follows:

$$q_d = \sum_{0 \leq i < j < |B|} \sqrt{|b_i - b_j|}. \tag{7.1}$$

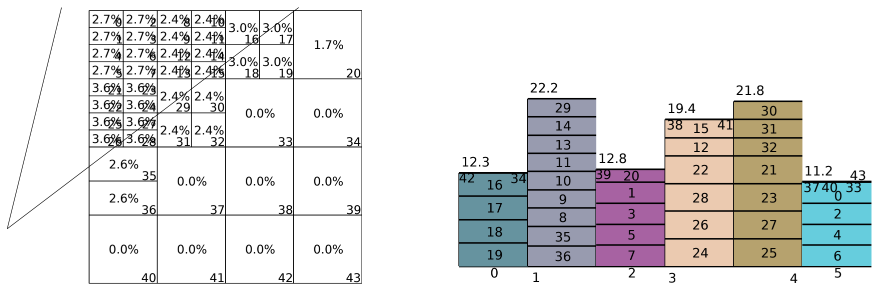
The optimization step ends when all possible device pairs have been considered without inducing a swap.

7.2.2 Load Balancing

Splitting Bricks. In the simplest case, rendering a brick translates into one job that is handled by one device. A good balancing of load between devices can be achieved that way when the major rendering load is distributed across many bricks. However, when the rendering of only one or very few bricks constitutes the major share of the overall



(a) No Job Splitting



(b) With Job Splitting

Figure 7.12: Splitting jobs improves the balancing of load across devices with certain camera configurations. The assignment of bricks to devices is taken from Fig. 7.9.

cost, the bricks need to be split to allow for more equal load distribution (Fig. 7.12). To achieve this, each brick is split recursively as long as each job's estimated rendering cost exceeds a certain value. Please refer to Frey and Ertl [2011] for details regarding the job splitting process.

Scheduling The scheduler needs to decide in every frame what jobs a device needs to render such that every brick is rendered exactly once and the maximum load occurring on any device is minimal. Formally, this is an optimization problem that is closely related to the class of packing problems. Given is a set of devices D, a set of jobs J and a cost function $c : D \times J \rightarrow \mathbb{R} \cup \infty$, with $c(d, j) = \infty$ if the brick belonging to the job $j \in J$ is not present on device $d \in D$. The goal is to find a surjective assignment function $a : D \rightarrow J$ such that $\max_{d \in D}(\sum_{j \in a(d)} c(d, j))$ is minimal. As our scheduler needs to generate only minimal overhead, we opted for a quick and simple approach that still delivers good results instead of solving this complex problem optimally. For this, we employ a best-fit decreasing heuristic that can be used for bin packing problems in general (see Fig. 7.12 for an exemplary result):

1. Sort all jobs in order of their anticipated cost in descending order.
2. Iterate through the job list and assign each job to a device such that the overall estimated execution time of all jobs of a device across all devices is minimal.

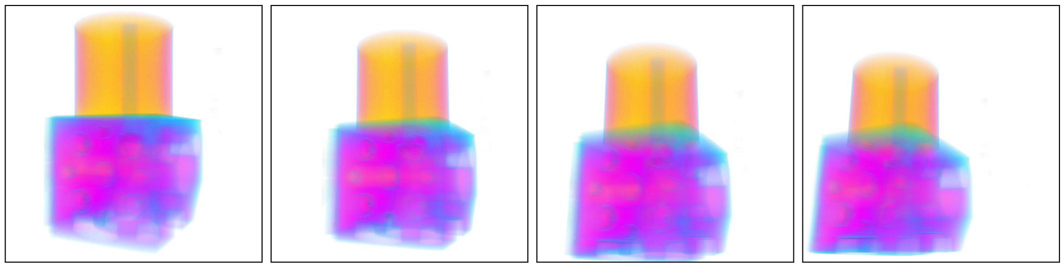


Figure 7.13: Renderings from the first four camera and focus positions in our camera path.

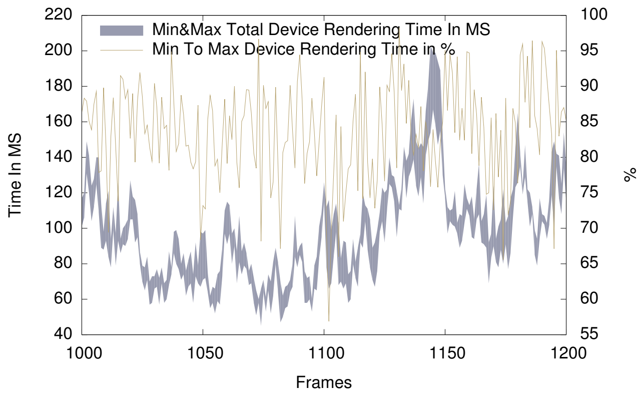


Figure 7.14: Change of the rendering times on the camera path per frame. *Min & Max Total Device Rendering Time* shows both the average minimal and maximal render time of a device along the camera path. *Min To Max Device Rendering Time* explicitly depicts the min-max ratio.

7.2.3 Results

For the evaluation of our approach, we employed a conventional volume raycaster and a simple compositer written in CUDA. We used a cluster featuring eight nodes connected via gigabit Ethernet. Each node is equipped with a NVIDIA GTX285 (featuring 1 GB of graphics memory) and a Quad-Core AMD Opteron Processor with 2.3 GHz. We further used a 1024^3 volume data set with 16 bit accuracy (i.e., a total size of 2GB) and rendered 1024^2 images. A brick size of 352^3 was used as it causes only marginal compositing overhead compared to common sort-last rendering, and still bricks can be distributed well enough to enable good load balancing. Accordingly, there is a total number of $3^3 = 27$ bricks and each device is capable of saving 10 bricks. For evaluation, we use a pre-defined camera path (Fig. 7.13, refer to [Frey and Ertl, 2011] for details). Fig. 7.14 demonstrate that with each step the rendering cost might change significantly. The magnitude of the changes in our camera path might even be less favorable in terms of frame coherency than the average interactive volume rendering session. Nevertheless, Fig. 7.15 shows that the relative differences between predictions and actual render time are within the range of a few percent. Another estimation approach would be required in low frame rate scenarios to be able to deal appropriately with rapid navigation.

Scaling. The strong scaling results depicted in Fig. 7.16 show that the maximal render

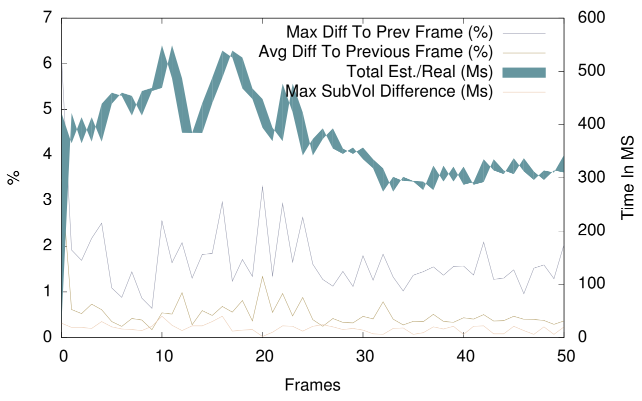


Figure 7.15: Estimated versus actual render time for the first 50 frames of our camera path. *Max Diff To Prev Frame* displays the maximum difference between any two jobs, while *Avg Diff to Previous Frame* compares the percentual difference of the total execution time over all jobs. *Total Est./Real* gives the range between estimated and measured rendering time in total. *Max SubVol Difference* depicts the maximal time difference between two jobs.

time per device significantly decreases with an increasing number of devices. Its scaling factor is even slightly larger than one. For instance, the longest time any device needs to render its bricks is 122.9 ms with four nodes and 55.5 ms with eight nodes. The reason for that is that with more devices more memory becomes available and the level of redundancy rises, which allows the scheduler to find a more optimal device-brick assignment with less load imbalance (as can be seen from the difference between the maximum and minimum time taken for rendering by a device). The increasing brick redundancy ϱ is also plotted in Fig. 7.16 by means of

$$\varrho = \sum_{d \in D} \frac{|B_d|}{|B|}, \tag{7.2}$$

with D being the set of devices, $|B|$ the number of bricks, and $|B_d|$ the number of bricks stored on a device $d \in D$.

Brick Distribution Variations. Our approach consists of a set of optimizations that can be enabled or disabled to study their benefit. It can be seen from Table 7.2 and Fig. 7.17 that the improvement of using a certain optimization depends on the amount of devices that are involved in the computation. In detail, the following variants are evaluated:

Standard The normal approach with all features enabled.
No Redundancy Each brick is assigned to only one device.
Brick Cluster Bricks belonging to a device are not distributed across the volume but concentrated in one area. This is computed by using the inverse quality function in the optimization step of the initial brick distribution (see Section 7.2.1). In

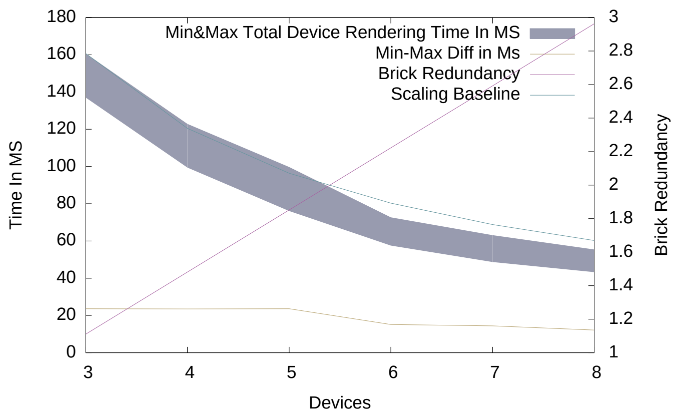


Figure 7.16: Examining strong-scaling behavior. *Min & Max Total Device Rendering Time* shows both the average minimal and maximal render time of a device along the camera path. *Min-Max Diff* explicitly depicts the difference. *Brick Redundancy* shows the increasing redundancy factor, i.e., bricks are distributed to more devices (Eq. 7.2). *Scaling Baseline* shows the baseline scaling curve based on the performance with three nodes.

Table 7.2: Averaged maximum render times (in ms) for different variants and numbers of devices with our camera path.

Variants	**8**	**4**
Standard	122.9	55.5
Brick Cluster (BC)	140.8	58.6
No Redundancy (NR)	122.6	74.6
BC & NR	141.5	85.2
No Job Split (NJ)	123.2	75.2
BC & NJ	142.2	68.2

combination with *No Redundancy*, this approximates the common one-brick-per-device strategy.

No Job Split Splitting of jobs into smaller jobs is disabled. This means that a brick translates into exactly one job.

The table shows the aforementioned effect that the technique for the standard variant scales a little better than linearly due to an increased level of redundancy. Furthermore, it can be seen that the negative effect of brick clustering is worse for few nodes, as the scheduler has less possibilities to soften the negative effects. This becomes more apparent when considering the variant with clustered bricks and no redundancy (BC& NR). While the impact is only minor for setups with few devices (as there is no high level of redundancy to begin with), the required rendering time for 8 nodes is significantly higher. This is also true to a smaller extent when considering the *No Redundancy* case only. Additionally, due to the higher amount of scheduling possibilities, disabling job splitting also has a much higher impact with large number of devices. When a brick is only distributed once or twice, there is not much room for a scheduler to widely distribute the expensive job.

General System Timings. The overall execution time of the whole application is largely dominated by the volumetric raycasting performance. Compositing the 27

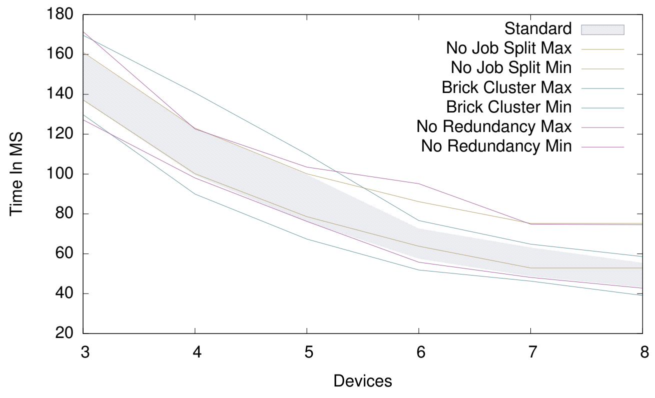


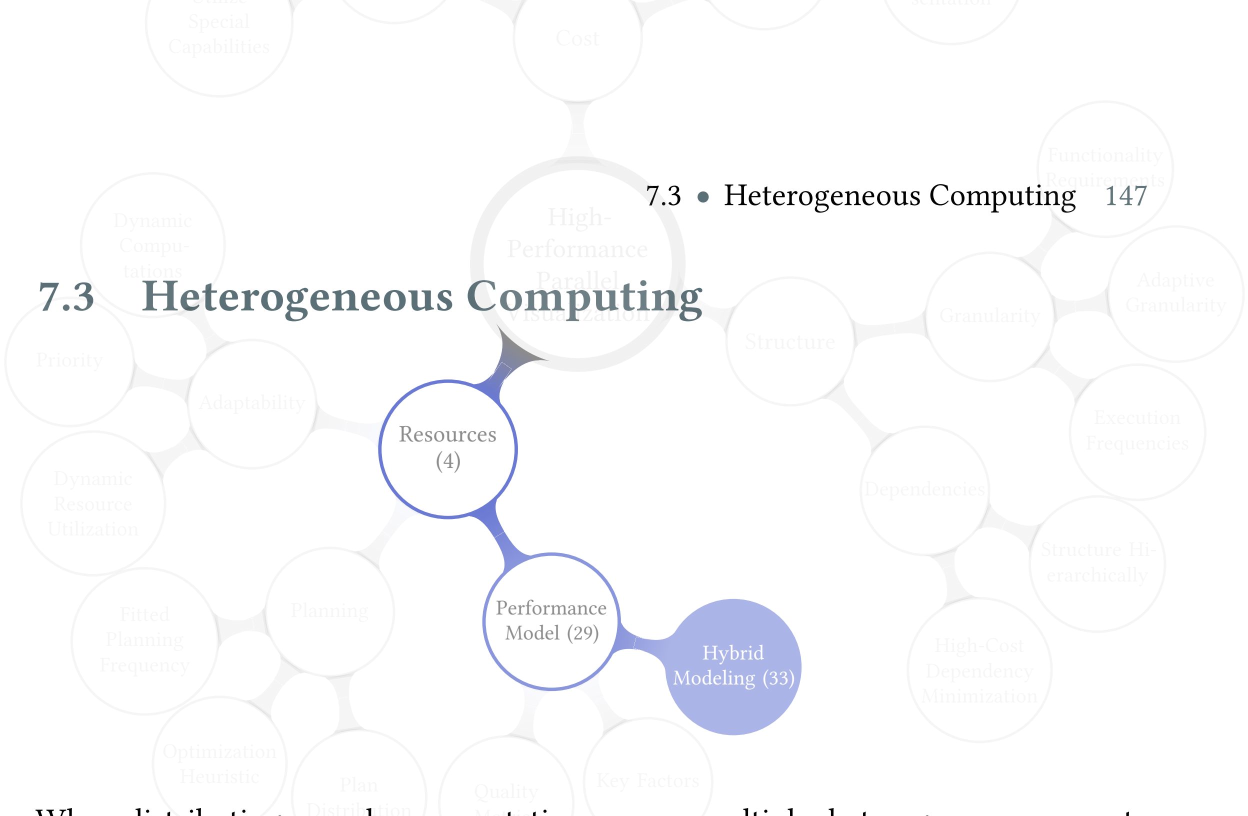
Figure 7.17: Comparison of the normal version to variants with the denoted optimization switched off (also see Tab. 7.2).

images (or more in case of job splits) takes approximately 10 ms in total with our naive compositor written in CUDA. Transferring the render image over the network is done in parallel to raycasting and does not have a significant performance impact in our testing scenario. Distributing the job render times for scheduling is a n-to-n operation (every node needs to send its job render time to all other nodes), but its size is only a few bytes and did not contribute significantly to the overall execution time either. Finally, our simple scheduler delivers very high performance and takes significantly less than a millisecond to run in our scenario, even with a large number of jobs and devices.

7.2.4 Directions for Further Research

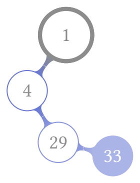
This work focuses on the volume raycasting part and largely neglects the performance impact induced by compositing. While negligible on our small cluster test system, this impact can be expected to grow significantly for larger systems. Taking compositing timing effects into consideration would be important for these systems. This would also involve considering advanced techniques from that area, e.g., Ma et al. [1993]; Makhinya et al. [2010]. Additionally, a more elaborate scheduler could be integrated, that is able to compute a closer to optimal solution, but still preserves the good complexity and execution time properties of our current scheduler. Finally, the usage of redundancy not only for the purpose of efficient load balancing but also for implementing fault-tolerance could be a promising approach.

7.3 Heterogeneous Computing



When distributing complex computations across multiple, heterogeneous compute devices, load imbalance is a major issue: the performance of executing a task on different devices or even device classes can differ significantly, and complicated dependency structures further impair the balance. The framework PaTraCo (**P**arallel **T**ransparent **C**omputation) has been introduced to support the efficient handling of this heterogeneity. As a basis, for each task, optimized implementations for different device classes may be provided by using any API. For the respective task items, required and provided data is then also specified by the programmer. This allows PaTraCo to automatically take care of the data exchange between task items. Furthermore, from this, an application graph is generated that depicts the dependencies (and their size) between all task items created for the computation. While the application graph models the system from the software side, a device graph is utilized to model the hardware. In detail, the device graph depicts the capabilities and characteristics of each device as well as their connectivity (bandwidth and latency). These abstract figures are combined with previous measurements to predict timings for computation and transfer.

Strategy 33 ***Hybrid Modeling***
Generate separate models (e.g., for hardware and software), and combine them to a hybrid model to achieve a specific prediction. This initial separation improves clarity as well as flexibility.



Using the device graph and the application graph, the scheduler determines the mapping of task items to devices based on the critical path method. In particular, since the best hardware is not necessarily the hardware that processes the problem fastest, also the cost of transferring the input data to the device as well as the availability of the device are taken into account. Determining this mapping prior to the actual computation not only enables explicitly minimizing the runtime, but also allows data to be transferred to

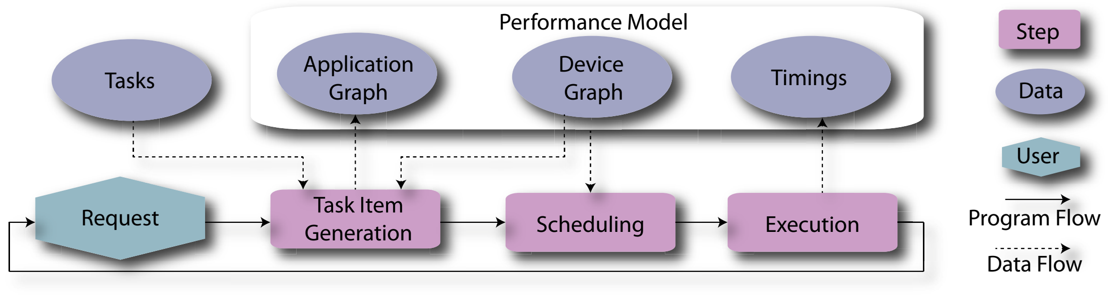


Figure 7.18: PaTraCo computation stages for an interactive application.

a compute device before the actual need for it arises (Strat. 27, *Fitted Planning Frequency*). Transferring data in parallel to computations reduces device idling times and leads to a more efficient device utilization overall (Strat. 31, *Optimization Heuristic*).

7.3.1 Overview

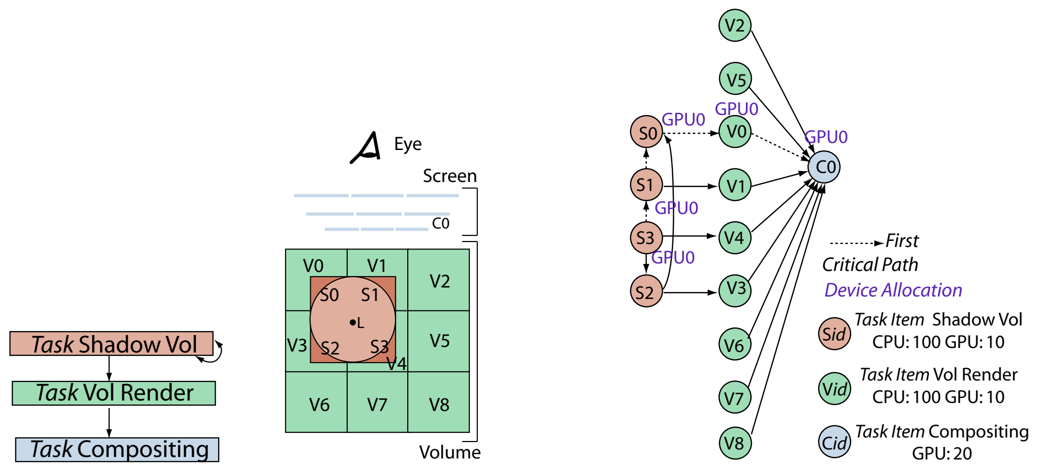
Initially, a user requests a new computation (Fig. 7.18). With respect to the parameters provided with that, task items are generated from the respective tasks. In this process, a new application graph is generated, that is part of the overall performance model, along with the device graph and previously determined timings. Using this performance model, the scheduler assigns task items to devices by employing a load-balancing heuristic based on critical paths. Finally, the determined plan is executed and timings are measured in process. Similar to Sec. 7.2, the measurements are distributed to all involved nodes, and the same scheduling procedure is executed by each of them. Task item generation and scheduling is discussed below, please refer to [Frey and Ertl, 2010] for a more detailed description of the complete procedure.

Task Item Generation. For every task, task items are generated as implemented by the programmer with respect to user-defined input data and characteristics of available compute devices. From the generated task item, an application graph is constructed. It is a directed acyclic graph (DAG) in which both the edges and the vertices are weighted (e.g., Fig. 7.19, right). The vertices stand for task items and feature one weight for each device class implementation which is available for the respective task. This weight is determined from a combination of measurements (see Figure 7.20) and task item characteristics, which may be specified by the user in task subdivision.

The application graph is constructed back-to-front from the task items that provide the result data to the task items that contain the input data. For every visited task item, edges are created to the vertices (task items) that provide required data and subsequently these are visited. Next, task items that are not required to compute the final result are removed. After that, the resulting graph is checked for cycles to ensure a seamless execution and provide feedback about potential problems.

Scheduling. The scheduler needs to assign task items to devices considering the induced compute time, the implied transfer costs, and the availability of the device. Our scheduling heuristic iteratively identifies the longest, critical and thus the most time consuming paths through the graph until all vertices (task items) are assigned to a compute device. The critical path contains those computations that, when delayed, lead to a later completion of the overall task described by all vertices in the graph. At the end of each iteration, a critical path is identified, compute devices are allocated accordingly (if the respective task item has not yet been assigned to a device in a previous iteration), and the respective edges are removed from the application graph. The sooner a task item is identified as part of a critical path of the remaining graph, the earlier it may allocate the most suitable device for a certain time span.

In each iteration, the longest (critical) path is determined by using a modified version of the Bellman-Ford algorithm [Bellman, 1958] (negative costs are used to determine the most expensive instead of the least expensive path). In addition to the original Bellman-Ford, we also consider edge weights which depend on the chosen compute device for a vertex. Initially, the distance of all vertices (task items) to the destination (final task item of the computation) is set to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. Edge and vertex weights vary with the compute devices that are assigned to the vertices. The vertex weight is determined by the product of the complexity measure of each task item (derived from measurements), general device class suitability for the given task (specified by the user), and device speed relative to other devices of its class (from the device graph). Additionally, a device might already be busy for the requested time frame. Device availability is taken into account by adding the wait time until it can be used according to its schedule. The edge weight is calculated by multiplying the transfer speed between two devices, as determined by means of the device graph and Dijkstra's algorithm, with the size of the data required to satisfy the dependencies (as specified in the application graph). Accordingly, the compute device is chosen greedily for a task item (vertex) that leads to the shortest path considering the suitability of the device for the task block, the device availability, and transfer cost from and to the device. To guarantee the finding of the shortest path, the edges must be scanned as often as there are task items. However, according to our experiments, a fraction of this suffices to achieve close to optimal results already in our context.



(a) Tasks and their dependency relation

(b) Task item generation upon request

(c) Scheduling with the generated application graph

Figure 7.19: Exemplary implementation of distributed volume raycasting featuring shadows.

7.3.2 Results

Volume Rendering Example. For evaluation, we implemented an interactive distributed volume renderer incorporating shadow volumes (illustrated in Fig. 7.19). In the following, a simplified version of the application is discussed, for the full version please refer to Frey and Ertl [2010]. The distributed volume renderer employs object-space data distribution (bricks and task items V0–V8) and uses shadow volumes for determining the illumination contribution of an omnidirectional point light source L with a limited radius (Figure 7.19(a) and (b)). This example application consists of three tasks: generation of the shadow volume, volume rendering and compositing (Fig. 7.19(a)). The first task generates a shadow volume for all volume bricks within the radius of the light source by naively sending a ray for each voxel towards the light source (task items S0–S3, for a more sophisticated algorithm see Hadwiger et al. [2006]). For all adjacent volume bricks, where a ray exits is originating brick on its way to the light source, a one voxel thin shadow volume layer needs to be available for light value contribution lookup (Fig. 7.19(b)). This results in an in-task dependency structure. The second task implements volume raycasting and employs the shadow volume for lighting each sample point (task items S0–S3). The third task combines the renderings of all bricks generated in the previous task into the final image (task item C0). For both shadow volume generation and volume rendering CPU and GPU implementations using CUDA are provided. Compositing is implemented on the GPU only using CUDA.

The effort to compute a volume rendering task item depends on the view matrix (i.e.

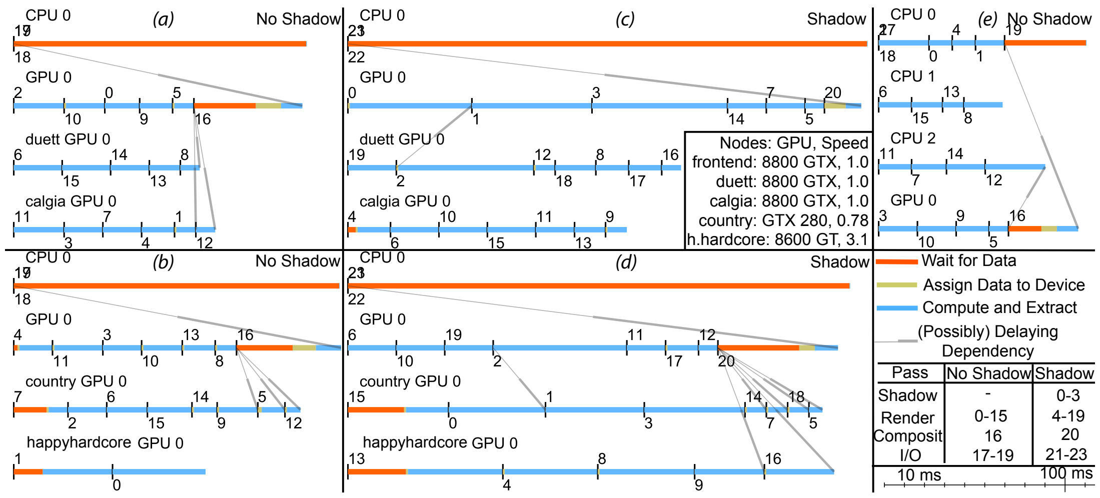


Figure 7.20: Five performance monitoring outputs generated by PaTraCo using a timeline for each device that computes at least one task item. Time lines from different nodes can be shifted to each other as computations do not start exactly at the same time. The black bars depict the beginning of a task item with its identification number. Device ids without host information belong to the node collecting the results. Task item ids can be matched with tasks using the bottom right table. `wait` (red) depicts the time until all data is available to start computing a task item, `assign` (yellow) the time to prepare the data for computation (e.g. GPU upload) and `compute` (blue) the time for task item computation and result extraction (e.g. GPU download).

camera position and orientation) and the size and position of the associated volume brick as well as the screen resolution. Using this input data, the relative cost of rendering a brick is estimated in the task item generation by computing the rendering size of the brick on the screen.

Evaluation. All nodes used in the evaluation were connected with gigabit Ethernet, and equipped with a Intel Core2 Quad 2.4 GHz CPU. They featured three different kinds of GeForce graphics cards: 8600 GT, 8800 GTX and GTX 280. In our scenario, we rendered a 1600×1024 frame using a 512^3 volume data set with 16 bit accuracy, which was split into 16 bricks by the the scheduler, so that each brick has the dimension of $128 \times 256 \times 256$. The timings in Fig. 7.20 include all steps from providing input data until the output data is available. However, they do not include scheduling, which took between $5 - 30$ ms, depending on the amount of devices given and the number of iterations specified. In our testing scenario, three iterations per critical path were already enough to achieve a good schedule. Even though CPU implementations are available for shadow volume generation and volume rendering, the computation ran on the GPU exclusively due to its vastly superior performance in this context. Using the CPU for the execution of a task item would have significantly slowed down the overall computation with our hardware setup (the CPU was slower by two orders of magnitude).

As the relative performance might substantially differ using different setups or slightly different tasks, this case shows the importance of explicitly considering device speed and not simply using an available device.

In the first series of measurements, no shadow volume is generated because we moved the light source out of the volume and gave it a tiny radius. With two identical machines each equipped with a GeForce 8800 GTX besides the frontend node (Figure 7.20 *(a)*), an even distribution of costs is achieved. In a heterogeneous setup using a node with a 8600 GT and a GTX 280 (Figure 7.20 *(b)*), the adaptation of the scheduler to different device speeds can be seen. The node featuring the 8600 GT is not assigned more than two task items, even though the device and further tasks are available, because this would result in an overall slowdown. Next, we artificially decreased the CPU cost for volume rendering artificially by sending less rays through the volume. The GPU and three CPU cores of one node were used. The results show the advantage of systems featuring many compute devices locally as opposed to distributed systems which cause a lot of network traffic, even though our framework hides transfer times by predictively copying data in parallel to ongoing computations (Figure 7.20 *(e)*).

For the second series of measurements, we moved the light source inside the volume and set its radius such that four volume bricks were covered. The primary critical path was determined by the scheduler to consist of task items with the following ids: 22 (input) – 0 (shadow) – 1 (shadow) – 3 (shadow) – 14 (render) – 20 (composit) – 23 (output) that dominates the total computation time (Figure 7.20 *(c)*). The volume rendering task items 5 and 7 are also scheduled on the frontend GPU, because they require the shadow volumes generated by the shadow tasks 0 and 1 for lighting and the transfer over the network of a shadow volume brick is too expensive. The allocation of other devices for these blocks would have substantially increased the total computation time, even though the other devices idle otherwise. When running the application with a node featuring a 8600 GT and a node featuring a GTX 280 besides the frontend node, it can be seen that the scheduler exploits the fast GTX 280 for execution of the primary critical path (Figure 7.20 *(d)*). The overall computation is not significantly faster compared to the previous case though, because the 8600 GT is significantly slower than the 8800 GTX (the system measured a factor of 3.1). The advantage of pre-copying data also manifests itself in the shadow volume measurements: task item 2 requires data from task item 1 and task item 3 requires data from task item 2, but there is no waiting time involved even though volumes are processed on different nodes.

7.3.3 Directions for Further Research

The semi-static scheduling scheme could be made more flexible during the computation by supplementing it with a dynamic work stealing mechanism. In order to enable such changes in device assignment, the node on which the originally assigned device is located needs to be informed about the changes. This node should then pass rerouting information to nodes requesting relocated data from it. Furthermore, adding the possibility to transfer precomputed device schedules can help nodes with weak CPUs by removing the scheduling load. Additionally network interconnects could be treated like compute devices, with the ability to schedule them in order to take bandwidth occupancy into account. Additionally, a graphical user interface supporting the user in implementing and connecting tasks would be very helpful in the development process. Furthermore, the scheduling algorithm could be compared to other scheduling strategies (i.e. first-come-first-serve) in a wide range of computing scenarios.

Chapter 8

Strategy Application, Discussion and Conclusion

In the previous chapters, generic strategies were extracted from the presented approaches. The collection of these strategies now forms our strategy tree (Fig. 8.1). It not only gives an overview on the explored directions of this work, but it can further be of use beyond this. In the following, we discuss the suitability of the strategy tree for the classification of existing techniques on the one hand, and the development of new techniques on the other hand. In detail, after an overview on available guidelines and taxonomies in visualization and parallel computing (Sec. 8.1), the usage of the strategy tree for the development of applications in research is outlined. Sec. 8.3 then demonstrates the utilization of the strategy tree as a taxonomy to classify research papers, conferences etc. Then, characteristics, properties and limitations of the strategy tree are discussed in Sec. 8.4. Finally, Sec. 8.5 summarizes and concludes this work.

8.1 Guidelines and Taxonomies in Visualization and Parallel Computing

Both in visualization and parallel computing, taxonomies are very commonly used to classify hardware architectures and software. Some of the most popular taxonomies have been covered already in Sec. 2, including Flynn's taxonomy for multi-processor computer architectures, programming models, different volume rendering techniques,

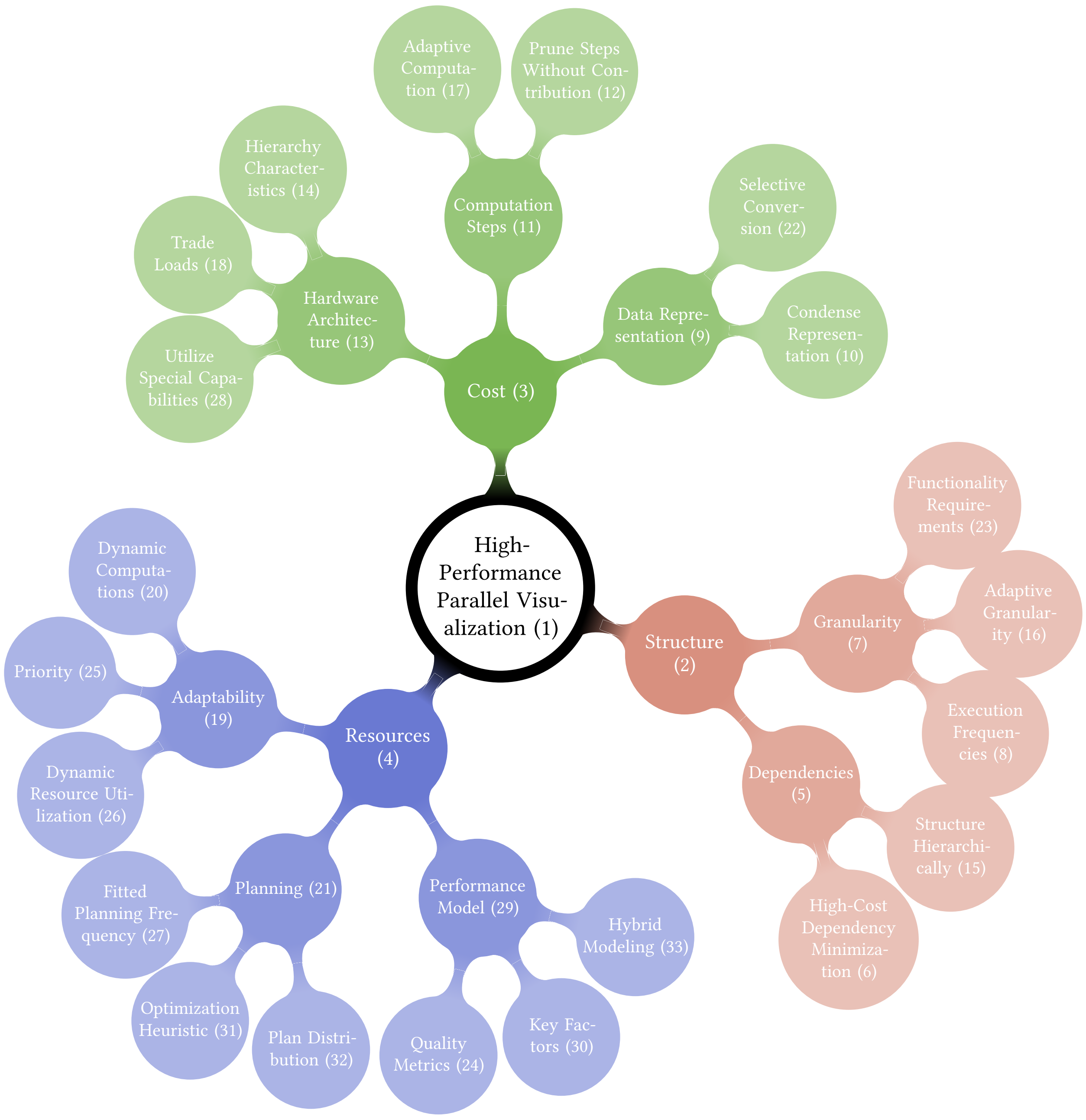

High-Performance Parallel Visualization (1)
Cost (3)
Hardware Architecture (13)
Trade Loads (18)
Hierarchy Characteristics (14)
Utilize Special Capabilities (28)
Computation Steps (11)
Adaptive Computation (17)
Prune Steps Without Contribution (12)
Data Representation (9)
Selective Conversion (22)
Condense Representation (10)
Structure (2)
Granularity (7)
Functionality Requirements (23)
Adaptive Granularity (16)
Execution Frequencies (8)
Dependencies (5)
Structure Hierarchically (15)
High-Cost Dependency Minimization (6)
Resources (4)
Adaptability (19)
Dynamic Computations (20)
Priority (25)
Dynamic Resource Utilization (26)
Planning (21)
Fitted Planning Frequency (27)
Optimization Heuristic (31)
Plan Distribution (32)
Performance Model (29)
Hybrid Modeling (33)
Key Factors (30)
Quality Metrics (24)

Figure 8.1: The strategy tree organizes the strategies that were extracted from the projects presented in this work.

or Molnar's classification of (distributed) volume rendering. Another widely used classification scheme is the ACM Computing Classification System that covers the whole field of computing [ACM, 2002]. Due to its focus on generality, it only superficially covers the area of visualization, and parallel computing and visualization are addressed seperately altogether. For generic program development, numerous parallel programming books have been published that give guidelines for certain types of parallel environments. For instance, Mattson et al. [2004] propose a pattern language for parallel programming that is organized into four design spaces, *Finding Concurrency*, *Algorithm Structure*, *Supporting Structures*, and *Implementation Mechanisms*.

In visualization, classifications are frequently given with respect to the input data, like the distinction between scientific visualization and information visualization (Card et al. [1999], among others). This allows scientists to quickly identify various techniques that can be applied to their domain of interest. However, the benefit of this prominent distinction has been subject to ongoing discussion for quite some time [Rhyne et al., 2003]. For information visualization, Chi [2000] proposes a classification based on the data state model, that can be seen as a variation of the visualization pipeline discussed in Sec. 2.2.1. Tory and Möller [2004] classify visualization algorithms with respect to the assumptions they make about the data, considering a user's conceptual model. Lumsdaine et al. [2012] present a taxonomy and conceptual framework to examine the effects of dynamically changing data on the interpretability of illustrations in information visualization. Heer and Shneiderman [2012] differentiate with respect to the kind of input data and/or the chosen visualization techniques, concentrating on the expressiveness of the results.

8.2 HPV Development with the Strategy Tree

Research typically features a high degree of uncertainty in the development process. Accordingly, development schemes featuring quick iterations with frequent evaluation of the current state have proven to be a good match for this [Highsmith, 2002]. Assess, Parallelize, Optimize, Deploy (APOD) is such a scheme introduced by NVIDIA Corporation [2013b] for the development and optimization of CUDA applications. It bases on the presumption that speedups resulting from changes can be accomplished, tested, and deployed fairly quickly. This process can be repeated by identifying further optimization opportunities, seeing additional speedups, and then deploying the even faster versions of the application into production. The scheme also contains specific advice on GPU-hardware specific optimization.

To demonstrate the use of our strategy tree for development, a modified version of APOD is discussed in the following. It is denoted as Assess, Select, Implement, Deploy (ASID), and uses the strategy tree as an integral part. It focuses on the identification

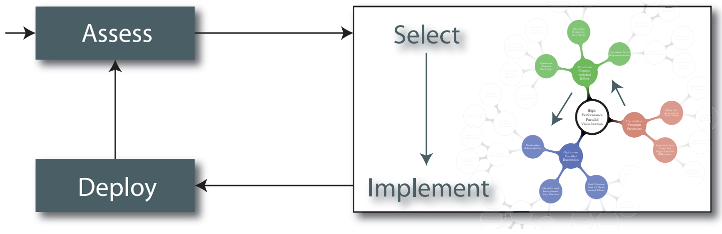


Figure 8.2: Cyclic development model ASID built around the strategy tree. It is based on the ASOP scheme by NVIDIA Corporation [2013b].

and implementation of promising strategies for HPV research (Fig. 8.2). The phases Assess and Deploy are largely carried over from APOD, yet shifted from the original industry focus to our research focus. The subsequent stages Parallelize and Optimize in APOD constitute a top-down approach, similar to [Mattson et al., 2004], with a largely domain-independent GPU focus. In more detail, the phases of our ASID model are as follows:

Assess Understand the requirements and constraints of the visualization application. Locate the parts of the application that are worthwhile for optimization, e.g., that are responsible for the bulk of the execution time.

Select Traverse the strategy tree and analyze how promising the respective strategies are for tackling the considered parts of the application.

Implement Then, exploit the selected improvement potentials by developing and applying modifications by following the identified strategy.

Deploy Thoroughly test the improved application for important use cases, and evaluate the impact of the committed changes in comparison to original expectations.

Naturally, a prerequisite for employing the scheme is a basic understanding of both the visualization and the parallel programming aspects involved. Additional knowledge, e.g., about the data set or application scenarios, can further enable the specific optimization towards certain use cases.

In the Select and Implementation phases, the multitude of options can be confusing, particularly for developers who are inexperienced in the field. As a start, according to our experience, it typically makes sense in many cases to consider strategies in their order of introduction during the course of this work, from Strat. 2 (*Structure*), over Strat. 3 (*Cost*) to Strat. 4 (*Resources*). Thus, initially, the structuring (or parallelization) of the application for performing efficiently on parallel hardware is considered (Strat. 2, *Structure*). Next, different options for reducing the cost of the selected parts of the application should be regarded (Strat. 3, *Cost*). Finally, various options for optimizing the execution on the target hardware are considered (Strat. 4, *Resources*). This sequence is also roughly employed in the related guidelines discussed above (e.g., in APOD or [Mattson et al., 2004]).

The flexibility of an iterative development process is also important as a single change of the application following a strategy can trigger other strategies to be reconsidered. For instance, the approach tackling divergence on GPUs works by load-balancing during the execution (Strat. 4, *Resources*). However, in order to make this work efficiently, an optimized implementation of the respective algorithms is required (Strat. 3, *Cost*). Note that a possible use of the strategy tree by means of ASID has merely been outlined here, and a detailed discussion and evaluation remains for future work.

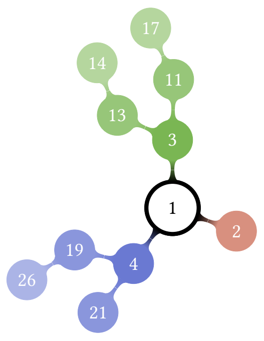
8.3 Classification With The Strategy Tree

In the following, further research projects are briefly discussed (Sec. 8.3.1). In particular, the utilized strategies are highlighted, thereby exemplifying their categorization in the context of the strategy tree (the respective strategies are graphically depicted for each project). Subsequently, we show the classification of sets of research papers published in the proceedings of different conferences (Sec. 8.3.2).

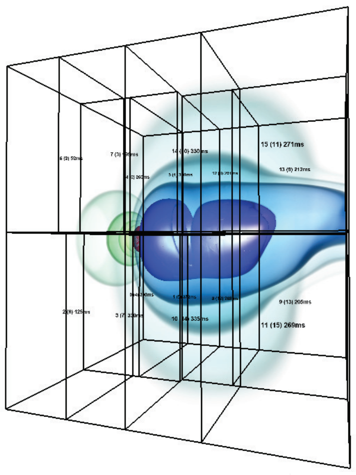
8.3.1 Further Research Projects

Interactive High-Quality Visualization of Higher-Order Finite Elements [Üffinger, Frey, and Ertl, 2010]

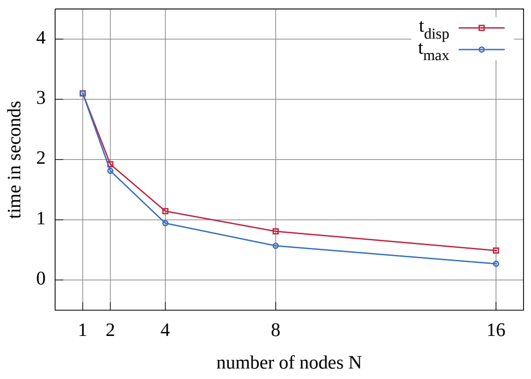
In the discussion so far, we concentrated on the widespread "lower-order" type of volume visualization. It employs trilinear interpolation on uniform grids which is directly supported by graphics hardware (Strat. 13, *Hardware Architecture*). Recently higher-order finite element methods have emerged as another important discretization scheme for simulation. It utilizes an hp-adaptive field representation. h-adaptivity means that a mesh is locally refined and can consist of different cell types. p-adaptivity denotes that a polynomial field representation is used, and that the polynomial order may differ per cell. These characteristics make the direct interactive visualization of hp-adaptive higher-order representations particularly challenging.



Different strategies need to be employed to still reach the required low response times. A high level of parallelism is achieved by both partitioning the visualization technique in image and the object space (Strat. 2, *Structure*). We use that to distribute the volume



(a) K-d tree object-space partitioning.



(b) Scaling with the number of rendering nodes.

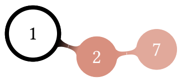
Figure 8.3: Distributed visualization of higher-order data on a GPU cluster.

on different GPUs, with each GPU tracing rays through its respective volume block in parallel. Furthermore, an adaptive evaluation scheme is employed to minimize the number of expensive polynomial evaluations (Strat. 17, *Adaptive Computation*). For this, the optimal step size is estimated per cell based on the transfer function frequency and the maximum field gradient. To lower the cost per sample, additionally a cell-barycentric monomial basis representation is used, allowing for an efficient evaluation (Strat. 9, *Data Representation*). With this basis only the coefficients need to be stored and thus the whole polynomial can be cached in shared memory for chunks of rays to reduce the amount of expensive lookups from GPU main memory (Strat. 14, *Hierarchy Characteristics*).

The cost for rendering a certain block not only depends on its size and the complexity of the cells it contains, but also strongly varies with dynamically changing parameters (this issue has already been discussed in Cha. 7, among others). This particularly involves the camera configuration as well as the transfer function (Strat. 30, *Key Factors*), and potentially causes severe load-imbalance. To allow for balancing the load, the blocks are represented by means of a k-d tree which can be adjusted dynamically (Fig. 8.3(a)) (Strat. 26, *Dynamic Resource Utilization*). This is based on fine-granular render time predictions, which are generated by distributing the measured rendering time of each block over all its cells relative to their geometric complexity and their featured number of monomials (Strat. 29, *Performance Model*). The k-d tree is finally adjusted by traversing from its root to its leaves (each leaf represents a GPU), shifting the split planes on each level to equally balance the predicted render time on both sides (Strat. 21, *Planning*). As shown by the scaling behavior in Fig. 8.3(b), this allows to achieve low response times using a small GPU cluster.

Memory Saving Fourier Transform on GPUs

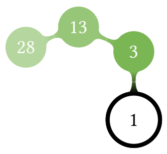
[Kauker, Sanftmann, Frey, and Ertl, 2010]



DFT is a popular method that is widely used in imaging software, among others. For accelerated computation, NVIDIA provides a library for their CUDA-enabled graphic cards. However, it requires a large buffer for storing intermediate results for the two-dimensional Discrete Fourier Transform. Here, the separability of the Fourier transform can be exploited to compute the same overall operation with a more fine-granular set of task items (Strat. 7, *Granularity*). More specifically, rows and columns are transformed independently, i.e., a 2D DFT is substituted by 1D DFT operations. This more fine-granular partitioning significantly reduces the memory requirements (thus enabling the efficient handling of larger data) at comparable processing times.

CUDA-Accelerated Continuous 2D Scatterplots

[Bachthaler, Frey, and Weiskopf, 2009]



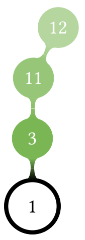
Continuous scatterplots make use of continuously defined data by drawing the scatterplot in a dense way, i.e., instead of rendering discrete glyphs, the density of the data samples is drawn in the scatterplot domain. Depending on data set size, the time to compute a scatterplot on the CPU with the original implementation could take up to several minutes—too long to efficiently use a continuous scatterplot for exploring a data set interactively. While promising some gain in performance, straight-forward porting from the CPU to the GPU, however, would lead to inefficient device utilization, partly due to branch divergence.

When considering a tetrahedral grid, the density of each tetrahedron needs to be computed to generate a continuous scatterplot. This requires a separate handling of tetrahedra based on their different projection footprints. To achieve optimal performance on the GPU, tetrahedra are sorted such that only tetrahedra of the same category are computed by a warp to avoid branch divergence issues (Strat. 28, *Utilize Special Capabilities*).

GPU-Based Two-Dimensional Flow Simulation Steering using Coherent Structures

[Ament, Frey, Sadlo, Ertl, and Weiskopf, 2011]

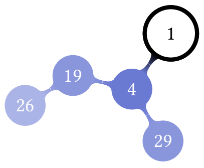
Flow simulations exhibit high computational cost and generate vast amounts of multi-attribute data. The analysis of their results usually leads to a redesign of boundary conditions and to a reparametrization of the simulation. Typically, this time-consuming process is iterated until the desired goal is achieved. Interactive investigation and manipulation (e.g., changing boundary conditions like obstacles or velocity profiles) provides a means of achieving targeted flow behavior in an intuitive way, since changes of the parametrization of the simulation are directly reflected in the visualization. In order to achieve the required fast response times, the numerous redundant computations between two frames are identified and significantly reduced in this project by reutilizing previously computed values (Strat. 12, *Prune Steps Without Contribution*).

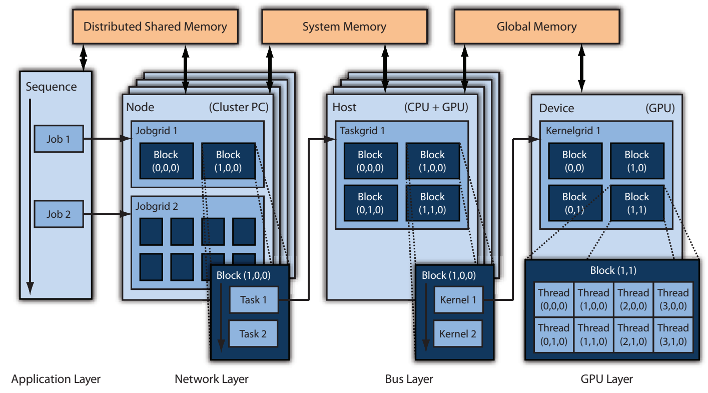


DIANA: A Device Abstraction Framework for Parallel Computations

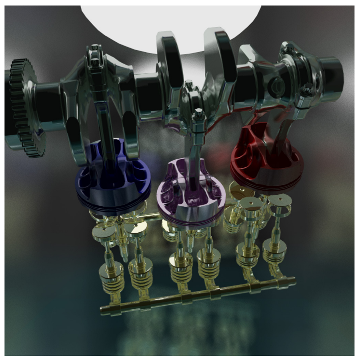
[Panagiotidis, Kauker, Frey, and Ertl, 2011]

In general, different APIs are used to program for different device classes. Porting a program to a new set of devices and/or APIs requires an adjustment or even rewrite of the existing implementation. A common interface can be used to hide the complexity of managing different APIs, SDKs, and libraries for different many-core devices. This allows for easier maintainability as well as higher flexibility and portability (Strat. 26, *Dynamic Resource Utilization*). Further, a database stores information about available devices and computations, debug messages, and profiling data (Strat. 29, *Performance Model*), which can then be used for efficient work distribution.





(a) Schematic overview of all four abstraction layers of the CUDASA programming environment. The topmost layer is placed left, with decreasing level of abstraction from left to right.

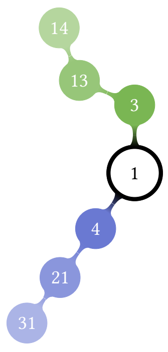


(b) Path tracing with 150 rays per pixel and a maximum of six ray bounces.

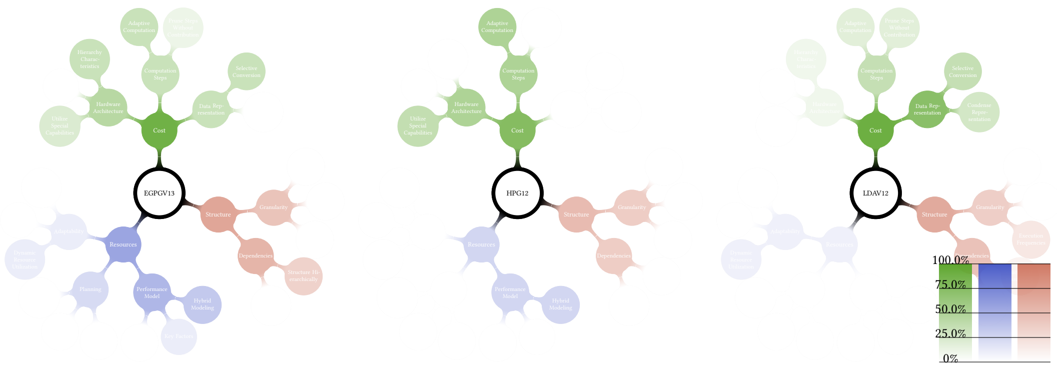
Figure 8.4: CUDASA transparently scales computations across a GPU cluster to significantly reduce the computation time.

A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality

[Müller, Frey, Strengert, Dachsbacher, and Ertl, 2009]



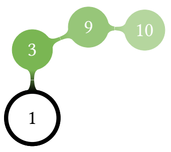
CUDASA is a development environment for distributed GPU computing (Fig. 8.4). It is based on CUDA and logically extends its parallel programming model for graphics processors to higher levels of parallelism, namely the PCI bus and network interconnects. While the extended API mimics the full function set of current graphics hardware – including the concept of global memory – on all distribution layers, the underlying communication mechanisms are handled transparently for the application developer (Strat. 14, *Hierarchy Characteristics*). However, there is potentially a lot of data that has to be shifted when the scheduling assignment changes. This cost for communication between different processing or storage entities can severely limit the overall performance of the system. To alleviate this issue, tasks and data are distributed explicitly for good data locality properties by an automatic GPU-accelerated scheduling mechanism (Strat. 31, *Optimization Heuristic*).



(a) EGPGV13 from 9 papers. (b) HPG12 from 8 papers. (c) LDAV12 from 11 papers.

Figure 8.5: Strategy classification of various conferences, with color saturation depicting the prevalence of certain strategy classes.

Space-Time Volumetric Depth Images for In-Situ-Visualization
[Fernandes, Frey, Sadlo, and Ertl, 2014]



Volumetric Depth Images (VDIs) compress the scalar data along view rays into sets of coherent supersegments (Sec. 5.1). This project introduces space-time VDIs that achieve the data reduction that is required for efficient in-situ visualization, while still maintaining spatiotemporal flexibility. For efficient space-time representation of VDI streams inter-ray and inter-frame coherence are exploited (Strat. 10, *Condense Representation*). A particular focus lies on the introduction of only small computational overhead, and an easy integration into existing simulation environments.

8.3.2 Taxonomy of Conference Papers

Research papers can be classified with respect to the strategies they employ toward HPV. Fig. 8.5 shows the condensed classification of papers that were presented at different conferences. For this, the opacity of the nodes in the strategy tree is adjusted to depict the frequency with which certain strategies were identified (in contrast to the previous visualizations of the strategy tree, the saturation of the nodes does not automatically decrease from the inside to the outside). In short, the criterion for classifying a strategy is its explicit description and discussion in the paper (beyond a brief mention). Most prominently, strategies are featured that are part of the paper's contribution. The detailed classification of individual papers is listed in Appendix A.

Fig. 8.5 lists the results for three conferences, all of which focus on high-performance visualization and rendering. Their self-descriptions are given below:

EGPGV13 (Fig. 8.5(a)) "EGPGV [focuses] on parallel graphics and visualization technology, where novel solutions exploiting and defining new trends in parallel hardware and software architectures are presented. The aim of the symposium is [to discuss] parallel and distributed visual computing and its application to all aspects of computer graphics and data visualization."
`http://www.vis.uni-stuttgart.de/egpgv/egpgv2013`

HPG12 (Fig. 8.5(b)) " High Performance Graphics [focuses on] performance-oriented graphics systems research including innovative algorithms, efficient implementations, and hardware architecture. The conference [addresses] the complex interactions of massively parallel hardware, novel programming models, efficient graphics algorithms, and novel applications."
`http://highperformancegraphics.org`

LDAV12 (Fig. 8.5(c)) "This new symposium [...] aims [...] at develop[ing] the next-generation data-intensive analysis and visualization technology. [It covers] large data management, analysis, and visualization, [and their impact on] data intensive computing and knowledge discovery "
`http://www.ldav.org`

The topical bias of these conferences reflects in their identified strategy usage, and several differences between them can be seen from Fig. 8.5. First of all, EGPGV13 exhibits the most balanced representation of all the basic strategy categories overall. For LDAV12, a particular focus lies on Strat. 9 (*Data Representation*), which can be accounted to the conference's large data focus. The partitioning of the application is of high importance, too, to be able to handle the large amounts of data efficiently (Strat. 2, *Structure*). Finally, HPG12 particularly covers the reduction of computational cost of certain rendering algorithms (Strat. 11, *Computation Steps*) as well as the customization and optimization toward certain hardware architectures (Strat. 13, *Hardware Architecture*). A particular focus lies on exploiting special functionality of certain target computation devices (Strat. 28, *Utilize Special Capabilities*).

The analysis at hand already gives a first, expressive characterization of three conferences with a similar scope. For future work, a more complete, detailed analysis with considering more specific strategies (i.e., more refined, deeper tree levels) could provide more insight. In particular, more details and a wider range of classified papers could give some indication of technological trends, and maybe even help to identify promising possibilities that are underused currently.

Considering the prevalence of strategies with respect to certain fields, a particularly interesting question for future research would be to investigate the reasons behind this. There is a couple of possible aspects that can be identified, including the following.

- Some techniques are more promising for research because they feature parts which exhibit a large room for improvement with optimized or novel algorithms.
- General trends or a high degree of familiarity by the researchers working in the field. For instance, the computer graphics and visualization community was among the pioneers of GPGPU, one of the reasons certainly being that historically they always intensively worked with graphics hardware for image synthesis.
- The degree of complexity involved in applying a strategy successfully to a field.
- Availability of tools that either have the strategy implemented already or ease its development. For instance, in visualization, this applies to techniques that are already implemented in a basic variant in frameworks like VisIt [Childs et al., 2010] or ParaView [Ahrens et al., 2005].

As already indicated above, an important question is whether some strategies are hardly considered in a certain field due to the reasons discussed above, even though they might be promising. Possibly, identifying them and looking deeper into these directions might be an interesting path to new research projects. A detailed consideration and evaluation of this remains for future work.

Here, papers were classified manually, which consumes a considerable amount of time. Automatic classification would be desirable especially when looking at large collections of papers. For future work, document classification techniques and toolkits could be used for the automatic classification of research papers on the basis of the previously classified documents (using it as training data). Intuitively, while some strategies are not trivial to detect as their description requires a higher level of understanding, others might be relatively easy to identify by means of the accumulation specific keywords.

8.4 Limitations of the Strategy Tree

So far, we have discussed the utilization of the strategy tree both in the development of new techniques and as a taxonomy for the analysis of existing approaches. There are also some limitations of the strategy tree that need to be considered:

Scope The strategy tree developed in this work is not complete, in a sense that it encompasses all strategies that may be useful in any given situation in HPV. Note however, that the strategy tree structure can easily be extended toward a more complete representation in future work.

Generality In this work, we consider projects from the field of scientific visualization, with a focus on volume visualization. As this is only a subset of the wide areas of topics covered by HPV in general, this might cause a certain bias in our determined manifestation of the strategy tree .

Optimality Due to the generality of the strategies, they might not always provide the best solution. For instance, in some cases, if there is specific knowledge about the problem (e.g., the employed hardware), there might be a quasi-optimal specialized solution that could even contradict some strategies. However, the strategies provided in this context are intended be a good starting point and provide a reasonable solution in the vast majority of cases.

Detail In this work, strategies are formulated abstractly from specific algorithms, data structures etc. In a significantly extended, more detailed (deeper) strategy tree, they would eventually appear on the leaf level, as a specializaton of the more generic concepts.

Subjectiveness The construction of taxonomies is often a highly subjective process in that different experts might reach different results [Liu et al., 2012]. Here, the strategy tree is completely derived from research done in the context of this work which inherently includes a certain bias.

8.5 Conclusion

Interactive visualization allows the viewer to explore the data in an ad-hoc fashion, which strongly supports the gain of new insights and a deeper understanding of underlying principles [Ferster, 2012]. A fundamental prerequisite for this is a very short delay in responding to user requests. This can be very difficult to achieve for large data sets and complex visualization techniques, even when employing powerful state of the art parallel hardware.

To address this, we developed novel performance-oriented visualization techniques which tackle specific issues in different areas of scientific visualization. First, the analysis of time-dependent data is addressed. It is challenging not only in terms of finding meaningful representations, but also in the context of efficient data processing. Sec. 3 introduced techniques for the the analysis of time-dependent field and particle data, with a particular focus on structuring the application for efficient parallel computation. Second, while raycasting for volume visualization has some favorable properties, like great flexibility and good parallelization characteristics, it also computationally expensive. Sec. 4 introduced techniques improving the performance of interactive volumetric raycasting on GPUs. Third, transferring data at large scale for post-processing visualization and analysis is not feasable in many situations, and storage and network bandwidth constraints more and more become the limiting factor for overall compute performance. Sec. 5 discussed view-dependent representations for volume rendering as a way to both reduce the data size of the representation and accelerate rendering. Sec. 6 further introduces remote and in-situ visualization systems that are designed to process the data as local as possible. Fourth, in parallel volume visualization, significant load imbalance may be caused by the inherent heterogeneity, like the inhomogeneous distribution of costs or varying processor processor speeds. Sec. 7 concentrates on techniques to balance this load. Overall, the discussed techniques and projects share the general objective of supporting or even enabling the efficient exploration and analysis in scientific visualization.

From these projects, generic strategies toward HPV were extracted. In scope, they range from the parallelization of the program structure and the cost optimization of the visualization procedure to the efficient execution on parallel hardware. These were structured in the newly introduced strategy tree, a consistent and comprehensive hierarchical classification of the employed strategies in the context of this work (Sec. 8). The potential use of this strategy tree was discussed for supporting the development process in visualization research or providing classification of published papers or conferences in the field. On the one hand, during the development of a visualization application in research, it could help in identifying and exploiting potentials for improving performance (Sec. 8.2). On the other hand, the use of the strategy tree as an expressive taxonomy for research on HPV has been exemplified in Sec. 8.3 by illustrating the different foci of conferences with a similar scope.

The work presented in this thesis opens multiple directions for future work. For each individual project and its presented technique, a variety of possibilities for their further development have been discussed in the respective sections. Generically speaking, this particularly encompasses ideas how to further improve scaling behavior, optimize toward a specific use case, or generalize the techniques to be successfully applicable in other scenarios. Regarding the strategy tree, future work includes complementing it with possibly missing strategies from a larger set of visualization research papers as well as a adding strategies for a higher degree of detail (e.g., specific data structures for skipping computations as a specialization of Strat. 17 (*Adaptive Computation*)). Comparisons to other fields besides scientific visualization should also be promising, as identifying similarities and particularly differences might provide new perspectives and possibilities, and considering them for a longer period of time might reveal certain trends. Finally, automatic classification would be crucial for this, and while some strategies should be relatively easy to detect by the accumulation of specific keywords, others might require a higher level of understanding, hence imposing a special challenge.

Appendix A

Strategy Classification of Visualization Papers

A.1 EGPGV13

GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting	Strat. 5 (*Dependencies*), Strat. 7 (*Granularity*), Strat. 33 (*Hybrid Modeling*)
In Situ Pathtube Visualization with Explorable Images	Strat. 22 (*Selective Conversion*), Strat. 15 (*Structure Hierarchically*)
Scalable Parallel Feature Extraction and Tracking for Large Time-varying 3D Volume Data	Strat. 22 (*Selective Conversion*), Strat. 15 (*Structure Hierarchically*), Strat. 5 (*Dependencies*), Strat. 7 (*Granularity*)
Scalable Seams for Gigapixel Panoramas	Strat. 14 (*Hierarchy Characteristics*), Strat. 15 (*Structure Hierarchically*), Strat. 7 (*Granularity*), Strat. 5 (*Dependencies*)
Rendering Molecular Surfaces using Order-Independent Transparency	Strat. 28 (*Utilize Special Capabilities*)
VtkSMP: Task-based Parallel Operators for VTK Filters	Strat. 5 (*Dependencies*), Strat. 7 (*Granularity*), Strat. 17 (*Adaptive Computation*), Strat. 21 (*Planning*), Strat. 30 (*Key Factors*)
Practical parallel rendering of detailed neuron simulations	Strat. 2 (*Structure*), Strat. 22 (*Selective Conversion*), Strat. 17 (*Adaptive Computation*), Strat. 12 (*Prune Steps Without Contribution*), Strat. 33 (*Hybrid Modeling*)
Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays	Strat. 17 (*Adaptive Computation*), Strat. 14 (*Hierarchy Characteristics*), Strat. 28 (*Utilize Special Capabilities*), Strat. 33 (*Hybrid Modeling*)
Image-parallel Ray Tracing using OpenGL Interception	Strat. 14 (*Hierarchy Characteristics*), Strat. 26 (*Dynamic Resource Utilization*), Strat. 21 (*Planning*)

A.2 HPG12

Design and Novel Uses of Higher-Dimensional Rasterization	Strat. 28 (*Utilize Special Capabilities*)
Adaptive Image Space Shading for Motion and Defocus Blur	Strat. 17 (*Adaptive Computation*)
High-Quality Parallel Depth-of-Field Using Line Samples	Strat. 5 (*Dependencies*), Strat. 7 (*Granularity*), Strat. 17 (*Adaptive Computation*)
Maximizing Parallelism in the Construction of BVHs, Octrees and k-d Trees	Strat. 17 (*Adaptive Computation*), Strat. 13 (*Hardware Architecture*), Strat. 7 (*Granularity*)
kANN on the GPU with Shifted Sorting	
SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets	Strat. 33 (*Hybrid Modeling*), Strat. 5 (*Dependencies*), Strat. 28 (*Utilize Special Capabilities*)
Algorithm and VLSI Architecture for Real-Time 1080p60 Video Retargeting	Strat. 28 (*Utilize Special Capabilities*), Strat. 5 (*Dependencies*), Strat. 7 (*Granularity*), Strat. 17 (*Adaptive Computation*)
Power Efficiency for Software Algorithms running on Graphics Processors	Strat. 33 (*Hybrid Modeling*)

A.3 LDAV12

Panning and Zooming the Observable Universe with Prefix-Matching Indices and Pixel-Based Overlays	Strat. 5 (*Dependencies*), Strat. 10 (*Condense Representation*)
Interactive Exploration of Large-Scale Time-Varying Data using Dynamic Tracking Graphs	Strat. 17 (*Adaptive Computation*), Strat. 22 (*Selective Conversion*)
Interactive Transfer Function Design on Large Multiresolution Volumes	Strat. 10 (*Condense Representation*)
Query-driven Parallel Exploration of Large Datasets	Strat. 26 (*Dynamic Resource Utilization*), Strat. 9 (*Data Representation*)
Efficient Parallel Extraction of Crack-Free Isosurfaces from Adaptive Mesh Refinement (AMR) Data	Strat. 22 (*Selective Conversion*), Strat. 5 (*Dependencies*), Strat. 7 (*Granularity*)
Parallel Stream Surface Computation for Large Data Sets	Strat. 14 (*Hierarchy Characteristics*), Strat. 5 (*Dependencies*)
Salient Time Steps Selection from Large Scale Time-Varying Data Sets with Dynamic Time Warping	Strat. 5 (*Dependencies*), Strat. 12 (*Prune Steps Without Contribution*), Strat. 7 (*Granularity*)
On the Use of Graph Search Techniques for the Analysis of Extreme-Scale Combustion Simulation Data	Strat. 22 (*Selective Conversion*), Strat. 8 (*Execution Frequencies*), Strat. 12 (*Prune Steps Without Contribution*)
Visual Analysis of Massive Web Session Data	Strat. 8 (*Execution Frequencies*), Strat. 22 (*Selective Conversion*), Strat. 10 (*Condense Representation*)
Gaussian Mixture Model Based Volume Visualization	Strat. 9 (*Data Representation*), Strat. 17 (*Adaptive Computation*)
Virtual Rheoscopic Fluids for Dense, Large-Scale Fluid Flow Visualizations	Strat. 5 (*Dependencies*), Strat. 15 (*Structure Hierarchically*)

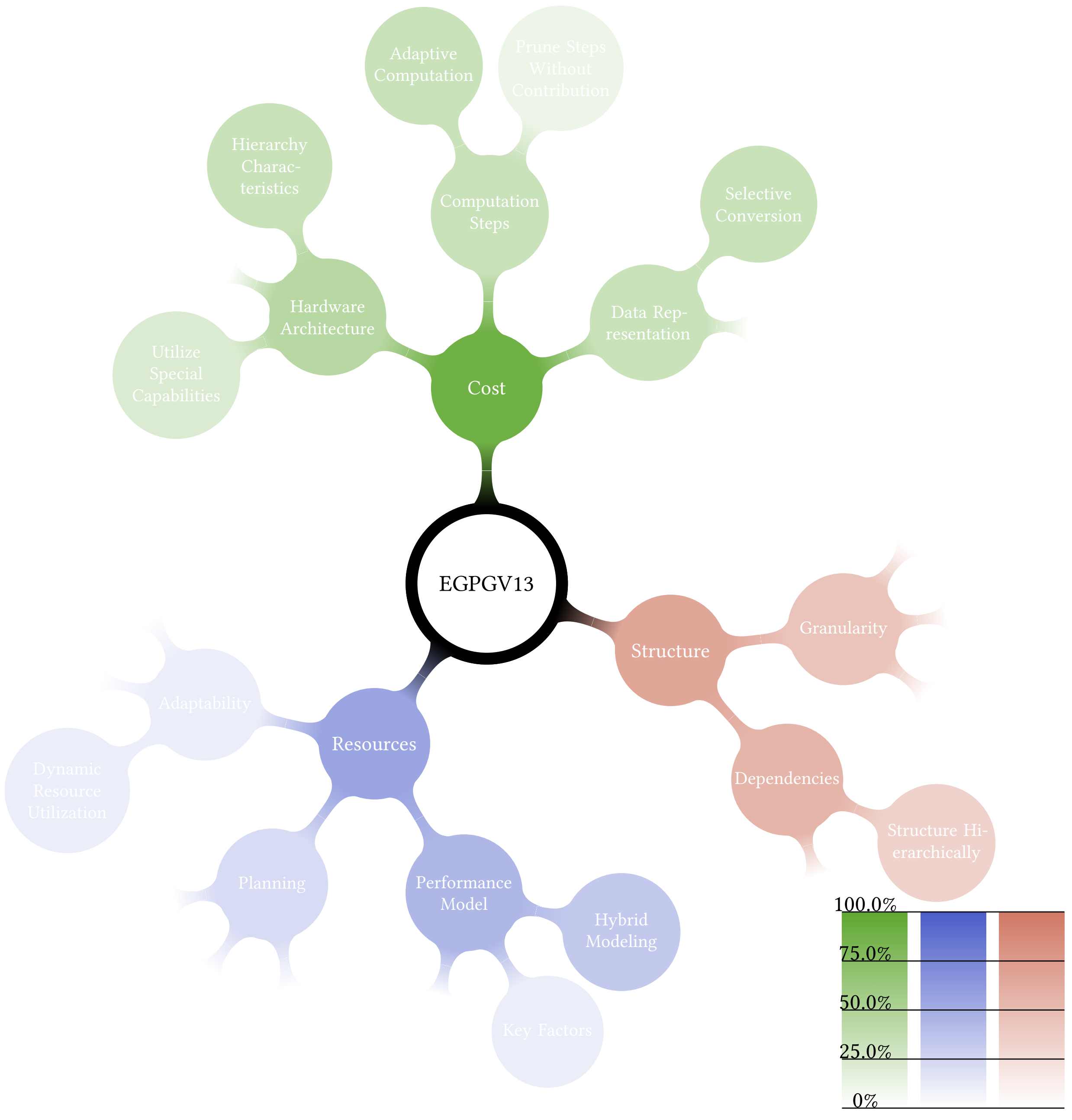


Figure A.1: Strategy classification of EGPGV13 from 9 papers.

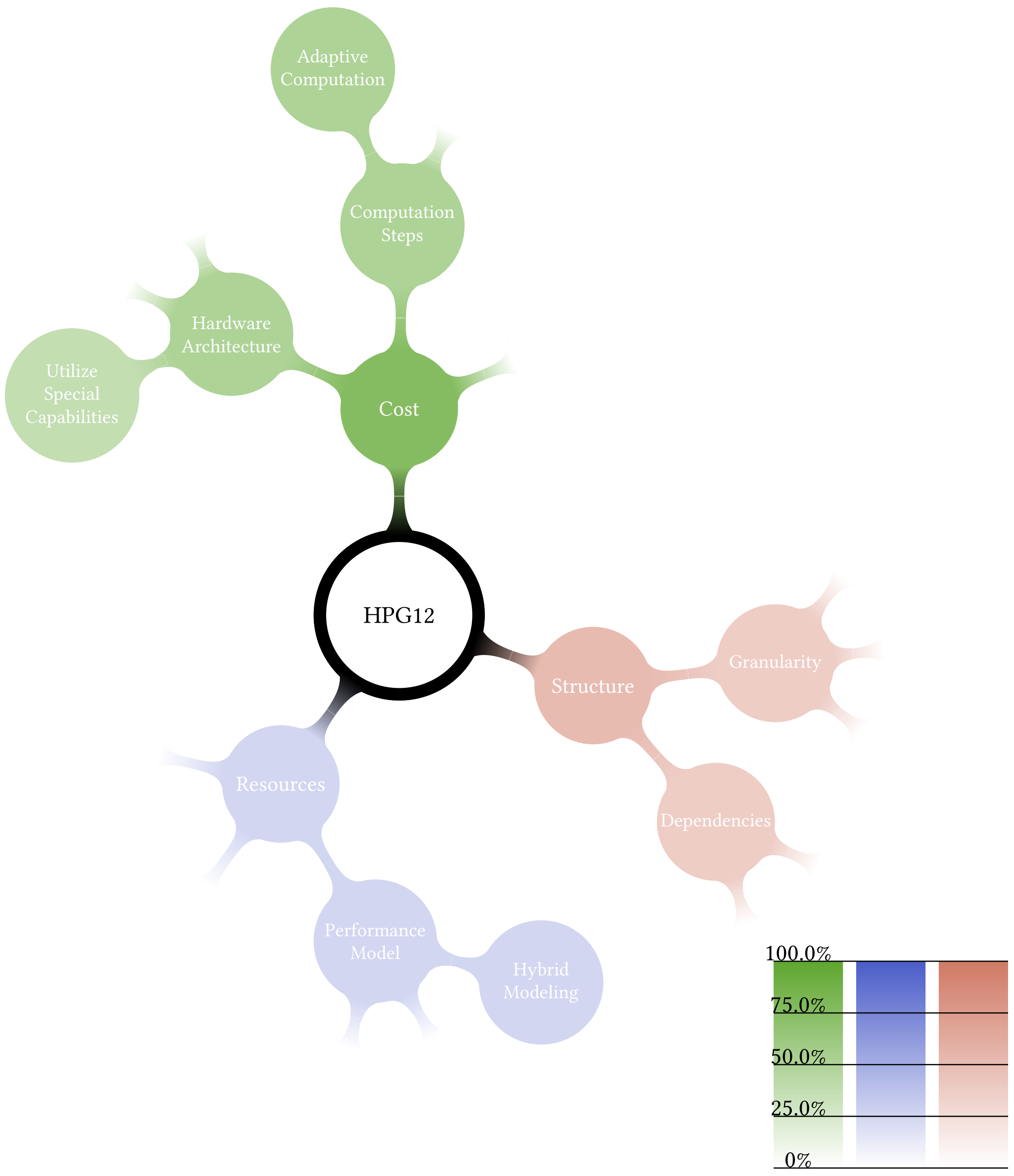


Figure A.2: Strategy classification of HPG12 from 8 papers.

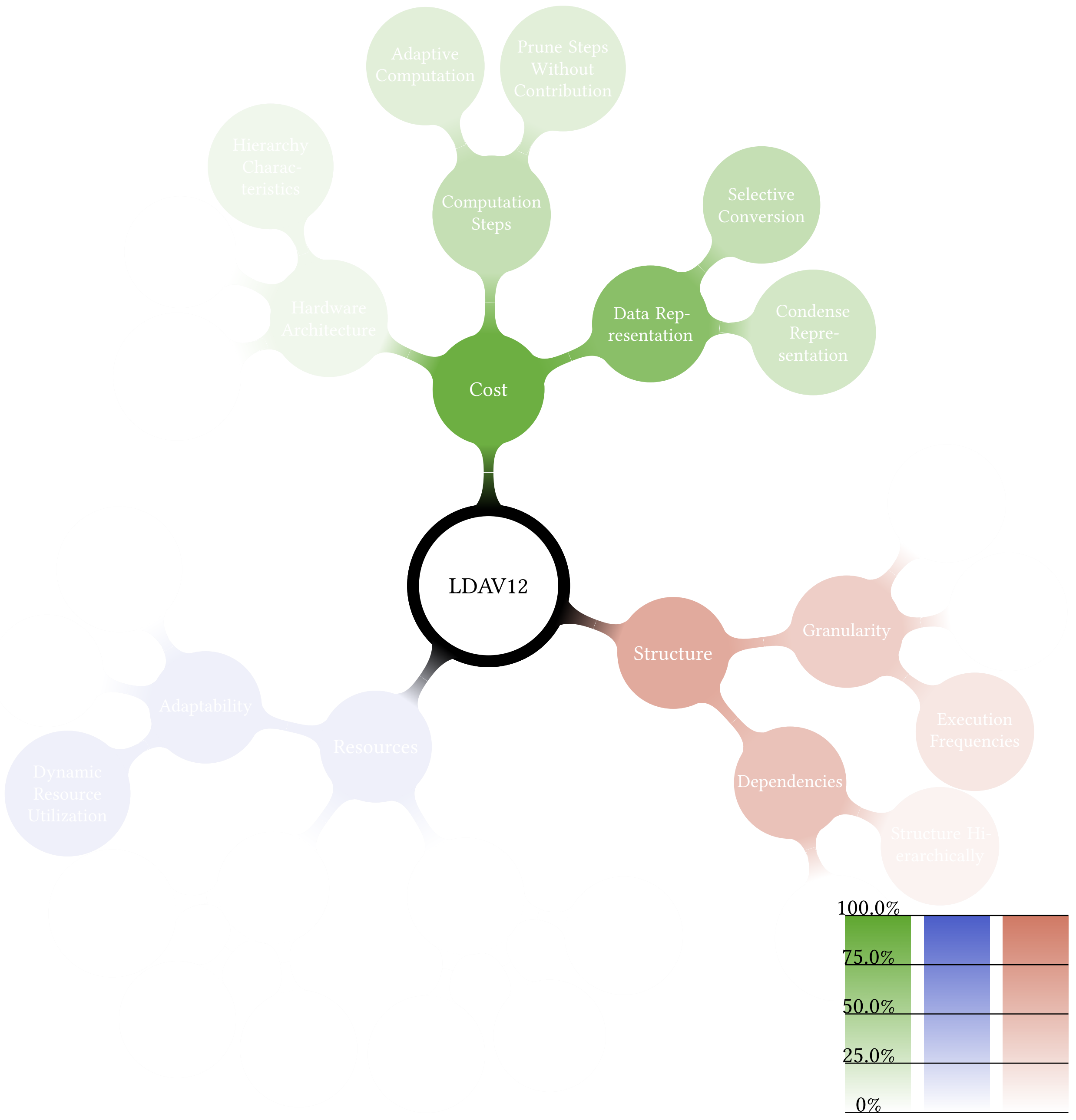


Figure A.3: Strategy classification of LDAV12 from 11 papers.

Appendix B

Volume Datasets

Name (Rendering)	Grid	Acquisition	Source
Chameleon (Fig. 4.10)	$1024 \times 1024 \times 1080$	CT	University of Texas, Austin
Ellipses (Fig. 4.4(a))	$256 \times 256 \times 256$	CT	Daimler AG
Engine (Fig. 5.3(a))	$256 \times 256 \times 256$	CT	General Electric
Flow (Fig. 5.3(k))	$2018 \times 220 \times 1085$	Simulation	Institute of Aerodynamics and Gas Dynamics, University of Stuttgart
Flow Around Sphere (Fig. 8.3(a))	Unstructured	Simulation	Institute of Aerodynamics and Gas Dynamics, University of Stuttgart
Flower (Fig. 4.5)	$1024 \times 1024 \times 1024$	CT	Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab, University of Zurich
Hydrogen Atomic (Fig. 7.8(a))	wave function $\psi_{3,2,1}$	Model	–
Jet (Fig. 4.12(a) & (b))	$720 \times 320 \times 320$	Simulation	University of Stuttgart
Vertebra (Fig. 5.3(f))	$512 \times 512 \times 512$	CT	Viatronix Inc.
Vortex (Fig. 4.12(c)–(e))	$529 \times 529 \times 529$	Simulation	Institute of Aerodynamics and Gas Dynamics, University of Stuttgart
Supernova (Cover)	$432 \times 432 \times 432$	Simulation	North Carolina State University
Toy Car (Fig. 4.2)	$559 \times 1023 \times 347$	CT	Computer Graphics Group, University of Erlangen
Zeiss (Fig. 4.4(c))	$r \times r \times r$ (varying r)	CT	Daimler AG

Bibliography

ACM. *ACM Computing Classification System.* Association for Computing Machinery, 2002. 157

N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An overview of the bluegene/l supercomputer. In *ACM/IEEE 2002 Conference on Supercomputing*, pages 60–60, 2002. 12, 19

J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization. *Visualization Handbook*, pages 717–731, 2005. 32, 166

T. Aila and T. Karras. Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of the Conference on High Performance Graphics*, pages 113–122. Eurographics Association, 2010. 21

T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*, pages 145–149, 2009. 20, 130, 132

M. Ament, S. Frey, F. Sadlo, T. Ertl, and D. Weiskopf. Gpu-based two-dimensional flow simulation steering using coherent structures. In P. Iványi and B. H. V. Topping, editors, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, United Kingdom, 2011. Civil-Comp Press, paper 18. 162

M. Ament, S. Frey, C. Müller, S. Grottel, T. Ertl, and D. Weiskopf. GPU-accelerated visualization. In E. W. Bethel, H. Childs, and C. Hansen, editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, pages 223–260. Chapman and Hall/CRC, 2012. 32

D. P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004. 21

J. Anderson, J. Bennett, and K. Joy. Marching diamonds for unstructured meshes. In *VIS 05*, pages 423 – 429, 2005. 28

C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 5704, pages 863–874, Delft, The Netherlands, Aug. 2009. Springer. 20

L. Avila. *The VTK user's guide.* Kitware, Inc, Clifton Park, New York, 2010. 24

T. O. Aydin, M. Čadík, K. Myszkowski, and H.-P. Seidel. Video quality assessment for computer graphics applications. SIGGRAPH ASIA '10, pages 161:1–161:12, New York, NY, USA, 2010. ACM. 69, 84

S. Bachthaler, S. Frey, and D. Weiskopf. CUDA-accelerated continuous 2-D scatterplots. In *Vis '09, posters*, 2009. 161

R. Baker, A. Downing, K. Finn, E. Rennison, D. D. Kim, and Y. H. Lim. Multimedia processing model for a distributed multimedia I/O system. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 164–175, London, UK, 1993. Springer-Verlag. 21

M. Balzer, T. Schlömer, and O. Deussen. Capacity-constrained point distributions: A variant of Lloyd's method. *SIGGRAPH '09*, 28(3):86:1–8, 2009. 44, 47, 52, 53, 74

A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM parallel virtual machine. Technical report, Knoxville, TN, USA, 1991. 21

R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. 149

A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5): 757–763, 1966. 14

E. W. Bethel, H. Childs, and C. Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. Chapman & Hall/CRC, 1st edition, 2012. 1

G. Bishop, H. Fuchs, L. McMillan, and E. J. S. Zagier. Frameless rendering: Double buffering considered harmful. SIGGRAPH '94, pages 175–176, New York, NY, USA, 1994. ACM. 68

M. R. Bolin and G. W. Meyer. A frequency based ray tracer. SIGGRAPH '95, pages 409–418. ACM, 1995. 30, 108

M. R. Bolin and G. W. Meyer. A perceptually based adaptive sampling algorithm. SIGGRAPH '98, pages 299–309. ACM, 1998. 30

D. Bommes, B. Lévy, N. Pietroni, E. Puppo, C. Silva, M. Tarini, and D. Zorin. State of the art in quad meshing. In *Eurographics STARS*, pages 159–182. The Eurographics Association, 2012. 28

S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011. 16

S. P. Callahan and C. T. Silva. Accelerating unstructured volume rendering with joint bilateral upsampling. *J. Graphics, GPU, & Game Tools*, 14(1):1–15, 2009. 30

S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 157

S. Cervini. European Patent EP 1531391 A2: System and Method for Efficiently Executing Single Program Multiple Data (SPMD) Programs, 2005. 21

B. Chen, A. Kaufman, and Q. Tang. Image-based rendering of surfaces from volume data. In *Symposium on Volume Graphics*, VG'01, pages 281–300. Eurographics Association, 2001. 31

E. H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, INFOVIS '00, pages 69–75, Washington, DC, USA, 2000. IEEE Computer Society. 157

H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. Weber, and E. Bethel. Extreme scaling of production visualization software on diverse architectures. *Computer Graphics and Applications, IEEE*, 30(3): 22–31, 2010. 11, 166

J.-J. Choi and Y.-G. Shin. Efficient image-based rendering of volume data. In *IEEE Computer Graphics and Applications*, pages 70–78, 1998. 31

I. Christadler and V. Weinberg. Facing the multicore-challenge. chapter RapidMind: Portability Across Architectures and Its Limitations, pages 4–15. Springer-Verlag, Berlin, Heidelberg, 2010. 20

P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998. 101

D. Cohen and Z. Sheffer. Proximity clouds - an acceleration technique for 3d grid traversal. *The Visual Computer*, 11:27–38, 1994. 29

C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM. 30

B. Curless and M. Levoy. A volumetric method for building complex models from range images. SIGGRAPH '96, pages 303–312, 1996. 28

J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of the 1992 Workshop on Volume Visualization*, VVS '92, pages 91–98, New York, NY, USA, 1992. ACM. 30

A. Dayal, C. Woolley, B. Watson, and D. Luebke. Adaptive frameless rendering. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. 68

P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 11–20, 1996. 31

K. Devine, B. Hendrickson, E. Boman, M. S. John, and C. Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proc. Intl. Conf. on Supercomputing*, pages 110–118, Santa Fe, New Mexico, 2000. 22

T. Dey and J. Levine. Delaunay meshing of isosurfaces. In *Shape Modeling and Applications*, pages 241 –250, 2007. 28

R. Diekmann. *Load Balancing Strategies for Data Parallel Applications*. PhD thesis, Universität Paderborn, 1998. 22

J. Diepstraten, M. Görke, and T. Ertl. Remote line rendering for mobile devices. In *Computer Graphics International, 2004. Proceedings*, pages 454–461, 2004. 32

C. Dietrich, C. Scheidegger, J. Schreiner, J. Comba, L. Nedel, and C. Silva. Edge transformations for improving mesh quality of marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):150 –159, 2009. 28

R. Dolbeau, Bihan, B. S., and F. H. A. hybrid multi-core parallel programming environment. A hybrid multi-core parallel programming environment. *First Workshop on General Purpose Processing on Graphics Processing Unit.*, 2007. 20

C. Donner and H. W. Jensen. Light diffusion in multi-layered translucent materials. *ACM Transactions Graph.*, 24 (3):1032–1039, 2005. 31

R. O. Duda and P. E. Hart. *Pattern classification and scene analysis*. Wiley New York, 1973. 43, 47

M. S. Ebeida, A. Patney, J. D. Owens, and E. Mestreau. Isotropic conforming refinement of quadrilateral and hexahedral meshes using two-refinement templates. *International Journal for Numerical Methods in Engineering*, 88 (10):974–985, 2011. 97, 98

H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 454–463, 2000. 37

D. Ellsworth, B. Green, C. Henze, P. Moran, and T. Sandstrom. Concurrent visualization in a production supercomputing environment. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):997–1004, Sept. 2006. 32

K. Engel, T. Ertl, P. Hastreiter, B. Tomandl, and K. Eberhardt. Combining local and remote visualization techniques for interactive volume rendering in medical applications. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 449–452. IEEE Computer Society Press, 2000. 32

K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, page 29, New York, 2004. ACM. 62

T. Erl, R. Puttini, and Z. Mahmood. *Cloud Computing: Concepts, Technology and Architecture*. Prentice Hall, 1 edition, 5 2013. 19

T. Etiene, L. G. Nonato, C. Scheidegger, J. Tierny, T. J. Peters, V. Pascucci, R. M. Kirby, and C. T. Silva. Topology verification for isosurface extraction. *IEEE Transactions on Visualization and Computer Graphics*, 18(6):952–965, 2012. 103

J.-P. Farrugia and B. Péroche. A progressive rendering algorithm using an adaptive perceptually based image metric. *Computer Graphics Forum*, 23(3):605–614, 2004. 30

L. A. Feldkamp, L. C. Davis, and J. W. Kress. Practical cone-beam algorithm. *Journal of the Optical Society of America*, 1:612–619, February 1984. 117, 118

O. Fernandes, S. Frey, F. Sadlo, and T. Ertl. Space-time volumetric depth images for in-situ-visualization. In *The 4th IEEE Symposium on Large Data Analysis and Visualization*, pages 59–65, Nov 2014. 94, 164

B. Ferster. *Interactive Visualization: Insight through Inquiry*. The MIT Press, 2012. 1, 167

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972. 8

I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. 19

S. Frank and A. Kaufman. Dependency graph approach to load balancing distributed volume visualization. *The Visual Computer*, 25(4):325–337, 2009. 32

J. Freund and K. Sloan. Accelerated volume rendering using homogeneous region encoding. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 191–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press. 30

S. Frey and T. Ertl. Accelerating Raycasting Utilizing Volume Segmentation of Industrial CT Data. In W. Tang and J. P. Collomosse, editors, *EG UK Theory and Practice of Computer Graphics, Cardiff University, United Kingdom, 2009.*, pages 33–40. Eurographics Association, 2009. 59

S. Frey and T. Ertl. PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks. In *EGPGV, Norrköping, Sweden*, pages 131–140. Eurographics Association, 2010. 125, 148, 150

S. Frey and T. Ertl. Load Balancing Utilizing Data Redundancy in Distributed Volume Rendering. In T. Kuhlen, R. Pajarola, and K. Zhou, editors, *EGPGV, Llandudno, Wales, UK, 2011. Proceedings*, pages 51–60. Eurographics Association, 2011. 125, 142, 143

S. Frey, C. Müller, M. Strengert, and T. Ertl. Concurrent CT Reconstruction and Visual Analysis Using Hybrid Multi-resolution Raycasting in a Cluster Environment. In *ISVC '09: Proceedings of the 5th International Symposium on Advances in Visual Computing*, pages 357–366, Berlin, Heidelberg, 2009. Springer-Verlag. 105, 118, 119

S. Frey, T. Schlömer, S. Grottel, C. Dachsbacher, O. Deussen, and T. Ertl. Loose Capacity-Constrained Representatives for the Qualitative Visual Analysis in Molecular Dynamics. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium*, pages 51–58, Washington, DC, USA, 2011. IEEE Computer Society. 33, 53, 54

S. Frey, G. Reina, and T. Ertl. SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms. In *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '12, pages 399–406, Washington, DC, USA, 2012a. IEEE Computer Society. 125, 131

S. Frey, F. Sadlo, and T. Ertl. Visualization of temporal similarity in field data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2023–2032, 2012b. 33, 37, 40

S. Frey, F. Sadlo, and T. Ertl. Mesh Generation From Layered Depth Images Using Isosurface Raycasting. In *ISVC '13: Proceedings of the 9th International Symposium on Advances in Visual Computing*, pages 373–383, Berlin, Heidelberg, 2013a. Springer-Verlag. 87, 96

S. Frey, F. Sadlo, and T. Ertl. Explorable volumetric depth images from raycasting. Los Alamitos, 2013b. 26th Conference on Graphics, Patterns and Images (SIBGRAPI), IEEE Computer Societys Conference Publishing Services. 87

S. Frey, F. Sadlo, K. Ma, and T. Ertl. Interactive progressive visualization with space-time error control. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2397–2406, 2014. 59

M. Friendly. Visions and Re-Visions of Charles Joseph Minard. *Journal of Educational and Behavioral Statistics*, 27 (1):31–51, 2002. 22

W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007. 21

M. N. Gamito and S. C. Maddock. Ray casting implicit fractal surfaces with reduced affine arithmetic. *The Visual Computer: International Journal of Computer Graphics*, 23(3):155–165, Feb. 2007. 29

B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous computing with OpenCL*. Morgan Kaufmann, 2011. 16

S. Ghosh. *Distributed systems : an algorithmic approach*. Chapman & Hall/CRC computer and information science series. Chapman & Hall/CRC, Boca Raton, London, New York, 2007. 12, 19

S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. SIGGRAPH '96, pages 43–54, 1996. 31

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994. 21

R. Grosso and T. Ertl. Progressive iso-surface extraction from hierarchical 3d meshes. *Computer Graphics Forum*, 17(3):125–135, 1998. 28

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*, (PrePrints), 2014. 52

S. Guthe and W. Strasser. Real-time decompression and visualization of animated volume data. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 349–356, Washington, DC, USA, 2001. IEEE Computer Society. 30

R. B. Haber and D. A. McNabb. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In *Visualization in Scientific Computing*. 1990. 22

M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005. 29, 136

M. Hadwiger, A. Kratz, C. Sigg, and K. Bühler. GPU-accelerated deep shadow maps for direct volume rendering. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 49–52, New York, NY, USA, 2006. ACM. 150

M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski. Advanced illumination techniques for gpu volume raycasting. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 1:1–1:166, New York, NY, USA, 2008. ACM. 29

R. Haimes and D. E. Edwards. Visualization in a parallel processing environment. In *Proceedings of the 35th AIAA Aerospace Sciences Meeting*, pages 97–0348, 1997. 32

T. Han and T. S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM. 21, 131

T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of transfer functions with stochastic search techniques. In *Visualization*, pages 227–234, 1996. 31

J. Heer and B. Shneiderman. Interactive dynamics for visual analysis. *Commun. ACM*, 55(4):45–54, Apr. 2012. 157

M. Held. FIST: Fast industrial-strength triangulation of polygons. Technical report, Algorithmica, 2000. 28, 100

R. Herzog, S. Kinuwaki, K. Myszkowski, and H.-P. Seidel. Render2MPEG: a perception-based framework towards integrating rendering and video compression. In *Computer Graphics Forum*, volume 27(2), pages 183–192. Blackwell, 2008. 32

J. Highsmith. *Agile software development ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 157

W.-M. Hwu, K. Keutzer, and T. Mattson. The concurrency challenge. *Design Test of Computers, IEEE*, 25(4):312–320, 2008. 16

J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995. 79

C. Johnson and C. Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004. 22

JPEG. ISO/IEC IS 10918-1, ITU-T Recommendation T.81, 1992. 108, 109

D. Kauker, H. Sanftmann, S. Frey, and T. Ertl. Memory Saving Fourier Transform on GPUs. In *International Conference on Computer and Information Technology*, pages 67–75. IEEE, 2010. 161

I. Keidar. ACM SIGACT news distributed computing column 32: the year in review. *SIGACT News*, 39(4):53–54, Nov. 2008. 12

J. E. Kelley, Jr and M. R. Walker. Critical-path planning and scheduling. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173, New York, NY, USA, 1959. ACM. 22

D. B. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2012. 7, 17, 20

T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware. In C. Silva and E. Gröller and H. Rushmeier, editor, *Proceedings of IEEE Visualization '05*, pages 223–230. IEEE, 2005. 30

A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. D. Hansen, and H. Hagen. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum*, 28(1):26–40, 2009a. 101

A. M. Knoll, I. Wald, and C. D. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer: International Journal of Computer Graphics*, 25(3):209–225, 2009b. 29

L. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *SIGGRAPH '01*, pages 57–66, 2001. 28

L. P. Kobbelt and M. Botsch. An interactive approach to point cloud triangulation. In *Proc. Eurographics*, 2000. 28

D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno. Protected interactive 3D graphics via remote rendering. SIGGRAPH '04, pages 695–703. ACM, 2004. 32

A. Kratz, J. Reininghaus, M. Hadwiger, and I. Hotz. Adaptive screen-space sampling for volume ray-casting. Technical Report 11-04, ZIB, Takustr.7, 14195 Berlin, 2011. 30

M. Kraus and T. Ertl. Cell-projection of cyclic meshes. In *Proceedings of the conference on Visualization '01*, VIS '01, pages 215–222, Washington, DC, USA, 2001. IEEE Computer Society. 28

J. Krüger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society. 27, 28, 29

Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996. 22

P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. SIGGRAPH '94, pages 451–458, New York, NY, USA, 1994. ACM. 28

A. Lagae and P. Dutré. A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum*, 27(1):114–129, 2008. 52

E. LaMar and V. Pascucci. A multi-layered image cache for scientific visualization. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 61–68, Washington, DC, USA, 2003. IEEE Computer Society. 31

K. Lee, D. Chu, E. Cuervo, J. Kopf, S. Grizan, A. Wolman, and J. Flinn. Delorean: Using speculation to enable low-latency continuous interaction for cloud gaming. Technical Report MSR-TR-2014-115, August 2014. 32

W.-J. Lee, V. Srini, and T.-D. Han. Adaptive and Scalable Load Balancing Scheme for Sort-Last Parallel Volume Rendering on GPU Clusters. In *International Workshop on Volume Graphics*, 2005. 31

M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990a. 27, 30

M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, 1990b. 30

M. Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, July 1990c. 4, 29

M. Levoy and P. Hanrahan. Light field rendering. SIGGRAPH '96, pages 31–42, 1996. 31

Z. Li, C. Wang, and R. Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. *International Symposium on Parallel and Distributed Processing*, 1:312–317, 2002. 21

S. Lietsch and O. Marquardt. A CUDA-supported approach to remote rendering. In *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part I*, ISVC'07, pages 724–733. Springer-Verlag, 2007. 32

X. Liu, Y. Song, S. Liu, and H. Wang. Automatic taxonomy construction from keywords. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pages 1433–1441, New York, NY, USA, 2012. ACM. 167

P. Ljung, C. Winskog, A. Persson, C. Lundstrom, and A. Ynnerman. Full body virtual autopsies using a state-of-the-art volume rendering pipeline. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):869–876, 2006. 30

E. L. Lloyd. Critical path scheduling with resource and processor constraints. *J. ACM*, 29(3):781–811, 1982. 22

W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987. 27, 28

R. A. Lorie and H. R. Strong. US Patent 4,435,758: Method for conditional branch execution in SIMD vector processors, 1984. 21

E. J. Luke and C. D. Hansen. Semotus visum: A flexible remote visualization framework. In *Proceedings of the Conference on Visualization '02*, VIS '02, pages 61–68, Washington, DC, USA, 2002. IEEE Computer Society. 31

A. Lumsdaine, C. Weaver, and J. A. Cottam. Watch this: A taxonomy for dynamic data visualization. In *Proceedings of the 2012 IEEE Conference on Visual Analytics Science and Technology (VAST)*, VAST '12, pages 193–202, Washington, DC, USA, 2012. IEEE Computer Society. 157

K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2000. 32

K.-L. Ma, M. F. Cohen, and J. S. Painter. Volume seeds: A volume exploration technique. *The Journal of Visualization and Computer Animation*, 2(4):135–140, 1991. 31

K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering*, pages 15–22, 1993. 31, 146

M. Makhinya, S. Eilemann, and R. Pajarola. Fast compositing for cluster-parallel rendering. In J. Ahrens, K. Debattista, and R. Pajarola, editors, *EGPGV*, pages 111–120, Norrköping, Sweden, 2010. Eurographics Association. 146

S. Marchesin, C. Mongenet, and J.-M. Dischler. Dynamic Load Balancing for Parallel Volume Rendering. In *EGPGV*, pages 51–58. Eurographics Association, 2006. 32

W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Symposium on Interactive 3D graphics*, pages 7–16, 1997. 31

S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of the Conference on Visualization '94*, VIS '94, pages 100–107, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 101

N. Marwan, M. Carmenromano, M. Thiel, and J. Kurths. Recurrence plots for the analysis of complex systems. *Physics Reports*, 438(5-6):237–329, 2007. 38

T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004. 4, 157, 158, 159

T. G. Mattson, D. Scott, and S. R. Wheat. A teraflop supercomputer in 1996: The ASCI TFLOP system. In *IPPS*, pages 84–93, 1996. 12, 19

N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1 (2):99–108, 1995. 25, 26, 91

L. McMillan and G. Bishop. Plenoptic modeling: an image-based rendering system. SIGGRAPH '95, pages 39–46, 1995. 31

J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 235–246, 2010. 21

M. Meyer, B. Nelson, R. M. Kirby, and R. Whitaker. Particle systems for efficient and accurate high-order finite element visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1015–1026, 2007. 28

S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994. 31

K. Moreland. A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3): 367–378, Mar. 2013. 11, 24, 32

K. Moreland, D. Lepage, D. Koller, and G. Humphreys. Remote rendering for ultrascale data. *Journal of Physics: Conference Series*, 125(1):012096, 2008. 32

S. Moy and E. Lindholm. US Patent 6,947,047: Method and System for Programmable Pipelined Graphics Processing with Branching Instructions, 2005. 21

K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999. 28

J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, Feb. 1999. 23

C. Müller, M. Strengert, and T. Ertl. Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *EGPGV*, pages 59–66. Eurographics Association, 2006. 32

C. Müller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):605–617, 2009. 163

B. Nelson, R. M. Kirby, and R. Haimes. GPU-Based Interactive Cut-Surface Extraction From High-Order Finite Element Fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1803 –1811, 2011. 29

U. Neumann. Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering*, pages 97–104, 1993. 31

G. M. Nielson. Dual marching cubes. VIS '04, pages 489–496, 2004. 28

J. Novák, V. Havran, and C. Dachsbacher. Path Regeneration for Interactive Path Tracing. In *Proceedings of EUROGRAPHICS 2010, short papers*, pages 61–64, 2010. 20

NVIDIA Corporation. NVIDIA CUDA C programming guide, 2013a, Version 5.5. 9, 17, 18, 19, 20, 128

NVIDIA Corporation. CUDA C best practices guide, 2013b, Version 5.5. ix, 4, 157, 158

OpenACC. The OpenACC application programming interface version 2.0, 2013. 20

OpenMP Architecture Review Board. OpenMP application program interface version 3.1, May 2011. 20

R. S. Overbeck, C. Donner, and R. Ramamoorthi. Adaptive wavelet rendering. *ACM Transactions on Graphics (SIGGRAPH Asia 09)*, 28(5):1–12, 2009. 30

C. A. Pagot, J. Vollrath, F. Sadlo, D. Weiskopf, T. Ertl, and J. Comba. Interactive isocontouring of high-order surfaces. In *Scientific Visualization: Interactions, Features, Metaphors*, 2011. 28

D. Pajak, R. Herzog, E. Eisemann, K. Myszkowski, and H.-P. Seidel. Scalable remote rendering with depth and motion-flow augmented streaming. *Computer Graphics Forum*, 30(2):415–424, 2011. 32

M. E. Palmer, S. Taylor, and B. Totty. Exploiting deep parallel memory hierarchies for ray casting volume rendering. In *Proceedings of the IEEE symposium on Parallel rendering*, PRS '97, pages 15–ff., New York, NY, USA, 1997. ACM. 31

A. Panagiotidis, D. Kauker, S. Frey, and T. Ertl. DIANA: A Device Abstraction Framework for Parallel Computations. In *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, United Kingdom, 2011. Civil-Comp Press. 162

H. Park, D. S. Turaga, O. Verscheure, and M. van der Schaar. A framework for distributed multimedia stream mining systems using coalition-based foresighted strategies. In *ICASSP '09: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1585–1588, Washington, DC, USA, 2009. IEEE Computer Society. 21

S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of the conference on Visualization '98*, VIS '98, pages 233–238, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. 27, 136

R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the gpu. In *Innovative Parallel Computing*, page 9, May 2012. 116

D. Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. 12

T. Peterka, R. Ross, H. Yu, K.-L. Ma, W. Kendall, and J. Huang. Assessing improvements to the parallel volume rendering pipeline at large scale. In *Ultrascale Visualization, 2008. UltraVis 2008. Workshop on*, pages 13–23, Nov 2008a. 32

T. Peterka, H. Yu, R. Ross, and K.-L. Ma. Parallel volume rendering on the ibm blue gene/p. In *Proc. Eurographics Parallel Graphics and Visualization Symposium 2008*, pages 73–80, 2008b. 31

H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. SIGGRAPH '00, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. 28

M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, USA, 2nd edition, 2010. 76

C. S. R. Prabhu. *Grid and Cluster Computing.* Prentice-Hall of India Pvt.Ltd, 2010. 21

H. Qu, M. Wan, J. Qin, and A. Kaufman. Image based rendering with stable frame rates. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 251–258. IEEE Computer Society Press, 2000. 30, 85

M. Ramasubramanian, S. N. Pattanaik, and D. P. Greenberg. A perceptually based physical error metric for realistic image synthesis. SIGGRAPH '99, pages 73–82, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. 30

P. Rautek, S. Bruckner, and E. Gröller. Semantic layers for illustrative volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1336–1343, 2007. 31

J. Reininghaus, N. Kotava, D. Gunther, J. Kasten, H. Hagen, and I. Hotz. A scale space based persistence measure for critical points in 2d scalar fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2045–2052, 2011. 37

J.-F. Remacle, F. Henrotte, T. Baudouin, C. Geuzaine, E. Béchet, T. Mouton, and E. Marchandise. A frontal delaunay quad mesh generator using the l∞ norm. In *20th Meshing Roundtable*, pages 455–472. 2012. 28

C. Rezk-Salama, S. Todt, and A. Kolb. Raycasting of light field galleries from volumetric data. In *Computer Graphics Forum*, EuroVis, pages 839–846, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. 31

C. Rezk-Salama, M. Hadwiger, T. Ropinski, and P. Ljung. Advanced illumination techniques for GPU volume raycasting. In *ACM SIGGRAPH Courses Program*, 2009. 29

T. Rhyne, M. Tory, T. Munzner, M. Ward, C. Johnson, and D. Laidlaw. Information and scientific visualization: separate but equal or happy together at last. In *Visualization, 2003. VIS 2003. IEEE*, pages 611–614, Oct 2003. 157

C. Rocchini, P. Cignoni, F. Ganovelli, C. Montani, P. Pingi, and R. Scopigno. The marching intersections algorithm for merging range images. *The Visual Computer*, 20(2-3):149–164, 2004. 28

T. Ropinski, J. Prassni, F. Steinicke, and K. Hinrichs. Stroke-based transfer function design. In *International Symposium on Volume and Point-Based Graphics*, pages 41–48. Eurographics Association, 2008. 31

P. Rosenthal and L. Linsen. Direct isosurface extraction from scattered volume data. In *EuroVis*, pages 99–106, 2006. 28

S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Visualization*, pages 109–116, 2000. 28

F. Sadlo, M. Üffinger, C. Pagot, D. Osmari, J. Comba, T. Ertl, C.-D. Munz, and D. Weiskopf. Visualization of cell-based higher-order fields. *Computing in Science and Engineering*, 13:84–91, 2011. 29

N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10, 2009. 49

C. E. Scheidegger, S. Fleishman, and C. T. Silva. Triangulating point set surfaces with bounded error. In *EG symposium on Geometry processing*, 2005. 28

H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-based CT reconstruction using the Common Unified Device Architecture (CUDA). *SIAM Journal of Applied Mathematics*, 2007. 118

R. Schneiders. Refining quadrilateral and hexahedral element meshes. In *5th International Conference on Grid Generation in Computational Field Simulations*, pages 679–688. CRC Press, 1996. 97

J. Schreiner, C. Scheidegger, and C. Silva. High-quality extraction of isosurfaces from regular and irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 12:1205–1212, 2006. 28

W. Schroeder, K. M. Martin, and W. E. Lorensen. *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. 24

F. J. Seinstra, J.-M. Geusebroek, D. Koelma, C. G. Snoek, M. Worring, and A. W. Smeulders. High-performance distributed video content analysis with parallel-horus. *IEEE MultiMedia*, 14(4):64–75, 2007. 21

K. Seshadrinathan and A. C. Bovik. Motion tuned spatio-temporal quality assessment of natural videos. *Transactions on Image Processing*, 19(2):335–350, 2010. 69, 73

J. Shade, S. Gortler, L.-w. He, and R. Szeliski. Layered depth images. SIGGRAPH '98, pages 231–242, 1998. 31, 95

N. Shareef, T.-Y. Lee, H.-W. Shen, and K. Mueller. An image-based modelling approach to GPU-based rendering of unstructured grids. In *Volume Graphics*, pages 31–38, 2006. 31

H. Sheikh and A. Bovik. Image information and visual quality. *IEEE Transactions on Image Processing*, 15(2):430–444, 2006. 69

H.-W. Shen and C. R. Johnson. Differential volume rendering: a fast volume visualization technique for flow animation. In *Proceedings of the conference on Visualization '94*, VIS '94, pages 180–187. IEEE Computer Society Press, 1994. 30, 85

P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics*, pages 63–70, 1990. 28

H. Shum and S. B. Kang. A survey of image-based rendering techniques. In *In SPIE Videometrics*, pages 2–16, 1999. 30

N. Stankovic and K. Zhang. A distributed parallel programming framework. *IEEE Transactions on Software Engineering*, 28(5):478–493, 2002. 22

M. Steffen and J. Zambreno. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 237–248, 2010. 21

S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics*, pages 187–195, 2005. 28, 29

J. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010. 20

H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, Sept. 2005. 20

J. D. Teresco, J. Faik, and J. E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science and Engineering*, 7(2):40–50, 2005. 21

H. Theisel. Exact isosurfaces for marching cubes. *Computer Graphics Forum*, 21(1):19–32, 2002. 28

A. Tikhonova, C. Correa, and K.-L. Ma. Explorable images for visualizing volume data. In *IEEE Pacific Visualization Symposium*, pages 177–184, 2010a. 31

A. Tikhonova, C. D. Correa, and K.-L. Ma. An exploratory technique for coherent visualization of time-varying volume data. In *Computer Graphics Forum*, pages 783–792. Eurographics Association, 2010b. 31

M. Tory and T. Möller. Rethinking visualization: A high-level taxonomy. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pages 151–158, Washington, DC, USA, 2004. IEEE Computer Society. 157

H. Turbell. *Cone-Beam Reconstruction Using Filtered Backprojection.* PhD thesis, Linköping University, Sweden, February 2001, Dissertation No. 672. 118

G. Turk and M. Levoy. Zippered polygon meshes from range images. SIGGRAPH '94, pages 311–318, 1994. 28

S. Tzeng, A. Patney, and J. D. Owens. Task Management for Irregular-Parallel Workloads on the GPU. In M. Doggett, S. Laine, and W. Hunt, editors, *Proceedings of the Conference on High Performance Graphics*, pages 29–37. Eurographics Association, 2010. 20

F. Vega-Higuera, P. Hastreiter, R. Fahlbusch, and G. Greiner. High performance volume splatting for visualization of neurovascular data. In *VIS '05*, pages 271–278, 2005. 28

I. Viola, A. Kanitsar, and E. Gröller. Importance-driven volume rendering. In *VIS '04*, pages 139–146, 2004. 30

S. Viswanathan, B. Veeravalli, and T. G. Robertazzi. Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Transactions Parallel Distributed Systems*, 18(10):1450–1461, 2007. 21

C. Wang, J. Gao, and H.-W. Shen. Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *EGPGV*, pages 23–30, 2004. 31

L. Wang, Y.-z. Huang, X. Chen, and C.-y. Zhang. Task scheduling of parallel processing in cpu-gpu collaborative environment. In *ICCSIT '08: Proceedings of the 2008 International Conference on Computer Science and Information Technology*, pages 228–232, 2008. 21

Z. Wang, E. Simoncelli, and A. Bovik. Multiscale structural similarity for image quality assessment. In *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1398–1402, Nov 2003. 107, 112

D. Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 23

D. H. D. West. Updating mean and variance estimates: An improved method. *Commun. ACM*, 22(9):532–535, 1979. 77

R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. SIGGRAPH '98, pages 169–177, New York, NY, USA, 1998. ACM. 27

R. Westermann, L. Kobbelt, and T. Ertl. Real-time exploration of regular volume data by adaptive reconstruction of iso-surfaces. *The Visual Computer*, 15:100–111, 1999. 28

L. A. Westover. *Splatting: a parallel, feed-forward volume rendering algorithm.* PhD thesis, 1991, UMI Order No. GAX92-08005. 27, 28

D. F. Wiley, H. R. Childs, B. F. Gregorski, B. Hamann, and K. I. Joy. Contouring curved quadratic elements. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, pages 167–176, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. 28

J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, July 1992. 30

C. Woolley, D. Luebke, B. Watson, and A. Dayal. Interruptible rendering. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, I3D '03, pages 143–151, New York, NY, USA, 2003. ACM. 70

S. Woop, J. Schmittler, and P. Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. Graph.*, 24:434–444, July 2005. 21

Y. Wu and H. Qu. Interactive transfer function design based on editing direct volume rendered images. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1027–1040, 2007. 31

B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on PC clusters. *IEEE Comput. Graph. Appl.*, 21: 62–70, July 2001. 31

F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. *IEEE Transactions on Nuclear Science*, pages 654–663, Jun 2005. 118

K. Yelick. Parallel Programming for Multicore. Lecture notes CS194-2, 2007. 14, 15

J. P. Zbilut and C. L. Webber. Embeddings and delays as derived from quantification of recurrence plots. *Physics Letters A*, 171(34):199–203, 1992. 38

E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU applications on the fly: Thread Divergence Elimination through Runtime Thread-data Remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126, New York, NY, USA, 2010. ACM. 21

Z. Zheng, N. Ahmed, and K. Mueller. iView: A feature clustering framework for suggesting informative views in volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1959–1968, 2011. 94

K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, and B. Guo. RenderAnts: interactive Reyes rendering on GPUs. In *SIGGRAPH Asia '09*, pages 155:1–11, New York, NY, USA, 2009. ACM. 22

Y. Zhou, B. Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 135–142, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press. 28

K. Zuiderveld, A. Koning, and M. Viergever. Acceleration of ray-casting using 3d distance transforms. In *Visualization in Biomedical Computing II, Proc. SPIE 1808*, pages 324–335, 1992. 29

M. Üffinger, S. Frey, and T. Ertl. Interactive High-Quality Visualization of Higher-Order Finite Elements. *CGF*, 29 (2):337–346, 2010. 95, 98, 101, 159