TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND
VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

In-Situ Visualisierung und Streaming von Plasmasimulationsdaten

Alexander Matthes
(Geboren am 7. November 1988 in Berlin, Mat.-Nr.: 3514123)

Betreuer: Dr. Sebastian Grottel (TU Dresden),
Dr. Michael Bussmann (HZDR)

Dresden, 14. März 2016

Aufgabenstellung

Im Rahmen der am HZDR durchgeführten Big-Data-Simulationen soll in dieser Diplomarbeit eine Bibliothek für Remote-In-Situ-Visualisierung entstehen, unabhängig von konkreten Simulations-Codes.

Der Fokus der Arbeit liegt hierbei auf dem für die Visualisierung notwendigen Rendering. Kommunikationsinfrastruktur und Interaktions-Front-End sollen nur so weit entwickelt werden wie notwendig, um das Gesamtsystem als funktionsfähige Anwendung evaluieren zu können. Vor allem die Laufzeitoptimierung soll sich auf das Rendering konzentrieren. Zentraler Aspekt ist die generische Abbildung von Simulationsdaten auf Visualisierungsdaten unter Vermeidung jeglicher Kopieroperationen, im Besonderen teurer Deep-Copy-Operationen.

Obwohl Simulation und Visualisierung verteilt auf Knoten eines HPC-Systems ausgeführt werden, kann in dieser Arbeit der Vereinfachung halber davon ausgegangen werden, dass die für die Visualisierung notwendigen Teildaten lokal auf den jeweiligen Knoten zur Verfügung stehen. Für die Visualisierung können drei Typen von Daten angenommen werden: Skalarfelder (maximal 3D), Vektorfelder und Partikeldaten. Diese Arbeit soll eine Volumenvisualisierung für Skalarfelder, sowohl direkte Darstellung als auch Iso-Flächendarstellung, in einem verteilten Rendering umsetzen. Die Visualisierung der Vektorfelder, beispielsweise durch semi-transparentes 3D LIC, und der Partikeldaten sind optional. Mögliche Technologien für die Umsetzung umfassen CUDA, CUDA-basiertes OpenGL, GPGPU-Software-Rendering. Zunächst soll eine Anforderungsanalyse an die Datenstrukturen aus Sichten der Visualisierung und Simulation erfolgen und hieraus ein Konzept für die Datenstrukturen und deren Umsetzung erarbeitet werden.

Ergänzend ist eine Literaturrecherche durchzuführen zu In-Situ-Visualisierung, im Besonderen Visualisierungen im Kontext von GPGPU-Simulationen, generischen Frameworks und verteiltem Volumenrendering. Eine weitere Recherche wird zu den weiteren notwendigen Softwarekomponenten erfolgen: der Netzwerkkommunikation, der Videokompression und über Technologien für darstellende Frontends. Mögliche Technologien, die allerdings nicht zwingend genutzt werden müssen, umfassen Graybat, GStreamer, HTML5+WebGL. Auf Basis der durchgeführten Analyse und Recherche wird der Bearbeiter die Software in allen Komponenten planen und implementieren, wobei natürlich weitgehend auf bestehenden Softwarepaketen aufgebaut werden sollte.

Abschließend soll in einer Evaluierung die Effektivität und Effizienz des umgesetzten Systems nachgewiesen werden. Zentral ist hierbei die zusätzliche Rechenlast auf dem HPC-System, gegenüber einer Simulation ohne die neue Visualisierungs-

komponeten und die Gesamtgeschwindigkeit, angegeben in Renderinggeschwindigkeit und Darstellungsgeschwindigkeit am Frontend, jeweils abhängig von den Datengrößen.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

In-Situ Visualisierung und Streaming von Plasmasimulationsdaten

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 14. März 2016

Alexander Matthes

“There is a building. Inside this building there is a level where no elevator can go, and no stair can reach. This level is filled with doors. These doors lead to many places. Hidden places. But one door is special. One door leads to the source.“

The Keymaker

Matrix Reloaded (2003)

Kurzfassung

Computersimulationen sind heutzutage ein wichtiges, wissenschaftliches Instrument. Visualisierungen dieser helfen dabei die Daten zu verstehen und zu interpretieren. Da auf heutigen Peta- und zukünftigen Exascalesystemen zu viele Daten pro Zeitschritt entstehen, um sie komplett in einem Postprocessingschritt visualisieren zu können, stellt diese Arbeit die In-Situ Visualisierungsbibliothek ISAAC vor, die die Daten direkt nach ihrer Entstehung verteilt visualisieren kann, ohne sie zu speichern oder zu übertragen. Mithilfe von Templates und Template Metaprogrammierung wird ein abstraktes, wiederverwendbares Interface beschrieben, welches trotzdem eine simulationsspezifische Optimierung erlaubt. Die Visualisierung mittels Raytracing erfolgt auf den Originaldaten der Simulation, auch wenn diese auf einem Rechenbeschleuniger wie Nvidia GPUs oder Intel Xeon Phis läuft. In diesem Fall nutzt auch ISAAC die Beschleunigerhardware. Des Weiteren wird die Möglichkeit beschrieben, beliebige Metadaten zwischen Clients und Simulationen auszutauschen, insbesondere um sie live steuern zu können.

Neben der C++ Visualisierungsbibliohtek für Simulationen wird weiterhin ein generischer, zentraler Server zur Videostreamerzeugung motiviert und beschrieben sowie ein einfacher HTML Referenzclient zur Anzeige und Steuerung der Simulation implementiert. Zur Evaluierung der Lösung wird mit ISAAC die verteilte, GPU-beschleunigte Plasmasimulation PIConGPU des HZDR visualisiert. Des Weiteren wird die Rendergeschwindigkeit in Abhängigkeit verschiedener Parameter gemessen und diskutiert.

Abstract

Computer simulations are important scientific instruments these days. Visualizations help to understand and interpret these data. Since with the classic post processing approach too much data is produced per time step on recent peta scale and future exa scale systems this thesis introduces the in situ visualization library ISAAC, which is able to visualize distributed data right after creation without the

need to store or transmit them. Using templates and template meta programming an abstract and reusable interface is described, which still enables simulation specific optimizations. The visualization over ray tracing works on the original data of the simulation, even if it runs on computation accelerators like Nvidia GPUs or Intel Xeon Phis. In these cases ISAAC is using the accelerators, too. Furthermore a way of sending and receiving meta data from and to simulations is described, especially to be able to steer them.

Besides this C++ library for visualization of simulations, a generic central server for creating video streams is motivated and described and also a simple HTML reference client for showing and steering simulations is implemented. For the evaluation of the solution, ISAAC is used to visualize the distributed gpu-based plasma simulation PIConGPU form the HZDR. Furthermore the speed of rendering dependent on different parameters is measured and discussed.

Inhaltsverzeichnis

1 Einleitung **3**

2 Verwandte Arbeiten **5**
2.1 In-Situ Processing 5
2.2 Volumenvisualisierung 7
2.3 Bisherige Implementierungen 8

3 Grundlagen **11**
3.1 Supercomputer 2016 12
3.1.1 Rechenbeschleuniger 13
3.1.2 Hochparallele Anwendungen 14
3.1.3 Verteilte Visualisierung 18
3.2 Volumenvisualisierung 20
3.3 Template Metaprogrammierung 25
3.3.1 Templateprogrammierung 25
3.3.2 C++ Metaprogrammierung 26
3.3.3 Rechenbeschleuniger abstrahieren 28
3.4 JavaScript Object Notation 30

4 Konzept **33**
4.1 In Situ Animation of Accelerated Computations 33
4.1.1 ISAAC Template Bibliothek 34
4.1.2 ISAAC Server 37
4.1.3 ISAAC Client 38
4.2 Einordnung im Visualisierungsprozess 38

5 Umsetzung **41**
5.1 In-Situ Bibliothek 41
5.1.1 Quellenbeschreibung 41
5.1.2 ISAAC initialisieren 44
5.1.3 Visualisierung 45
5.1.3.1 Functor-Chain Vorkompilierung 47
5.1.3.2 Volumenrenderkernel 50
5.1.3.3 Aufruf des Renderkernels 59

5.1.4 Paralleles Volumenrendering 61
5.1.5 Steuerung der Simulation 62
5.1.6 Simulationseinbindungen 63
5.2 ISAAC Server 66
5.2.1 Abstraktes Pluginsystem 66
5.2.2 Implementierung der Klasse MetaDataConnector 67
5.2.3 Implementierungen der Klasse ImageConnector 68
5.2.4 Der Broker 70
5.3 Referenzclient 71

6 Ergebnisse und Diskussion 75
6.1 Geschwindigkeit 75
6.1.1 Optimale IceT Strategie 77
6.1.2 Kernelgeschwindigkeit 78
6.1.3 Interaktionslatenz 90
6.1.4 Kompiliergeschwindigkeit 92
6.2 Skalierbarkeit 94
6.3 Wissenschaftlicher Nutzen am Beispiel PIConGPU 95

7 Zusammenfassung 101

8 Ausblick 103

Literaturverzeichnis 105

1 Einleitung

Computersimulationen sind heutzutage ein oft genutztes Mittel, um wissenschaftliche Daten zu generieren, z. B. weil die Untersuchungsobjekte astronomisch weit entfernt sind oder experimentelle Versuchsreihen zu kostspielig. Sie werden aber auch genutzt, um zusammen mit Experimenten Modelle zu validieren, die analytisch nicht lösbar wären, z. B. weil sehr viele Einzelpartikel involviert sind. Derartig große Datenmengen sind intuitiv schwer fassbar, weshalb sie zum besseren Verständnis oft visualisiert werden. In der Vergangenheit konnten die Daten generiert, abgespeichert und in einem Nachbearbeitungsschritt visualisiert werden. Auf heutigen hochparallelen Petascale-Hochleistungsrechnern und erst recht auf zukünftigen Exascale-Systemen entstehen in einem Zeitschritt jedoch so viele Daten, dass sie nicht zeitnah übertragen oder gespeichert werden können und der klassische Nachbearbeitungsansatz somit kaum noch praktikabel ist.

Ein Konzept zur Vermeidung dieses Verlusts ist In-Situ Processing, in dem die Daten live und simulationsnah visualisiert werden, meist ohne sie über ein Netzwerk zu übertragen oder auf Festplatte zu speichern. Die bisherigen Ansätze sind dabei sehr simulationsspezifisch und schwer auf andere Anwendungen übertragbar oder so generisch, dass sie es nicht schaffen das volle Potential der Rechencluster auszunutzen. Insbesondere, wenn die Simulationen auf modernen Rechenbeschleunigern wie Nvidia GPUs oder Intel Xeon Phis laufen, werden bei diesem Ansatz zeitaufwendige Kopieroperationen zwischen RAM und Rechenbeschleuniger ausgeführt und die Möglichkeiten der Rechenbeschleuniger außer Acht gelassen.

Diese Arbeit stellt deshalb eine In-Situ Visualisierungsmethode vor, die einerseits direkt auf den Simulationsdaten und auf etwaigen Rechenbeschleunigern arbeiten kann, ohne die Daten vorher kopieren oder konvertieren zu müssen, aber andererseits eine wiederverwendbare API definiert, sodass auch zukünftige Simulationen profitieren können. Es soll möglich sein, nicht mit, sondern auf den Originaldaten zu arbeiten. Dazu wird die Template Bibliothek ISAAC beschrieben und implementiert, die auf den schon vorhandenen Skalar- und Vektorfeldern einer hochparallelen wissenschaftlichen Anwendung Raycasting durchführen und damit sowohl Visualisierungen nach dem Emissions/Absorptions-Modell als auch Isoflächendarstellungen mit einem einfachen Beleuchtungsmodell erzeugen kann. Durch Template Metaprogrammierung sollen viele Aufgaben schon beim Kompilieren des Programm-

quelltextes abgearbeitet und so performantere und kompaktere Anwendungen erzeugt werden. Die Idee ist, möglichst viel Komplexität von der Ausführung in die Kompilierung zu verschieben. Mithilfe der Bibliothek ALPAKA sollen verschiedene Rechenbeschleuniger abstrahiert und ISAAC so universell, hardwareunabhängig eingesetzt werden können.

Der Fokus der Arbeit liegt in der performanten In-Situ Visualisierung, nichtsdestotrotz wird eine Infrastruktur mit einem zentralen Server und beliebig vielen beobachtenden Clients beschrieben werden, um die generierten Visualisierungen zu den WissenschaftlerInnen zu streamen und gleichzeitig Steuerinformationen an die Visualisierung und Simulation zu schicken und von diesen Metadaten zu empfangen.

Die GPU-beschleunigte Plasmasimulation PIConGPU des Helmholtz-Zentrums Dresden-Rossendorf (HZDR) und dessen Rechencluster Hypnos werden genutzt, um die benötigten Grundlagen zu erarbeiten, die Komponenten ISAACs zu entwerfen und die Tauglichkeit an einer realen, hochparallelen, wissenschaftlichen Software zu evaluieren.

In den verwandten Arbeiten wird dazu einerseits näher auf die bisherigen In-Situ Ansätze eingegangen, aber auch Parallelen zu anderen Arbeiten paralleler und genereller Volumenvisualisierung aufgezeigt. Das Kapitel Grundlagen wird die Recherchearbeit zu den vielen Themenkomplexen, die die Arbeit anschneidet, widerspiegeln. Es erklärt die Herausforderungen, effizient mit Supercomputern, Rechenbeschleunigern und wissenschaftlichen Simulationen zu arbeiten und zeigt die wichtigsten Konzepte der Metaprogrammierung in C++. Die Methodik und Umsetzung werden die einzelnen Komponenten ISAACs vorstellen und erläutern. Designentscheidungen werden begründet und deren Implementierung beschrieben. Am Schluss, um die Methodik und die Umsetzung zu verteidigen, wird ISAAC in PIConGPU eingebunden werden und Laufzeitmessungen auf einem realen Cluster mit verschiedenen Parametern durchgeführt und gezeigt, dass die vorgestellte Lösung die Probleme der bisherigen In-Situ Visualisierungen und des klassischen Postprocessings lösen kann, bevor eine Zusammenfassung das Erreichte noch einmal rekapituliert und ein Ausblick für mögliche Erweiterungen und alternative Lösungsstrategien gegeben wird.

2 Verwandte Arbeiten

Die Themen parallele In-Situ Volumenvisualisierung und deren Streaming und Steuerung verknüpfen verschiedene Themengebiete der Informatik miteinander. In Zeiten von Exa- und Petascalesystemen ist ein klassisches Postprocessing aller zustande kommender Daten nicht mehr realisierbar – insbesondere wenn Rechenbeschleuniger wie Nvidia GPUs oder Intel Xeon Phis involviert sind. Deshalb geht es einerseits um eine simulationsnahe (Zwischen-)Verarbeitung der wissenschaftlichen Daten, die mit möglichst wenig Festplatten- und Netzwerkzugriffen auskommt. Andererseits muss aber auch evaluiert werden, welche Visualisierungstechniken sich in diesem speziellen Anwendungsfall anbieten, vor allem wenn die Visualisierung dieselben Ressourcen wie die Simulation beansprucht. Da die Simulation und somit auch die Visualisierung hoch parallel ablaufen, muss es des Weiteren möglich sein die Teilergebnisse zu einem sinnvollen Gesamtbild zu kombinieren.

2.1 In-Situ Processing

Klassisches *In-Situ Processing* bedeutet Daten vor der Speicherung oder Übertragung in jedem Zeitschritt zu konvertieren, zu filtern oder zu komprimieren. In-Situ Visualisierung stellt hierbei einen Sonderfall dar, bei welchem die oft dreidimensionalen, wissenschaftlichen Daten zu zweidimensionalen Abbildern konvertiert werden, die sehr viel einfacher gespeichert oder übertragen werden können [Ma09]. Es zeichnet sich ab, dass ohne solche In-Situ Prozesse eine Verarbeitung der wissenschaftlichen Daten spätestens mit Exascale-Computern kaum mehr möglich sein wird [MWYT07]. *Exascale* bedeutet, dass mindestens ein ExaFLOPS, also 10^{18} (eine Trillion) Floatingpointoperationen pro Sekunde, erreicht werden. Zu der Zeit dieser Arbeit konnte der schnellste Superrechner Tianhe-2 (Guangzhou, China) im LinPACK Benchmark fast 34 PetaFLOPS erreichen [SDSM16].

Lakshminarasimhan et al. konnten zeigen, dass eine In-Situ Komprimierung wissenschaftlicher Daten um 85% bei nur 1% Informationsverlust möglich ist [LSE+11]. Eine weitere Möglichkeit besteht darin, nur eine repräsentative Stichprobe der vorhandenen Daten dem Postprocessing zuzuführen [WAF+11]. Die Ansätze unterscheiden sich dabei nicht nur in der Art der In-Situ Verarbeitung,

sondern auch in der Nähe des Prozesses zur laufenden Simulation. Man unterscheidet einen engen und einen lockeren Ansatz. Eine enge Verknüpfung zwischen Simulation und In-Situ Processing bedeutet, dass Simulation und Nachbearbeitungsprozess auf derselben Hardware laufen. Vorteil ist, dass Festplatten- und Netzwerkbandbreite gespart wird [TYRG+06, FBF+15]. Andererseits muss sich die Simulation nun die Ausführungszeit und den zur Verfügung stehenden Arbeitsspeicher mit dem In-Situ Prozess teilen. Anders beim lockeren Ansatz, bei dem die Simulationsdaten über eine Netzwerkverbindung direkt an dedizierte Weiterverarbeitungsknoten geschickt werden, die dann z. B. die Visualisierung übernehmen. Man bezeichnet diesen Ansatz auch als *In-Transit Processing* – im Gegensatz zum (klassischen) In-Situ Processing [BAB+12]. Im weiteren Verlauf dieser Arbeit wird mit „In-Situ Processing“ explizit der eng gekoppelte Ansatz gemeint sein und der lockere Ansatz konsequent als „In-Transit Processing“ bezeichnet. Um die Nachteile beider Systeme auszugleichen, existieren des Weiteren hybride Ansätze, die die zu übertragenden Daten z. B. erst lokal filtern oder komprimieren und dann extern weiterverarbeiten [RCMS11].

Ein konkretes Beispiel für eine In-Situ Visualisierung ist die Arbeit von Yu et al., die eine Verbrennungssimulation live visualisiert haben [YWG+09]. Hagan et al. zeigten, wie GPU Ressourcen sinnvoll verteilt werden können, wenn sowohl Simulation als auch Visualisierung auf den Beschleunigern laufen, um eine hohe Auslastung der Hardware zu erreichen [HC11]. Auch Yu et al. nutzten simulationsinterne Zwischenergebnisse zur direkten Visualisierung von Erdbeben und vermieden so eine hohe I/O-Auslastung [YTB+06].

Ein Problem bei In-Situ Visualisierungen kann sein, dass es oft nicht trivial möglich ist, Feedback an die Visualisierung zu schicken, um deren Parameter zu steuern. Ein Problem entsteht z. B. auch, wenn eine Simulation sehr lange oder in der Nacht läuft und der/die WissenschaftlerIn nicht stundenlang auf das interessierende Ereignis warten kann oder möchte. Kageyama et al. haben hierfür ein In-Situ System entwickelt, welches Tausende oder Millionen von verschiedenen Visualisierungen gleichzeitig erzeugt und als komprimierte Videos speichert. Eine spezielle Software kann aus diesen vielen Einzelvideos dann in einem Postprocessingschritt eine frei erkundbare Visualisierung berechnen [KY14].

Es kann auch ein Problem darstellen, dass die zu visualisierenden Simulationsdaten vor der Visualisierung vorverarbeitet werden müssen, z. B. indem für jedes Element die Länge bestimmt wird oder eine Verschiebung des Wertebereiches stattfinden muss. Klassische Ansätze würden hierfür mehrmals über alle Datenpunkte iterieren und sukzessive solche Abbildungen bzw. Filter anwenden. Moreland et al. beschreiben stattdessen ein System, in dem die einzelnen atomaren Filter (*Worklets*)

in einer konkatenierten Ausführung auf die Simulationsdaten angewendet werden. Dadurch erreichen sie gegenüber klassischen Visualisierungsframeworks wie VTK einen Speedup von bis zu 1 000 [MAGM11].

2.2 Volumenvisualisierung

Die Visualisierung wissenschaftlicher Daten ist ein gut erforschtes Themengebiet. In der Vergangenheit gab es viele Ansätze, die speziellen, beschleunigten Möglichkeiten von Grafikkarten auszunutzen, z. B. in Form von *Splatting* [LWMT97], *Shear-Warp* [Lac95] oder *Texture Based Volume-Rendering* [HS89]. State of the Art ist heutzutage das *Volume Ray Casting* Verfahren, in welchem das zu visualisierende Volumen mithilfe von frei programmierbaren Shadern pro Bildschirmpunkt schrittweise abgetastet wird. Jedem Skalar- oder Vektorwert, der in dem Volumen vorkommen kann, wird dabei über eine *Transferfunktion* eine Farbe zugewiesen, man spricht davon zu *klassifizieren* [RS02, CM93, KKH02, Hsu93].

Die einzelnen Abtastpunkte können dabei mit verschiedenen Methoden zu einer Gesamtfarbe kombiniert werden. Bei der *Maximum Intensity Projection* wird der Punkt mit der höchsten Intensität gewählt [NMR+92, PPL+03, SSKE05]. Bei der *Isoflächendarstellung* werden explizit Intensitäten angegeben, die eine Oberfläche definieren, welche dann angezeigt wird. Auf diese Art und Weise bekommt man eine räumliche Darstellung eines Objektes. Zur Verbesserung der räumlichen Vorstellung können die Gradienten des Punktes im Volumen zur Beleuchtung genutzt oder sogar Schattierung implementiert werden [WE98, TSH98, PPL+03, SSKE05, MCH03]. Es ist auch möglich alle Punkte des Sichtstrahls zu kombinieren, z. B. mit dem *Emissions/Absorptions-Modell*, welches einem Datenpunkt im Raum neben der Farbe auch eine Opazität zuweist und eine Art Darstellung als leuchtendes Gas ermöglicht [SSKE05, MCH03, Sab88, MSLP09].

Gerade die zuletzt genannte Methode kann dabei hohe Ansprüche an die zugrunde liegende Hardware stellen. Aus diesem Grund gibt es verschiedene Optimierungsstrategien, um die Visualisierung zu beschleunigen. Das Volumen wird zumeist von vorne nach hinten abgetastet. Wenn der kombinierte Farbwert eine Opazität von fast Eins besitzt, können dahinter liegende Abtastpunkte ignoriert werden (*early ray termination*). Mit entsprechenden Datentypen ist es auch möglich Regionen in einem Volumen zu überspringen, deren kumulierte Opazität Null beträgt, die also keinen Betrag zum Gesamtfarbwert leisten (*empty space skipping*). Bei hohen Schrittweiten im Raycasting können Artefakte im Volumen entstehen, insbesondere, wenn die Transferfunktion starke Sprünge enthält. Statt die Schrittweite zu verkleinern, bietet sich hier das *Vorintegrieren der Transferfunktion* an. Dabei

wird nicht für einen bestimmten Wert im Volumen eine Farbe und Opazität bestimmt, sondern für den *Slice* zwischen zwei Abtastpunkten. Auf diese Art und Weise können starke, lokale Ansprünge in der Transferfunktion (*Peaks*) besser abgebildet werden [Bru09, EKE01, KW03].

Eine weitere Möglichkeit stellt die Darstellung des Volumens mithilfe von Voxeln oder Partikeln dar [CNLE09], wobei hierbei auch Veränderungen über die Zeit (Trajektorien) [KKW05, JPPR05] oder die Hüllen von Partikelclustern [LVLRR08] dargestellt werden können. Des Weiteren bietet sich gerade bei Vektorfeldern die Möglichkeit einer Vektorfeldvisualisierung mittels *Line Integral Convolution* (LIC) für eine Schnittebene [CL93, WSEE05] oder direkt als dreidimensionale Darstellung im Raum [BSH96] an.

Für die parallele Visualisierung auf mehr als einem System sind vor allem die Arbeiten von Ma et al. zu nennen, die die Arten parallelen Renderings klassifizieren und effiziente Algorithmen – insbesondere für Volumenrendering mittels Raytraycing – beschrieben haben [MPHK93, MPHK94]. Neben diesen klassischen Ansätzen gibt es jedoch noch andere Herangehensweisen, z. B. indem das verteilte Rendern von Bildern mithilfe einer Modellierung auf Mapping- und Reduceschritte realisiert wird [SCMO10].

2.3 Bisherige Implementierungen

Sowohl für paralleles als auch In-Situ Volumenrendering gibt es schon fertige Softwarelösungen [EP07b]. Am bekanntesten sind dabei die Open Source Lösungen VisIt [CBW+12] und ParaView [CGM+06]. Der große Vorteil derartiger, freier Lösungen ist die relativ einfache Erweiterbarkeit. So gibt es verschiedene In-Situ Erweiterungen für beide [RCMS11]. Für VisIt gibt es eine In-Situ Bibliothek namens libsim. Simulationen müssen gegen diese Bibliothek gelinkt werden und verschiedene Funktionen aufrufen, um über die zu visualisierenden Daten zu informieren [WFM11]. Bei ParaView wird die Erweiterung ParaView Coprocessing sogar direkt mitgeliefert. Sie ermöglicht es Simulationsdaten zu filtern und zu komprimieren bevor sie gespeichert werden, um sie später in einem Postprocessingschritt zu visualisieren [FMT+11]. Des Weiteren gibt es eine alternative Erweiterung namens ICARUS, die ausnutzt, dass die meisten Simulationen einen Export in das wissenschaftliche Datenformat HDF5 ermöglichen. Sie bietet einen virtuellen Dateitreiber mit einer zu HDF5 kompatiblen API an. Auf diese Art und Weise muss nur wenig Simulationscode an die Visualisierung angepasst, aber es kann trotzdem ein eng gekoppelter Ansatz realisiert werden [BSO+11].

Neben diesen bibliotheksbasierenden, universellen Lösungen gibt es auch einige sehr eng an bestimmte Simulationen gekoppelte Ansätze. Der Vorteil liegt hier darin, dass ein etwaiges Konvertieren oder Kopieren der Simulationsdaten für API-Konformität entfällt [Ma95, NGYD10]. Insbesondere für Simulationen, die *Rechenbeschleuniger* wie GPUs nutzen, können diese auch für die Visualisierung benutzt werden und es müssen keine Simulationsdaten von den Rechenbeschleunigern heruntergeladen werden, um sie an Visualisierungsfunktionen zu übergeben [SVS13, Sch13].

3 Grundlagen

Das Mooresche Gesetz besagt, dass sich die Leistung von Prozessoren alle ein bis zwei Jahre verdoppelt. Diese auf Beobachtung und Erfahrung basierende Gesetzmäßigkeit galt ungefähr bis zur Jahrtausendwende. Seitdem zeigt sich, dass sich CPUs aus physikalischen Gründen nicht mehr beliebig durch Erhöhung der Taktfrequenz beschleunigen lassen. Stattdessen wird verstärkt auf Parallelität gesetzt. Quad- oder gar Octa-Cores in modernen Bürorechnern sind keine Seltenheit mehr. Gerade für verschiedene, parallel ausgeführte Anwendungen wie Browser, Officesuite oder Mediaplayer bietet diese Verteilung der Leistung auf mehrere Einheiten große Vorteile.

Anders sieht es jedoch bei einzelnen Anwendungen aus, die alleine viel Rechenleistung benötigen. In der Vergangenheit wurde probiert, dieses Problem mithilfe von Vektorrechnern zu lösen. Diese unterstützen sogenannte *Single Instruction Multi Data* (SIMD) Instruktionen, welche denselben Befehl auf mehrere Daten anwenden können. Jedoch sind auch solchen Rechnern Grenzen gesetzt, weshalb seit Anfang der 90ern Superrechner zunehmend aus mehreren unabhängig arbeitenden Prozessoren bestehen.

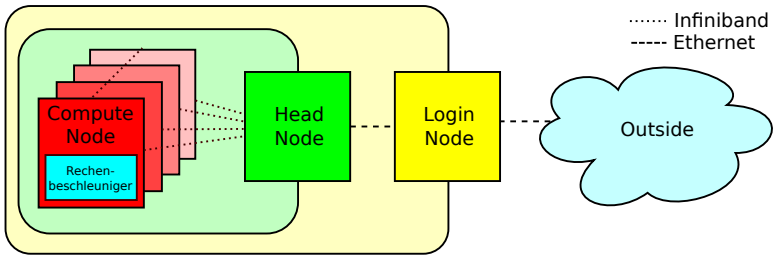

Abbildung 3.1: Exemplarischer Aufbau eines Hochleistungsrechenclusters. Herausforderungen ergeben sich bei den abgeschotteten (hier hellgelb und hellgrün markierten) Subnetzen und der potentiell langsameren Netzwerkverbindung ab dem Head Node.

3.1 Supercomputer 2016

Heutige Hochleistungsrechner bestehen deshalb aus vielen über ein Hochleistungsnetzwerk verbundene Einzelrechner. In Abbildung 3.1 ist exemplarisch der Aufbau eines heutigen, typischen Rechenclusters dargestellt. Die *Compute Nodes* oder *Rechenknoten* sind die hoch parallelen Recheneinheiten. Untereinander sind sie mit Hochgeschwindigkeitsnetzwerken wie InfiniBand ausgestattet. Auch die Verbindung zum *Head Node* bzw. *Kopfknoten* erfolgt meistens über InfiniBand. Der Head Node steuert die Verteilung der Aufgaben auf die Rechenknoten. Er und die Compute Nodes hängen am selben Hochgeschwindigkeitsnetzwerk, um gleichermaßen auf verteilte Dateisysteme zugreifen zu können, die einerseits Simulationsdaten als auch die Simulationsprogramme beinhalten. Ein direkter Zugriff auf den Head Node, z. B. aus dem Internet, ist für gewöhnlich nicht möglich. Stattdessen erfolgt der Zugriff von einem *Login Node*, welcher von außen erreichbar ist. Die Verbindung zwischen Head Node und Login Node und erst Recht zwischen Login Node und Außenwelt ist jedoch mit klassischem Ethernet LAN implementiert, welches innerhalb von Forschungseinrichtungen noch im Gigabit-, aber spätestens über das Internet nur noch im 100-MBit-Bereich liegen wird. Dieses Netzwerkbottleneck muss bei dem Entwurf einer In-Situ Anwendung, die Daten nach außen weitergibt und u.A. weit entfernt ausgewertet werden soll, beachtet werden.

In-Transit Processing ergibt also nur dann Sinn, wenn die Weiterverarbeitung innerhalb des in Abbildung 3.1 grün markierten Hochgeschwindigkeitsnetzwerkbereiches erfolgt, andererseits würde die Netzwerkgeschwindigkeit die Simulation zu stark ausbremsen. Eine große Herausforderung für In-Situ Processing ist also Daten innerhalb des Rechenclusters derartig zu filtern und zu komprimieren, dass ein Abspeichern auf relativ langsame Festplattenspeicher oder eine Übertragung über das Ethernet die Simulation nicht ausbremst oder eine unvermeidbare Ausbremsung wenigstens mit steigender Knotenanzahl skaliert, damit Exascalesysteme die zusätzliche Leistung ausnutzen können.

Softwareseitig wird zur Kommunikation der Rechenknoten untereinander meistens das *Message Passing Interface* (MPI) genutzt. MPI ist ein Standard sowohl für Punkt zu Punkt Kommunikation als auch für Gruppenkommunikation. Die Implementierung des Standards kann je nach Rechnersystem hochoptimiert erfolgen. Im Allgemeinen ist MPI wesentlich leistungsfähiger als klassische, socketbasierte Netzwerkverbindungen, auch wenn das zugrunde liegende Netzwerk das gleiche ist. Zwischen Head Node, Login Node und Außenwelt werden jedoch weiterhin klassische TCP oder UDP Socketverbindungen genutzt.

3.1.1 Rechenbeschleuniger

Eine weitere Möglichkeit, um mehr Rechenleistung nutzen zu können, ist, für spezielle Aufgaben gesonderte Hardwareeinheiten zu verbauen, welche Last von dem universellen Hauptprozessor nehmen – z. B. für Audioausgabe oder Grafik. Für die letzte Kategorie entwickelte man *Graphic Processing Units* (GPUs). Anfangs waren diese nur in der Lage schnell große, rechteckige Speicherbereiche zu kopieren, sogenanntes *Blitting*. Mit der Zeit wurden diese jedoch um Rasterungs- und Texturierungsmöglichkeiten ergänzt, um 3D Grafik zu ermöglichen. Heutige GPUs sind vollwertige, hoch parallele CPUs und werden deshalb nicht nur für die Computergrafik, sondern mittlerweile auch für wissenschaftliche Simulationen oder rechenintensive Aufgaben wie Videokodierung eingesetzt.

GPUs sind im Vergleich zu klassischen CPUs zwar niedriger getaktet, können dafür jedoch im besten Fall über tausend Instruktionen parallel ausführen. Als die freie Programmierung möglich wurde, erfolgte sie anfangs über Shader, welche stark auf die Nutzung zur Grafikbeschleunigung ausgelegt sind. Mittlerweile gibt es abstraktere Programmierungsmethoden wie OpenCL oder CUDA. Bei letzterer kann normaler C oder C++ Code über CUDA spezifische Schlüsselwörter erweitert werden, die Aussagen darüber treffen, ob ein Programmteil auf der CPU (genannt *Host*), der GPU (genannt *Device*) oder beiden ausgeführt werden soll. Der CUDA C++ Compiler erzeugt daraus eine Binärdatei, welche Bytecode für Host und Device enthält und wie eine klassische ausführbare Datei gehandhabt werden kann.

Da sich GPUs als hoch parallele, universelle Rechenbeschleuniger bewährt haben, gibt es auch Entwicklungen, die ohne Bezug zur Computergrafik einen Hardwarebeschleuniger implementieren, z. B. Intels Xeon Phi Produktreihe. Statt wie bei Multicorearchitekturen auf 4 bis 16 hoch getaktete Prozessorkerne zu setzen, werden hier ähnlich einer GPU über 50 vollwertige Prozessorkerne kombiniert. Da die zugrunde liegenden Kerne zudem der x86 Architektur angehören, lassen sich die Erfahrungen der Multicoreprogrammierung auch auf diese *Manycoresysteme* übertragen. Ein weiterer Vorteil ist, dass sie sich mithilfe von OpenMP programmieren lassen und kein Extraframework wie CUDA mehr benötigen. Die kommende, dritte Generation lässt sich sowohl als dedizierter Rechenbeschleuniger als auch als Haupt-CPU nutzen. In diesem Fall fällt die Notwendigkeit weg, zwischen Host- und Devicespeicher kopieren zu müssen [Sau16]. Ein weiteres Problem ist, dass CUDA zwar schneller als OpenCL ist [KDH10, FVS11, SCL+12], aber nur von Nvidia unterstützt wird. Hinzu kommt, dass Nvidia nur das veraltete OpenCL 1.2 unterstützt, obwohl die Spezifikationen für OpenCL 2.0 seit Ende 2013 definiert sind und von Konkurrenten wie AMD auch schon voll unterstützt werden.

Die beiden schnellsten Supercomputer der Welt Tianhe-2 (National Super Computer Center in Guangzhou, China) und Titan (Oak Ridge National Laboratory, USA) erreichen ihre LinPACK-Peakleistungen von $\approx$ 33,9 bzw. $\approx$ 17,6 PetaFLOPS durch den Einsatz von Rechenbeschleunigern [SDSM16]. Aufgrund der Peakleistung im PetaFLOPS-Bereich spricht man von Petascalesystemen. Tianhe-2 besteht dabei aus $\sim$ 16 000 Rechenknoten, Titan aus über 18 000. Das chinesische System nutzt pro Rechenknoten drei Intel Xeon Phis mit je 57 Einzelprozessoren zur Beschleunigung, wohingegen der US-amerikanische Cluster Nvidia Grafikkarten mit je über 2 500 Prozessoren pro Rechenknoten einsetzt. Damit ergibt sich sowohl das Problem Algorithmen zu implementieren, die auf so vielen unabhängig arbeitenden Prozessoren skaliert werden können, und andererseits, die entstehenden Daten weiter zu verarbeiten, indem sie z. B. auf Festplatte gespeichert, über das Netzwerk geschickt oder vor Ort weiterverarbeitet werden. Nur wenn diese Probleme effizient von den Computerwissenschaften gelöst werden können, ist es möglich in Zukunft Supercomputer im Exascalebereich voll auszunutzen.

3.1.2 Hochparallele Anwendungen

Supercomputer werden für viele verschiedene wissenschaftliche Aufgaben benötigt. Es ist z. B. nicht möglich die meisten Observablen astronomischer Effekte direkt zu messen. Auch auf der Erde realisierbare Vorgänge können riskant, teuer oder zeitaufwendig sein. Die Lösung sind Modelle, die die zu beobachtenden Effekte hinreichend beschreiben und auf Superrechnern berechnet werden können. Aber auch für experimentell nachvollziehbare Phänomene bieten Hochleistungsrechner die Möglichkeit die dahinter vermuteten und beschriebenen Modelle zu validieren, wenn sie analytisch nicht zu lösen sind, z. B. weil sie von einer sehr hohen Teilchenanzahl ausgehen.

Des Weiteren werden Hochleistungsrechner in der Zukunft mehr und mehr dafür genutzt werden, hochparallele Datenanalysen für wissenschaftliche Sensordaten mit hoher Auflösung und Frequenz durchzuführen. Heutzutage dominieren jedoch noch künstliche Hochfrequenzdatenquellen in Form von wissenschaftlichen Simulationen.

Eine solche wissenschaftliche Simulationsmethode sind elektromagnetische *Particle in Cell* (PIC) Algorithmen. In diesen werden zwei Domänen betrachtet: Felder sowie Teilchen. Die Teilchen werden durch ihre Position, Geschwindigkeit und u. U. noch weitere Merkmale wie z. B. Ladung oder Masse beschrieben. Felder werden über ein Gitter approximiert. In jedem Zeitschritt passieren vier Schritte, es werden

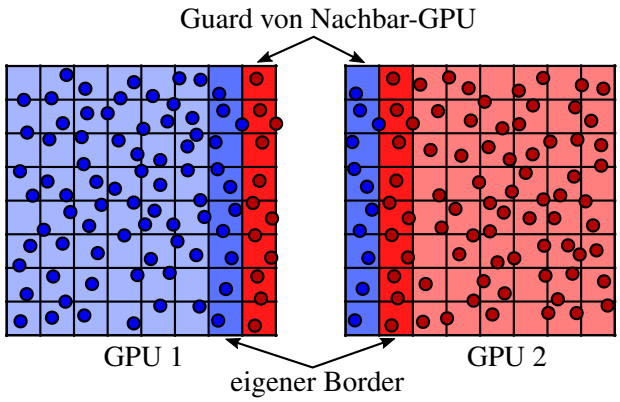

Abbildung 3.2: Überlappung von Gitter und Partikeldaten zwischen zwei GPUs der PIConGPU Simulationssoftware, um auf Daten von Nachbarknoten zugreifen zu können, ohne explizit mit diesen zu kommunizieren, was hohe Wartezeiten zur Folge hätte. Der Borderbereich einer GPU ist identisch mit dem Guardbereich des Nachbarn.

- aus den komplett berechneten und in den Gittern gespeicherten Feldern die Kräfte auf die Partikel berechnet,
- daraufhin die Teilchen um die berechnete Kraft im Raum bewegt,
- aus den Partikeln die Änderung des Stromes berechnet und
- daraus zuletzt die Änderungen der magnetischen und elektrischen Felder pro Gitterpunkt ermittelt.

PIConGPU Das am Helmholtz-Zentrum Dresden - Rossendorf (HZDR) entwickelte PIConGPU implementiert solch einen PIC Algorithmus für CUDA programmierbare GPUs [BBC+13, BWH+10]. Dazu wird das Gesamtgitter auf die zur Verfügung stehenden GPUs aufgeteilt und jede GPU berechnet lokal den PIC Algorithmus. Des Weiteren kümmert sich PIConGPU darum, Partikel, die ein Subgitter verlassen, an die entsprechenden GPUs weiterzuleiten. Ein Hilfsmittel hierfür sind sogenannte *Borders* und *Guard*s. Die Borders sind die äußersten Bereiche des Gitters, die von der GPU noch selbst berechnet werden. Daneben hält jede GPU jedoch noch die Borders der Nachbar-GPUs im Speicher. In der Abbildung 3.2 werden blaue Partikel und das blaue Gitter nur auf GPU 1 und rote Partikel und das rote Gitter nur auf GPU 2 berechnet. Trotzdem befindet sich eine Kopie des Borders jeweils im Guard der anderen GPU. Das erzeugt zwar einen geringen Speicheroverhead, ermöglicht aber das komplette Abarbeiten eines Zeitschrittes ohne dass auf die Ankunft von etwaigen Partikeln aus anderen GPUs gewartet werden muss. Indem die Berechnung von innen nach außen erfolgt, kann die Zeit bis der Border

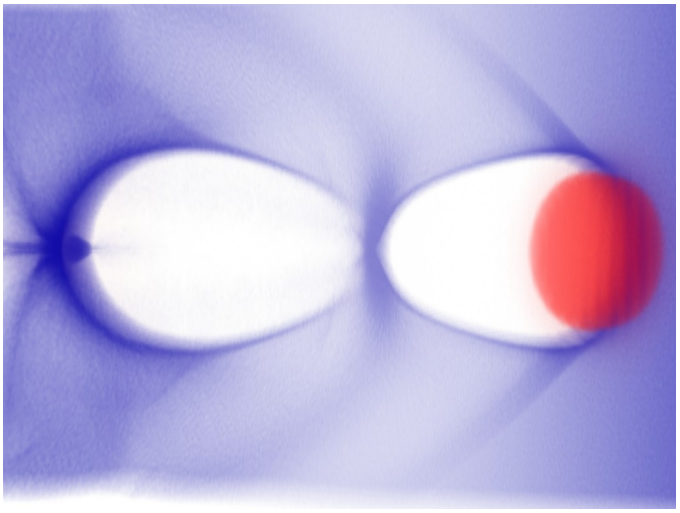

Abbildung 3.3: Visualisierung einer Laser-Kielfeld-Beschleuniger-Simulation. Der Laserpuls ist rot gezeichnet, Elektronendichte blau. Es fallen sowohl die kugelförmigen Bereiche niedriger Elektronendichte als auch die bugwellenartigen Verdrängungen von Elektronen direkt hinter dem Laserpuls auf.

vom Algorithmus betrachtet wird, genutzt werden, sie mit den Nachbarn auszutauschen.

Viele verschiedene Plasmasimulationen sind mithilfe von PIConGPU realisierbar. Ein Plasma beschreibt dabei ein Gemisch geladener Teilchen. Das bedeutet, dass Elektronen und Ionen voneinander getrennt sind und sich frei bewegen können, weshalb Plasmen u. a. elektrischen Strom leiten. Natürliche Plasmen sind Blitze oder Polarlichter, aber auch Sterne und astronomische Gaswolken. Generell ist die meiste Materie des Universums ionisiert, also im Plasmazustand. Man unterscheidet Plasmen bezüglich der Teilchendichte, des Drucks als auch der Temperatur. PIConGPU eignet sich vor allem für die Simulation von Plasmen mit niedriger Temperatur und Dichte, da Teilchenkollisionen so vernachlässigt werden können. Exemplarisch sollen hier drei vorgestellt werden.

Laser-Kielfeld-Beschleuniger Bei einem Laser-Kielfeld-Beschleuniger (*Wakefield*) [MFL$^+$02, FGP$^+$04, MMN$^+$04] wird ein Laserpuls in ein optisch unterdichtes Gas geschossen. Das bedeutet, dass der Lichtpuls das Gas durchdringen kann und nicht reflektiert wird, ähnlich wie das sichtbare Licht Glas durchdringen kann. Für sichtbares Licht ist Glas optisch unterdicht, für Röntgenstrahlung wäre z. B. der menschliche Körper optisch unterdicht. Da ein Laserpuls

nichts anderes als ein elektromagnetischer Puls ist, werden Elektronen von diesem beim Eintritt in das Gas abgestoßen. Der Laserpuls wird durch das Medium leicht abgebremst, besitzt aber immer noch fast Lichtgeschwindigkeit. Die sehr leichten Elektronen können deshalb abgelenkt werden, die sehr viel schwereren Ionen in dieser kurzen Zeit jedoch nicht. Dadurch wird das Gas lokal ionisiert, geht also in einen Plasmazustand über. Direkt hinter dem Puls, sozusagen in dessen Kiel, entsteht ein Bereich mit niedriger Elektronendichte und somit sehr hoher Ionendichte.

Abbildung 3.3 zeigt eine Visualisierung aus PIConGPU. Der Laserpuls ist rot dargestellt, hohe Elektronendichten sind blau eingefärbt. Neben dem schon beschriebenen, kugelförmigen Bereich hinter dem Puls, in dem kaum Elektronen verblieben sind, fällt eine bugwellenähnliche Verdrängung von Elektronen hinter dem Laserpuls auf. Hinter der Kugel mit hoher Ionenkonzentration, sammeln sich Elektronen. Diese wiederum erzeugen einen ähnlichen Effekt, wie der Laserpuls und es entstehen periodisch abgeschwächt immer mehr solcher Kielfelder. In den kugelförmigen Bereichen entsteht ein sehr starkes elektrisches Feld, welches Elektronen vorne abbremst und hinten stark beschleunigen würde. Gegenstand der Forschung ist es deshalb in den hinteren Teil dieses Feldes Elektronen zu injizieren, welche dann auf sehr kleinen Raum sehr viel stärker beschleunigt werden können als es mit klassischen, sehr großen Teilchenbeschleunigern bisher möglich ist. Kompakte Teilchenbeschleuniger ähnlich diesem sind u. a. wichtig für moderne und genauere Formen der Strahlentherapie zur Behandlung von Krebsgeschwüren.

Plasmainstabilitäten Fließen zwei Plasmen mit unterschiedlicher Geschwindigkeit, entstehen durch minimale Inhomogenitäten höhere Elektronen- und Ionendichten und somit kleine lokale Ladungstrennungen. Da die Ladungen jedoch bewegt sind, induzieren deren Ströme wiederum Magnetfelder. Durch diese Magnetfelder wirkt die Lorentzkraft auf die bewegten Ladungsträger und separiert sie weiter, wodurch wiederum noch stärkere Magnetfelder entstehen. Diesen sich immer weiter verstärkende Vorgang nennt man *Zwei-Strom Instabilität*.

Eine besondere Form dieser Instabilität ist die *Weibel-Instabilität*, bei der Plasmen unterschiedlicher Temperatur betrachtet werden. Da Temperatur die ungerichtete Bewegung von Teilchen angibt, kann eine Weibel-Instabilität als die Überlagerung vieler Zwei-Strom-Instabilitäten gesehen werden.

Wenn zwei Strömungen nebeneinander mit unterschiedlichen Geschwindigkeiten oder sogar entgegengesetzt fließen, entstehen an den Übergangsflächen kleine Verwirbelungen und daraus immer stärker werdende und sich selbst verstärkende Strömungen, die man *Kelvin-Helmholtz Instabilitäten* nennt. Ein bekanntes Beispiel ist

z. B. die Verwirbelung des Rauchs eines schon länger brennenden Räucherstäbchens, obwohl sich die Umgebungsluft nicht bewegt.

All diese Instabilitäten lassen sich sowohl in Laborversuchen als auch astrophysikalischen Prozessen beobachten. Wenn die Prozesse solcher Instabilitäten mithilfe von PIC Simulationen besser verstanden werden können, ist es möglich mithilfe des elektromagnetischen Spektrums, das von terrestrischen Teleskopen aufgenommen wird, Rückschlüsse auf die Zusammensetzung und das Verhalten astronomischer Phänomene zu ziehen [BBC+13].

3.1.3 Verteilte Visualisierung

Heutige Volumenvisualisierungen sind für gewöhnlich durch die Rechenbeschleuniger von sich aus schon hoch parallel. Die Verteilung der GPUs auf verschiedene Rechner stellt jedoch noch einmal neue Herausforderungen an die Algorithmen und Arbeitsabläufe.

Es ist unabdingbar, dass sich die Knoten über ihre zu visualisierenden Daten austauschen. Je nach Zeitpunkt und Art dieses Austausches unterscheidet man drei Paradigmen. Beim *Sort-First* Ansatz wird jedem Knoten ein Bereich des endgültigen Bildes zugewiesen und die zugrunde liegenden Primitive oder Volumendaten nur so weit betrachtet, um herauszufinden, für welchen Knoten sie relevant sind, und um sie so zu verteilen, dass jeder Renderer alle seine benötigten Daten hat. Das Projizieren und Rastern erfolgt dann jeweils lokal und die fertigen Teilbilder können trivial zusammengesetzt werden.

Beim *Sort-Middle* Ansatz werden für alle Primitive alle Schritte bis kurz vor die Rasterung ausgeführt. Dann wird ähnlich zum Sort-First Ansatz entschieden, welcher Renderer zuständig ist, die projizierten Primitive entsprechend ausgetauscht und wie bei Sort-First geendet.

Sort-Last geht einen anderen Weg. Hierbei rendert jeder Knoten die Daten, auf die er Zugriff hat. Erst nach dem Rastern werden die kompletten Teilbilder ausgetauscht und beim sogenannten *Compositing* zu einem Gesamtbild kombiniert [MCEF94].

Da die Arbeit von Peta- oder gar Exascale Anwendungen ausgeht, ist es nicht praktikabel die zu rendernden Volumendaten vor dem Rendern an andere Knoten zu schicken. Deshalb bietet sich hier nur der Sort-Last Ansatz an. Gerade wenn einzelne Knoten einen konvexen Teil des Gesamtvolumens simulieren und später auch visualisieren, ist die für Sort-Last dabei notwendige Sortierung der Subvolumen für die Kombination der Teilbilder trivial [MPHK93, LRN96, FCS+10, MSE06].

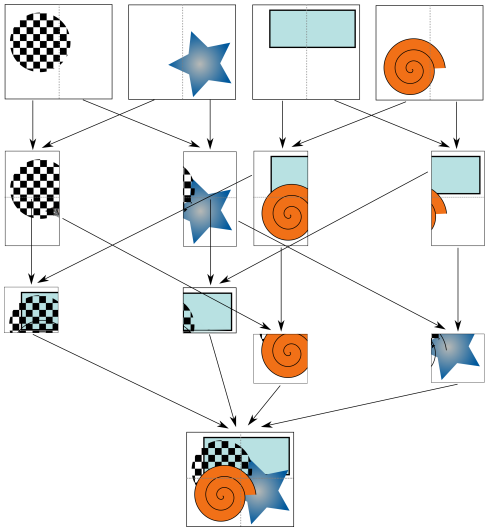

Abbildung 3.4: Binary Swap Algorithmus für vier Knoten. Immer zwei Knoten tauschen einen immer kleiner werdenden Teil der Bilder aus und kombinieren diese verteilt, wodurch sowohl die Knoten als auch das Verbindungsnetzwerk gleichmäßig belastet werden.

Die effizienteste Methode des Bilderaustauschs unter den Knoten hängt von der generellen Verteilung der Bildgrößen und der Anzahl der Knoten ab, auf denen am Ende ein fertiges Bild erzeugt werden soll. Der einfachste Ansatz ist, dass jeder Knoten sein fertiges Teilbild an jeden Knoten schickt, der das Gesamtbild bekommen soll, damit diese es dann lokal zusammensetzen können. Diese Methode nennt sich *Direct Send* [EP07a]. Wenn alle Knoten ihre Teilbilder dabei an wenige Anzeigeknoten schicken, belastet das einerseits das Netzwerk einseitig und andererseits lässt es nur diese zentralen Knoten die Bilder kombinieren. Deshalb gibt es verschiedene Konzepte die Netzwerk- und Rechenlast unter allen Rechnern aufzuteilen.

Eine Methode stellt *Binary Swap* dar. Hierbei tauschen immer zwei Knoten ihre bisher zusammengefügten Bilder aus, bis jeder Knoten einen Teil des Gesamtbildes enthält. Die Menge der zu übertragenden Daten halbiert sich dabei in jedem Schritt. Als letzter Schritt werden die kompletten Teilbilder an die Knoten geschickt, die sie für die Weiterverarbeitung benötigen (siehe Abbildung 3.4) [MPHK94]. Der Nachteil dieser Methode ist, dass einige Knoten nicht ausgelastet werden, wenn deren Anzahl keine Zweierpotenz ist. Yu et al. beschreiben eine Erweiterung für Binary Swap, die mithilfe von Direct Send die Auslastung für eine beliebige Anzahl an Knoten optimiert [YWM08].

Moreland et al. zeigten, dass der Algorithmus auch dann performant arbeitet, wenn mehrere Knoten nur Teilausschnitte des Gesamtbildes brauchen, z. B. um die Visualisierung auf mehreren Bildschirmen darzustellen [MWP01].

Für verhältnismäßig kleine (~1 GB) Datensätze lassen sich die Visualisierungen auch auf einem einzelnen Multi-GPU-System realisieren. In diesem Fall fällt Netzwerkkommunikationsaufwand weg und Bilder können direkt im Grafikkartenspeicher kombiniert werden [MMD08]. Wie in Abbildung 3.4 zu sehen ist, sind die (Teil-)Bilder oft nicht voll besetzt. Zur weiteren Netzwerklastreduzierung und somit Beschleunigung des Kombinationsprozesses bietet sich deshalb eine Komprimierung vor der Übertragung und ein Anpassen der Compositealgorithmen auf komprimierte Daten an [AP98]. Noch weiter lässt sich dieser Prozess mit dedizierter Compositing-Hardware beschleunigen [NKS+04].

Viele der oben genannten Ansätze wurden performant in der quelloffenen IceT Bibliothek [San16b] der Sandia National Laboratories implementiert und insbesondere hervorragend dokumentiert [San16a], weshalb sie auch für diese Arbeit genutzt wurde. Auch die Visualisierungslösung Paraview nutzt diese Bibliothek für ihr paralleles Rendering.

3.2 Volumenvisualisierung

Für wissenschaftliche Simulationen und für PIC Simulationen im besonderen bieten sich verschiedene Visualisierungstechniken an. Zum Beispiel wäre eine Animation der Feldbewegungen mithilfe von Line Integral Convolution oder ein direktes Zeichnen der Partikel – u. U. mit Trajektorie – denkbar. Da die Dichte von Partikeln jedoch trivial in ein Volumen konvertiert werden kann, sollen im Rahmen dieser Arbeit die Volumenvisualisierungen mittels Raycasting im Vordergrund stehen.

Beim Raycasting wird für jeden Bildpunkt der Sichtstrahl berechnet, der für den/die BeobachterIn bei einer perspektivischen Projektion im Volumen entsteht. In Abbildung 3.5 sieht man für einen Pixel den Sichtstrahl vom Auge durch das Volumen angezeigt. Da jeder Strahl unabhängig von allen anderen Sichtstrahlen ist, kann solch eine Aufgabe sehr gut mit hochparallelen Rechenbeschleunigern berechnet werden. Auf Basis dieser Strahlen können die Strecken im betrachteten Volumen berechnet werden, für welche der Raycast Algorithmus angewendet werden soll. Dieser tastet das Volumen von vorne nach hinten ab und erzeugt so die an den Pixeln zu sehenden Farben.

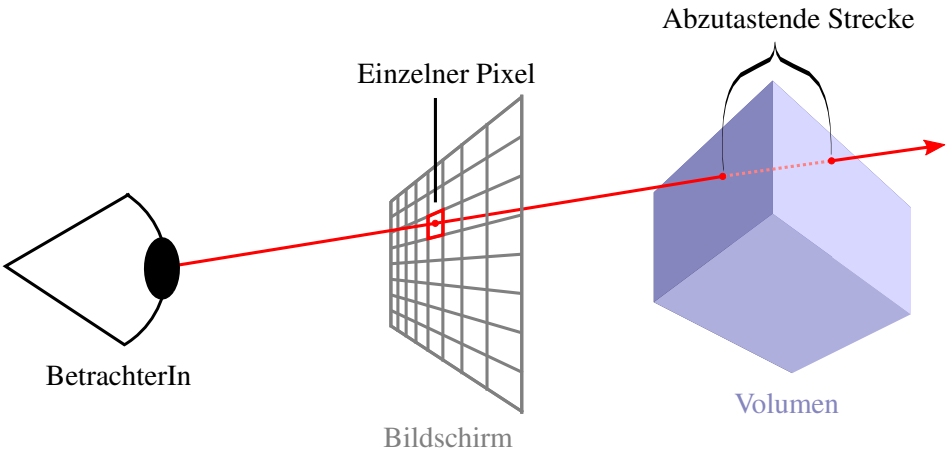

Abbildung 3.5: Darstellung eines Sichtstrahls für den Raycast Algorithmus. Nur die abzutastende Strecke in den violetten Volumendaten ist für den exemplarisch gezeigten Pixel relevant.

Klassifikation Um aus den Einzelabtastwerten Farben berechnen zu können, ist zunächst eine Klassifikation der Daten notwendig. Dabei wird jedem Wert (meist Skalarwert), der im Volumen vorkommt, eine Farbe und eine Opazität zugewiesen. Diese *Transferfunktion* wird meistens diskret als Array angegeben. Auf GPUs bieten sich hierfür Texturen an, die zusätzlich noch eine Interpolation vornehmen können. Werte größer oder kleiner dem Diskretisierungsbereich werden auf den größten bzw. kleinsten Wert gesetzt.

Emissions/Absorptions-Modell Das Emissions/Absorptions-Modell geht davon aus, dass jeder Abtastpunkt gleichermaßen Licht aussendet (emittiert) und weiter hinter liegende Abtastpunkte verdeckt, also deren Licht absorbiert. Dabei wird nicht davon ausgegangen, dass Licht reflektiert oder gebrochen wird, es wird stattdessen einfach ausgelöscht. Des Weiteren geht dieses einfache Modell davon aus, dass das leuchtende Volumen die einzige Lichtquelle ist und nicht von außen noch welches eindringt. Für diese Darstellung als leuchtendes Gas ergibt sich das Volumenintegral

$$I = \int_0^D \text{color}\left(x\left(\lambda\right)\right) \mathrm{e}^{-\int_\lambda^D \text{extinction}(x(\lambda'))d\lambda'} d\lambda$$

nach [EKE01] mit

- color(x), der klassifizierten Farbe an der Position x,

- extinction(x), der klassifizierten Auslöschung an der Position x,
- D, der Abtaststrecke des Integrals,
- λ, dem Abtastparameter des Integrals, und
- $x(\lambda)$, der Position auf der Abtaststrecke für den Abtastparameter λ.

Dieses Integral lässt sich iterativ mit der sogenannten *Blendingvorschrift* approximieren, indem der Sichtstrahl von vorne nach hinten mit der Abtastschrittweite d durchitertiert wird:

$$\begin{aligned} \dddot{C}_{i+1} &\rightarrow \dddot{C}_i + \left(1 - \dot{O}_i\right) c_{i+1}\alpha_{i+1}d \\ \dot{O}_{i+1} &\rightarrow \dot{O}_i + \left(1 - \dot{O}_i\right) \alpha_{i+1}d \end{aligned} \tag{3.2.1}$$

mit $\dddot{C}_0 = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^{\mathrm{T}}$ und $\dot{O}_0 = 0$.

Mithilfe des Emissions/Absorptions-Modell können mit einer guten Klassifikation viele Details eines Volumens erkundet werden. Wenn man z. B. ein Datenvolumen eines CT-Scanners derartig visualisiert, kann man verschiedene Dichten unterschiedlich einfärben und deren Durchlässigkeit anpassen. Es wäre beispielsweise denkbar, Blutbahnen rot, Knochen weiß und Muskelgewebe stark transparent rosa darzustellen.

Isoflächendarstellung Jedoch möchte man ein Volumen nicht immer mithilfe eines leuchtenden Gases darstellen. Bezogen auf oben genanntes Beispiel wäre es z. B. auch interessant die Oberfläche von den Blutbahnen abzubilden. Dies leistet die Isoflächendarstellung. Anstatt sukzessive die Klassifikationen aufeinander zu blenden, läuft der Sichtstrahl solange durch das Volumen, bis eine gesuchte Klassifikation gefunden wurde. Diese wird dann direkt gezeichnet, eine etwaige Opazität wird nicht beachtet. Dadurch ergeben sich Bilder von (bei stetigem Wertebereich geschlossenen) Oberflächen.

In diesem Fall kann es aber schwer werden, die Sortierung der Einzelflächen im Bild zu erkennen. Solange das Bild in Bewegung, z. B. einer Rotation, ist, kann man noch erahnen, welche Strukturen vor anderen liegen. Spätestens bei der Betrachtung eines Standbildes wird das sehr schwer. Deshalb gibt es verschiedene Methoden die Sortierung auszudrücken, wie z. B. Halo, eine leichte Umrandung von Strukturen, ähnlich den schwarzen Außenlinien in Comiczeichnungen, oder das Verblassen der Farbe bei weiter entfernten Objekten.

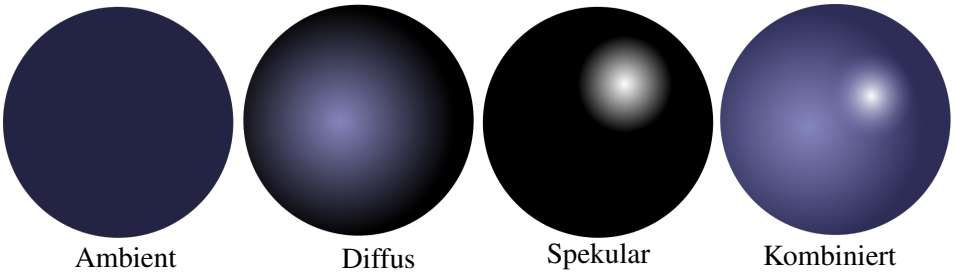

Abbildung 3.6: Aufspaltung der Lichtberechnung in drei Komponenten nach Phong. Durch die Approximation in ambientes, diffuses und spekulare Lichtanteile lässt sich ein hinreichend realistisches Beleuchtungsmodell etablieren.

Beleuchtung Mithilfe des Gradienten an einem Abtastpunkt kann auch eine Oberflächennormale approximiert und darauf basierend eine Beleuchtung implementiert werden. Ein einfaches Beleuchtungsmodell ist das Phong-Beleuchtungsmodell [Pho75], in welchem die Beleuchtung in drei Komponenten unterteilt wird,

- dem ambienten Anteil, welcher einer konstanten Hintergrundstrahlung entspricht, die aus allen Richtungen gleichzeitig scheint,
- dem diffusen Anteil, welcher nur in Abhängigkeit zu der Richtung einer Lichtquelle steht, und
- dem spekularen Anteil, welcher gleichermaßen von der Position der Lichtquelle als auch der Kamera abhängig ist.

Abbildung 3.6 zeigt die drei Komponenten einzeln und in Kombination. Es ergibt sich die Formel

$$I_{\text{out}} = I_{\text{a}} k_{\text{ambient}} + I_{\text{in}} k_{\text{diffus}} \cos\varphi + I_{\text{in}} k_{\text{spekular}} \frac{n+2}{2\,\pi} \cos^n \theta \tag{3.2.2}$$

mit

- dem gesamten in Richtung der Kamera emittierten Licht und damit der zu sehenden Farbe I_{out},
- der konstanten, ambienten Hintergrundbeleuchtung I_{a},
- der Intensität und Farbe der Lichtquelle I_{in},
- den material- und lichtkomponentenabhängigen Reflexionsfaktoren k_{ambient}, k_{diffus} und k_{spekular},
- dem Winkel zwischen Lichtquelle und Oberflächennormale φ,

- dem Winkel zwischen Kamera und Oberflächennormale θ und
- dem Shininess-Exponenten n, der bestimmt, wie glänzend bzw. spiegelnd die Oberfläche erscheint.

Wenn mehrere Lichtquellen in einer Szene vorhanden sind, gibt es entsprechend mehr $I_{\text{in}} k_{\text{diffus}} \cos\varphi + I_{\text{in}} k_{\text{spekular}} \frac{n+2}{2\pi} \cos^n \theta$ Terme, die hinzuaddiert werden, womit sich ein Problem des Modells für komplexe Beleuchtungen zeigt: Wenn viele Lichtquellen aktiviert sind, ist der Einfluss einzelner Lichtquellen nur noch schwer auszumachen und das gesamte Objekt wirkt stark überblendet. Für die Isoflächendarstellung ist dies jedoch nicht von Relevanz, da in ISAAC nur von einer Lichtquelle ausgegangen wird.

Verteiltes Volumenraycasting Es wurde gezeigt, dass das Emissions/Absorptions-Modell der klassischen Blending Operation entspricht. Genauso, wie sich die einzelnen Slices hintereinander blenden lassen, lassen sich auch die Gesamtbilder nach eben dieser Vorschrift kombinieren. Wichtig ist dabei, dass die endgültige Opazität $\dot{O}_n$ als Alphawert zusätzlich zu der Endpixelfarbe $\ddot{\dot{C}}_n$ gespeichert wird, damit sie für das Alphablending von IceT verwendet werden kann.

Eine weitere Voraussetzung ist, dass es eine eindeutige Ordnung zwischen allen Subvolumen untereinander gibt, damit von hinten nach vorne geblendet werden kann. Das bedeutet, dass sie konvex und überschneidungsfrei sein müssen. Eine gleichmäßige Verteilung der Volumendaten auf die Knoten ist nicht zwingend notwendig, sorgt jedoch für eine bessere Lastverteilung und somit insgesamt für ein schnelleres Rendering.

Die Isoflächendarstellung lässt sich mit der gleichen Methode parallel ausführen, insofern eine konstante Opazität von 1 angenommen wird. Prinzipiell wäre in diesem Fall auch ein Compositing mittels Z-Test denkbar, womit auch die gerade genannten Bedingungen der Konvexität und Überschneidungsfreiheit nicht mehr notwendig wären. Andererseits müssen die Daten diese Bedingungen für die Darstellung als leuchtendes Gas sowieso erfüllen und andererseits müsste der Z-Buffer zusätzlich mitgeführt werden. Theoretisch sollte es keinen Unterschied machen, ob man einen Alphakanal oder einen Z-Kanal mitführt, aber praktisch wird in Lösungen wie IceT der Alphakanal in jedem Fall mit reserviert, u. a. weil sich ein Bilddatenpunkt mit einer Größe von 4 Byte (je eines pro Kanal) gut vom Prozessor laden und verarbeiten lässt. Der Z-Buffer wäre also in jedem Fall zusätzlicher Aufwand, auch wenn der Alphakanal nicht genutzt wird.

```
1  int max(int a,int b)
2  {
3      if (a > b)
4          return a;
5      return b;
6  }
7
8  // Aufruf mit
9  int a = 42;
10 int b = 23;
11 int m = max(a,b); //42
```

Algorithmus 1: Naive Implementierung einer Maximumsfunktion für den Typen `int`, die nur aufwendig auf andere Datentypen portierbar ist.

3.3 Template Metaprogrammierung

Im Kapitel 2 wurden verschiedene In-Situ Ansätze dargestellt. Grob ließen sie sich in zwei Gruppen unterteilen. Die eine Gruppe schuf sehr simulationsnahe Implementierungen, die stark auf die Eigenheiten der Simulationen eingehen konnten und deshalb wenig Konvertier- und Kopieroperationen benötigten, aber dafür nicht trivial auf andere Simulationen übertragbar waren. Die andere Gruppe von In-Situ Visualisierungen definierte ein allgemeines Interface, z. B. eine C-API oder einen HDF5-Dateisystem-Treiber, mit dessen Hilfe verschiedene Simulationen ihre Daten an die Visualisierung übertragen konnten. In diesen Fällen müssen die Simulationen ihre Daten jedoch an die Visualisierung anpassen. Visualisierungen stellen eine wichtige Erkenntnisquelle für wissenschaftliche Daten dar. Nichtsdestotrotz sollte eine In-Situ Visualisierung die zu Grunde liegende Simulation nicht zu stark ausbremsen und möglichst keine Rechenzeit mit Konvertier- und Kopieroperationen verschwenden.

Dieses Arbeit geht deshalb einen hybriden Weg, der die Vorzüge beider Welten erhalten und die Nachteile vermeiden soll. Notwendig sind dazu *Templates* sowie *Template Metaprogrammierung* (TMP). Durch Templates kann typenunabhängiger Code geschrieben werden, um eine möglichst abstrakte Lösung zu erhalten. Bei der TMP werden Teile des Quelltextes derartig geschrieben, dass sie vollständig zur Kompilierzeit ausgewertet werden können.

3.3.1 Templateprogrammierung

Ein Beispiel: Würde man eine Funktion implementieren wollen, die stets das Maximum der Eingabeparameter zurück gibt, würde man in C++ für den Typ `int` den Algorithmus 1 implementieren.

```
1  template< typename MaxType >
2  MaxType max(MaxType a,MaxType b)
3  {
4      if (a > b)
5          return a;
6      return b;
7  }
8
9  //Aufruf mit
10 int a = 42;
11 int b = 23;
12 int m = max<int>(a,b); //42
```

Algorithmus 2: Template Implementierung einer Maximumsfunktion, die per Definition für beliebige Typen implementiert ist.

Möchte man selbige Funktion nun auch für `float`, `double` und andere Typen implementieren, müsste man den Code kopieren und `int` stets durch den neuen Typ ersetzen. Wenn diese Funktionen komplexer werden und Fehler behoben werden müssen, muss dieser Fehler für jede Kopie einzeln korrigiert werden, was selbst wieder fehleranfällig ist. Mithilfe von Templates kann eine Funktion mit einem abstrakten Typ definiert werden, siehe Algorithmus 2.

Erst bei dem Aufruf von `max()` muss angegeben werden, welcher Typ genutzt werden soll. In diesem Fall war die Angabe sogar optional, da sich `MaxType` indirekt aus dem Typen von `a` und `b` ergibt. Wenn nun ein Fehler auftritt, muss nur diese eine Template Funktion angepasst werden. Des Weiteren kann diese Funktion für jede Art von Typ benutzt werden, der die `<` Relation unterstützt. Das könnte z. B. auch `std::string` sein oder aber eine selbst definierte Klasse, die eine Ordnung definiert. Genau diesen Fakt möchte diese Arbeit ausnutzen. Der Typ der Simulation ist potentiell unbekannt, aber trotzdem wird eine Volumenvisualisierung auf Originaldaten implementiert.

3.3.2 C++ Metaprogrammierung

Das Auflösen von `MaxType` zu `int` in dem Beispiel passiert zur Kompilierzeit. Die erzeugte Anwendung beider Beispiele ist identisch. Mithilfe von Templates lassen sich jedoch nicht nur Typen abstrakt definieren, sondern es konnte gezeigt werden, dass sich eine Turing vollständige Metasprache ergibt [Vel03]. Die C++ Bibliothekensammlung Boost griff diese Idee auf und implementierte eine Template Bibliothek, die eine *Meta Programming Language* (MPL) [GA16] beschreibt, die es einem z. B. ermöglicht, zur Kompilierzeit Listen zu definieren und abzuarbeiten. Eine Template Bibliothek ist eine Bibliothek, die nur inkludiert und nicht gelinkt wird, da die Templateparameter erst zur Kompilierzeit des aufrufenden Programms durch konkrete Typen ersetzt werden können. Statt auf Variablen im Speicher ar-

beiten C++ Template Metaprogramme hierbei auf Typen, die bestimmte statische, konstante Werte wie `value` definieren.

Ein einfaches Beispiel: Boost::MPL bietet einem die Möglichkeit Typen zu definieren, die ganze Zahlen repräsentieren, ähnlich wie es `int` für normale C++ Programme macht:

```
typedef boost::mpl::int_< 42 > TypeA;
typedef boost::mpl::int_< 23 > TypeB;
```

Diese Typen `TypeA` und `TypeB` besitzen nun statische, konstante Werte `TypeX::value`, die auf `42` bzw. `23` gesetzt sind. Ähnlich dem Beispiel von oben bietet die Boost MPL auch eine Maximumsfunktion, die den größeren der beiden Werte zurückgibt:

```
typedef boost::mpl::max< TypeA , TypeB > MaxType;
```

Auf den konkreten Wert kann nun mit `MaxType::type::value` zugegriffen werden. Das Besondere ist, dass die komplette Auswertung, welche der beiden Zahlen größer ist, zur Kompilierzeit erfolgt, unabhängig von den Optimierungen des C++ Compilers.

Da man durch `typedef` definierte Typen in C++ zu einem späteren Zeitpunkt nicht umdefinieren kann, sind einmal gesetzte Werte unveränderlich. Damit ist die C++ TMP eine rein funktionale Programmiermethode – im Gegensatz zu C++ selbst. Die unterschiedlichen Programmierparadigmen von Sprache und Metasprache müssen aber in Kauf genommen werden, da das Gros des performanten Simulationscodes, für den diese Arbeit eine In-Situ Bibliothek bereitstellen möchte, in C bzw. C++ geschrieben ist. Aus dem funktionalen Charakter folgt aber auch, dass Schleifen ausschließlich über Rekursion realisierbar sind. Hierfür werden rekursiv Instanzen derselben Struktur erzeugt bis eine Spezialisierung den Rekursionskopf angibt. Algorithmus 3 zeigt ein einfaches Beispiel.

Die Boost::MPL arbeitet ausschließlich auf Typen, wohingegen klassische C++ Programme auf Variablen arbeiten. Boost::Fusion [GMS16] ist eine Erweiterung von Boost::MPL, die diese beiden Welten miteinander verbindet. Wenn man in Boost::MPL z. B. eine Typenliste erstellt, kann man nur über diese Typen iterieren, sie aber nicht instanziieren. Eine std::list wiederum hat den Nachteil, dass erst zur Laufzeit iteriert wird, was Performanceeinbußen mit sich bringen kann. Mit Boost::Fusion ist es möglich, eine Liste von Typen zu erstellen, diese zu instanzieren und dann zur Kompilierzeit über diese Instanzen zu iterieren.

Boost::Fusion bietet dafür eine Funktion namens `boost::fusion::for_each` an, die als Parameter eine iterierbare Boost::Fusion Struktur (wie z. B. eine boost::fusion::list) und einen Funktor erhält. Ein Funktor ist ein Struct, das den `()` Operator überschreibt und als Parameter die Elemente der iterierbaren Struktur besitzt. Ein einfa-

```
template <int I>
struct Loop
{
    inline static void call()
    {
        std::cout << I << "␣";
        Test<I-1>::call();
    }
};
template <>
struct Loop<0> //Rekursionskopf Template-Spezialisierung
{
    inline static void call()
    {
        std::cout << std::endl;
    }
};

//Aufruf mit
Loop<5>::call(); //5 4 3 2 1
```

Algorithmus 3: Beispiel einer rekursiven Schleifenauswertung zur Kompilierzeit mithilfe von TMP.

```
struct my_iterator
{
    template< typename TType >
    void operator()( TType &element )
    {
        std::cout << element << std::endl;
    }
};
//...
using boost::fusion::list< int, float > = MyList;
MyList myList( 23, 4.2f );
boost::fusion::for_each( MyList, my_iterator() );
```

Algorithmus 4: Einfaches Beispiel eines zur Kompilierzeit ausgeführten `for_each` der Boost::Fusion Bibliothek.

ches Beispiel ist in Algorithmus 4 zu sehen. Dieser Funktor gibt alle Elemente der Liste aus, unabhängig von deren Typ. Einzig der << Operator muss definiert sein.

3.3.3 Rechenbeschleuniger abstrahieren

Wie im Kapitel 2 angedeutet wurde, gibt es verschiedene Möglichkeiten Hardwarebeschleuniger zu programmieren. Leider unterscheiden sich diese Möglichkeiten zum Teil stark je nach eingesetzter Hardware. Für AMD GPUs wird primär die Spezialsprache OpenCL genutzt. Nvidia wiederum hat nur veraltete OpenCL Unterstützung und forciert die eigene C++ Spracherweiterung CUDA. Intel Xeon Phis werden mit einer erweiterten OpenMP Syntax programmiert. Für die Ansteuerung der Beschleunigerhardware wird deshalb die *Abstraction Library for Parallel Kernel Acceleration* (ALPAKA) [Wor16] genutzt, welche ein einheitliches Interface

für Hardwarebeschleuniger implementiert [ZWW+16, Wor15]. Das Besondere gegenüber klassischen Abstraktionsbibliotheken ist dabei, dass sich die kompilierte Anwendung durch den stark templatisierten Charakter der Bibliothek nicht von einer nativen Lösung unterscheidet. Die Abstraktionsschicht kann zur Kompilierzeit vollständig entfernt werden.

ALPAKA abstrahiert die Hardware dabei ähnlich wie es die Firma Nvidia in CUDA für ihre GPUs macht: Ein Programmstück, ein sogenannter *Kernel,* wird parallel für verschiedene Gitterpositionen ausgeführt. Mithilfe dieser Positionen kann ein Kernel dann algorithmusspezifisch bestimmen, auf welchem Datenstück er arbeiten soll. Das Gitter hat zwei verschiedene Dimensionen, mit welchen die Kernel adressiert werden. Das gesamte Gitter wird zuerst in Blöcke eingeteilt und diese werden wiederum in Threads unterteilt. Für die Anwendung macht es dabei keinen Unterschied, ob alle Kernel wirklich parallel oder u.U. sogar seriell ausgeführt werden. Es wird nur eine Gitter- und Kernelbeschreibung übergeben und ALPAKA stellt sicher, dass dieser Kernel für jeden Gitterpunkt einmal ausgeführt wird.

ALPAKA unterstützt verschiedene APIs, genannt *Accelerator*s, um auf verschiedene Hardwarebeschleuniger zuzugreifen, z. B. CUDA, OpenMP, verschiedene threadbasierte[1] Versionen und eine serielle Implementierung. Ähnlich zu CUDA bietet ALPAKA für die Einzelthreads innerhalb eines Blockes Synchronisationsmöglichkeiten an, aber keine Interblocksynchronität. Für Accelerators, die keine Synchronisation ermöglichen können, z. B. aufgrund serieller Ausführung, ist eine Blockgröße von 1 obligatorisch. In diesem Fall sind alle Threads des Blocks per Definition synchron.

Das Interface für Kernelausführung ähnelt dem Konzept zur Ausführung von Threads. Nach dem Aufruf des Kernels vom Host läuft dieser (parallel) im Hintergrund auf dem Device und der Fokus wird sofort zurückgegeben. Mittels Synchronisationsfunktionen kann später auf die Abarbeitung der Kernelinstanzen gewartet werden.

Device- und Hostprogramme sind genau wie bei CUDA komplett unabhängig voneinander. Sie können (für gewöhnlich) nicht direkt miteinander kommunizieren und auch nicht auf dieselben Daten zugreifen. Es gibt gesonderte Funktionen, um Host- oder Devicespeicher anzulegen, welcher dann nur von dem Host bzw. dem Device aus zugreifbar ist. Zur Verbindung beider Welten gibt es einerseits Kopierfunktionen, die zwischen Host und Device Daten kopieren können, und zum anderen die Möglichkeit beim Aufruf des Kernels Parameter mitzugeben. Diese werden dabei automatisch vom Host- in den Devicespeicher kopiert.

[1] Gemeint sind Threads im Sinne von Betriebssystemsthreads oder Boost::Fiber::Threads, nicht die parallel auf Rechenbeschleunigern ausgeführten Threads.

Neben den Kernels, die die Grenze zwischen Host und Device darstellen, gibt es auch auf dem Device Funktionen, die nur auf dem Device selbst ausführbar sind. Für CUDA gibt es das Schlüsselwort `__device__` und für ALPAKA analog das Präprozessor-Define `ALPAKA_FN_ACC`, welche eine Funktion als Devicefunktion definieren. Jede Funktion ohne solch ein Schlüsselwort wird automatisch als Hostfunktion angesehen und ist auf dem Device nicht nutzbar. Es gibt ein explizites `__host__` (CUDA) bzw. `ALPAKA_FN_HOST` (ALPAKA) Schlüsselwort, welches vor allem zusammen mit dem Device-Schlüsselwort genutzt wird, um eine Funktion auf Host und Device bereitzustellen bzw. verschiedene Versionen einer Funktion in Abhängigkeit des Ortes zu implementieren.

Dadurch ergibt sich jedoch das Problem, dass Funktionen aus klassischen Bibliotheken – auch Template Bibliotheken – nicht auf dem Device nutzbar sind, wenn sie nicht mit dem devicespezifischen Schlüsselwort definiert worden sind. Die erste Version des CUDA Toolkits kam im Juni 2007 heraus [Nvi16a]. In den letzten neun Jahren wurden deshalb viele Bibliotheken für GPUs portiert bzw. der Quelltext entsprechend angepasst. Neben der Standard C++ Bibliothek umfasst das auch die Template Bibliotheken Boost::MPL und Boost::Fusion, welche seit Version 1.56 (2014) für alle Methoden `BOOST_GPU_ENABLED` aktivieren, was für CUDA entsprechend mit `__host__ __device__` und für andere Compiler leer definiert ist.

3.4 JavaScript Object Notation

Für Livevisualisierungen ist die Übertragung von beliebigen Steuersignalen unabdingbar. Damit ISAAC erweiterbar und der Client austauschbar bleibt, wird deshalb die *JavaScript Object Notation* (JSON) für den Datenaustausch genutzt. JSON ist ein menschenlesbares Datenformat ähnlich XML. Im Unterschied zu diesem sind JSON Zeichenketten kürzer und einfacher verständlich. Tabelle 3.1 zeigt dieselben Daten mithilfe von XML und JSON kodiert.

JSON Daten haben außerdem den Vorteil, dass sie gültiges JavaScript bilden und somit einfach in Webanwendungen integriert und angezeigt werden können. Nichtsdestotrotz bietet sich diese Serialisierung aufgrund ihrer einfachen Struktur auch für andere Sprachen an und es gibt mittlerweile für jede größere Programmiersprache Implementierungen. In der RFC Norm 7159 [Bra16] sind für JSON folgende Datentypen definiert:

- *null* als fehlender Wert.
- *true* und *false* als boolsche Werte.

```
<Serie Name="Firefly"
       Staffeln="1"
       Abgeschlossen="True">
    <Produktion>
        <Person Nr="1">
            Joss Whedon
        </Person>
        <Person Nr="2">
            Tim Minear
        </Person>
    </Produktion>
    <Staffeln>
        <Staffel Folgen="14"/>
    </Staffeln>
</Serie>
```

```
{
    "Name" : "Firefly"
    "Produktion":
    {
        "Joss␣Whedon",
        "Tim␣Minear"
    },
    "Staffeln": 1
    "Folgen" : [14]
    "Abgeschlossen" : True
}
```

Tabelle 3.1: Vergleich von der XML (links) und JSON (rechts) Darstellung des semantisch selben Datensatzes. Die JSON Variante ist kompakter und intuitiver lesbar – ohne an Beschreibungsmächtigkeit zu verlieren.

- Beliebige Zahlen, einschließlich negativer Zahlen, Kommazahlen und einer Darstellung in der Zehnerexponentenform wie z. B. *4.2e1*.
- Zeichenketten, die mit *“* begonnen und geschlossen werden, z. B. *“Firefly”*.
- Arrays, in denen sortierte Elemente mit Kommata getrennt werden und die mit *[* beginnen und mit *]* enden. Arrays können beliebige – auch verschiedene – Typen beinhalten. Sie lassen sich also auch schachteln.
- Objekte, die einem Zeichenketten-Schlüssel mittels *:* einen beliebigen anderen Datentyp zuweisen. Sie beginnen mit *{* und enden mit *}*. Genau wie Arrays sind Objekte frei schachtelbar. Im Gegensatz zu einem Array besitzen Objekte jedoch keine feste innere Ordnung. Objekte sind außerdem der Wurzeltyp eines JSON Objektes. Ein Beispiel ist rechts in der tabellarischen Gegenüberstellung 3.1 zu sehen.

Diese grundlegenden Datentypen sind ausreichend, um beliebige Metadaten aus der Simulation an den Client zu senden sowie Steuerdaten vom Client an die Simulation bzw. die Visualisierung. Für diese Arbeit wird dabei die C-Bibliothek Jansson [Leh16] genutzt, aufgrund der schnellen und gut dokumentierten Implementierung. Das ist vor allem wichtig, damit auch die aufrufende, hoch parallele Anwendung einfach Feedbackdaten lesen und Metadaten zurückgeben kann.

4 Konzept

Das Berechnen einer Volumenvisualisierung aus Simulationsdaten bei Vermeidung von Host-Device Kopieroperationen und einer hohen Spezialisierung an konkrete Simulationen durch Templatisierung ist kein alleiniger Garant für eine niedrige Laufzeit. Die Simulation hat für gewöhnlich Priorität und die Visualisierung sollte deshalb nicht zu viel Rechenzeit des Rechenknotens bzw. des Rechenbeschleunigers beanspruchen und vor allem die Arbeit auf letzerem komplett einstellen, wenn die Simulation weiter läuft. Um Live-Visualisierungen steuern zu können, ist es essentiell, in der Lage zu sein, Nachrichten an die Visualisierungen schicken zu können. Da mehrere Clients eine Visualisierung steuern können, muss diese wiederum geänderte Steuerungseinstellungen an die Clients zurückschicken können. Auf demselben Kanal zwischen Simulation und den Clients soll es für die Simulationen auch möglich sein, visualisierungsunabhängige Simulationsmetadaten zu schicken bzw. von den Clients zu empfangen.

4.1 In Situ Animation of Accelerated Computations

Diese Arbeit stellt deshalb die Lösung *In Situ Animation of Accelerated Computations* (ISAAC) vor. Ziel ist eine generisch beschriebene, aber gleichzeitig hoch simulationsspezifische und damit schnelle Visualisierungslösung zu schaffen, die unabhängig von den genutzten Rechenbeschleunigern ist. ISAAC stellt dabei nur eine Annahme an: Die zu visualisierenden Daten befinden sich in einem dreidimensionalen Volumen in der Form eines Quaders, welcher auf die Rechenknoten des HPC-Systems aufgeteilt wird, siehe Abbildung 4.1. Jeder Knoten besitzt dabei genau einen disjunkten Teilquader des Volumens – später *lokales (Sub-)Volumen* genannt –, für welchen jede *Quelle* für jede Position in diesem einen Skalar- oder Vektorwert zurückgeben kann. Eine Quelle beschreibt eine Datenquelle für das Volumen, wie z. B. elektrische oder magnetische Felder oder auch Partikeldichten. Ausgehend von dem typischen Aufbau eines Rechenclusters in Abbildung 3.1 werden drei unabhängige Programmbestandteile beschrieben.

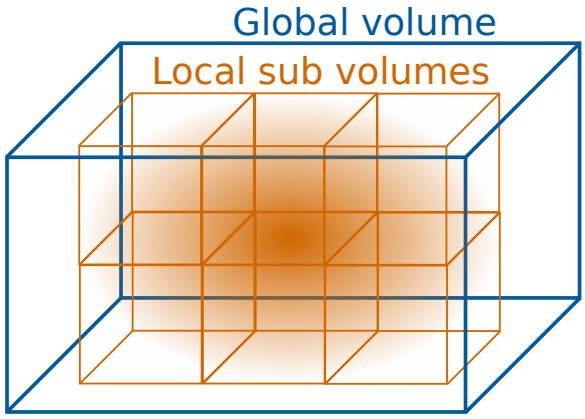

Abbildung 4.1: Von ISAAC genutzte Abstraktion der zu visualierenden Anwendung. Die Anwendung beschreibt ein globales Volumen, in dem Quellen Felder definieren. Diese Felder sind auf disjunkte Subvolumen unterteilt, wovon jeder Rechenknoten eines beschreibt.

4.1.1 ISAAC Template Bibliothek

Diese Bibliothek wird von einer Simulation eingebunden und soll die Volumenvisualisierung bereitstellen. Sie ist der Hauptteil dieser Arbeit. Sie läuft zusammen mit der Simulation auf den Rechenknoten des Clusters und profitiert beim Bildzusammenbau von dem Hochgeschwindigkeitsnetzwerk. Auf allen Rechenknoten läuft dieselbe Anwendung, nichtsdestotrotz bildet nur eine Instanz (meist mit dem MPI Rank 0) den Kontakt zur Außenwelt. Diese Instanz wird in dieser Arbeit *Master* genannt, alle anderen Ranks *Slaves*. Nur der Master erhält am Ende die komplette Volumenvisualisierung.

Das Interface zwischen Simulation und Volumenvisualisierung erfolgt mithilfe von Templates. ISAAC abstrahiert die Simulation als Liste von Quellen, die für das lokale Volumen eines Rechenknotens Skalar- und Vektorfelder beschreiben. Die Quellen werden dabei über Templates beschrieben, ISAAC erwartet nur bestimmte, konstante Eigenschaften wie die Vektordimension oder den Namen einer Quelle und eine Zugriffsmöglichkeit auf die Feldwerte. Auf diesen abstrakten Quellentypen wird in zeitkritischen Quelltextbestandteilen, wie z. B. dem Visualisierungskernel, mithilfe von TMP gearbeitet. Auf diese Art und Weise kann z.B. schon zur Kompilierzeit über die Quellen iteriert werden. Das Ziel ist damit nach außen ein sehr abstraktes und leicht zu implementierendes Interface bereitzustellen, aber intern trotzdem sehr simulationsspezifisch optimieren zu können – ohne dass ISAAC wissen muss, wie die aufrufende Anwendung aufgebaut ist. Durch die Nutzung von

Templates entsteht dabei kein Abstraktionsoverhead. Auch der Rechenbeschleuniger selbst wird mithilfe von ALPAKA vollständig abstrahiert.

Mithilfe von den Quellenbeschreibungen und TMP wird ein Raytracing Algorithmus implementiert, der das Emissions/Absorptions-Modell und eine Isoflächendarstellung unterstützt. Nur die Hauptschleife des Raytracers wird zur Laufzeit noch existent sein, da sie als einzige nicht schon zur Kompilierzeit bestimmt werden kann. Die Abtaststrecke wird für das lokale Volumen komplett vorberechnet, sodass keine Bereichszugriffsüberprüfung stattfinden muss. Beide Ansätze sparen einerseits Vergleichsoperationen, die für jeden Abtastwert erneut ausgeführt werden müssten, aber auch Register, wodurch weniger Variablen in den langsamen Hauptspeicher des Rechenbeschleunigers ausgelagert werden müssen. Das Beleuchtungsmodell für die Isoflächendarstellung wird mit Blick auf eine schnelle Ausführungszeit implementiert werden.

Simulationsdaten werden in den seltensten Fällen direkt visualisierbar sein. Einerseits ist die schon erwähnte Klassifikation von Skalarwerten zu Farb- und Opazitätswerten notwendig. Dazu nutzt ISAAC eine Transferfunktion, die als eindimensionales Array von Vierkomponentenvektoren für die drei Farb- und den Alphawert beschrieben ist. Der Wertebereich dieser Funktion geht jedoch nur von 0 bis 1. Werte kleiner als 0 werden wie 0 klassifiziert und analog Werte größer 1 wie ebendieser.

Es ist also von Nöten die Skalarwerte der Simulation in den Transferfunktionswertebereich zu verschieben und bei Simulationen, die Vektorfelder beinhalten, diese erst in Skalarfelder umzuwandeln. In Anlehnung an die Arbeit von More et al. [MAGM11] beschreibt diese Arbeit atomare *Functors*, die mithilfe des Pipezeichens | zu komplexen *Functor-Chains* kombiniert werden können. Wenn der Wertebereich eines Vektorfeldes zum Beispiel von $(-1, -2, -3)^{\mathrm{T}}$ bis $(4{,}5{,}6)^{\mathrm{T}}$ ginge, könnte die Functor-Chain `length | mul(0.114)` den Wertebereich auf 0 bis $1{,}0003$ verschieben. Natürlich ist das mit einem Informationsverlust des Vektorfeldes verbunden. Deshalb sollen diese Functor-Chains erst zur Laufzeit definiert werden und auch die Möglichkeit gegeben sein, einzelne Komponenten der Vektoren eines Vektorfeldes getrennt zu untersuchen, anstatt die Länge zu berechnen. Um keine Daten kopieren zu müssen, werden die Functor-Chains direkt bei der Iterierung durch das Volumen beim Raytracing on the fly angewendet.

Weiterhin muss die Abbildung der Simulationsdaten im Speicher nicht zwingend den physikalisch korrekten Verhältnissen entsprechen. Eine notwendige dimensionsabhängige Reskalierung soll an ISAAC übergeben werden können. Zu guter Letzt sollte ISAAC auch die Möglichkeit anbieten, beliebig viele Clippingebenen zu definieren. Dazu wird je ein Stützpunkt mit einem Normalenvektor übergeben. Nur die Seite der Ebene, in welcher dieser Vektor zeigt, wird angezeigt. Dazu

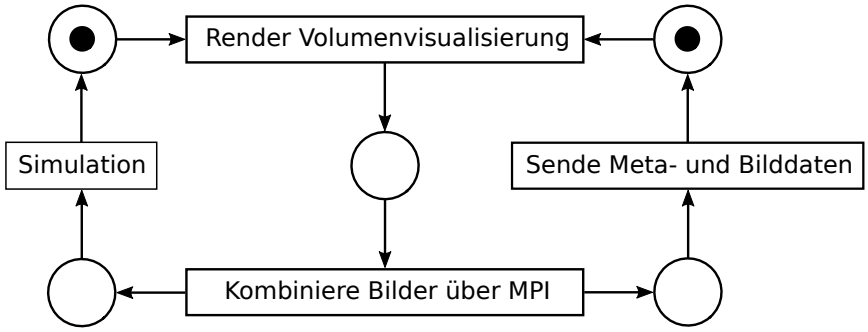

Abbildung 4.2: Petrinetzbeschreibung der Nebenläufigkeit von ISAAC. Nur das Senden von Meta- und Bilddaten kann nach diesem Modell parallel zu der Simulation laufen.

wird die vorberechnete Abtaststrecke des Sichtstrahles bei Bedarf gekürzt. Für die Schleife selbst ergibt sich so in keinem Fall eine schlechtere Laufzeit und, nur wenn Clippingebenen aktiviert sind, gibt es einmaligen, minimalen Mehraufwand in der Sichtstreckenberechnung.

Verteilte In-Situ Volumenvisualisierung Es hat viele Vorteile, dass Simulation und Visualisierung auf derselben Beschleunigerhardware laufen können. Andererseits würde die Visualisierung so aber bei einer parallelen Ausführung zur Simulation mit dieser um die Hardwareressourcen streiten. Der parallele Zugriff auf verschiedene Speicherbereiche könnte z. B. das automatische Caching behindern und somit für eine langsamere Ausführung sorgen als bei einer seriellen Ausführung. Zudem wird davon ausgegangen, dass sowohl die Simulation als auch ISAAC den Hardwarebeschleuniger alleine voll auslasten.

Des Weiteren nutzen sowohl die Simulation als auch ISAAC MPI, um mit den Nachbarknoten zu kommunizieren. Prinzipiell erlaubt der MPI-Standard eine Parallelausführung von MPI Befehlen in verschiedenen Threads [MPI16], jedoch ist die Implementierung in hochoptimierten, hardwarespezifischen Versionen nicht zugesichert und zumeist auch mit Performanceeinbußen verbunden.

Es hat sich bei Beobachtung von der exemplarischen Simulation PIConGPU jedoch gezeigt, dass die Host-Hardware nicht voll ausgelastet wird. Auch Rechenknoten, die primär auf Hardwarebeschleuniger setzen, besitzen relativ leistungsstarke CPUs. Im Falle des Rechenclusters Hypnos am HZDR besitzt ein Rechenknoten neben 4 Nvidia K20 Grafikkarten eine Intel Quad-Core Xeon CPU mit 2,4 GHz und durch Hyperthreading 8 Hardwarekernen, wovon 4 bei PIConGPU nicht genutzt werden.

Deshalb erhält ISAAC den vollen Programmfokus und unterbricht die Simulation so kurz wie möglich, um die Volumenvisualisierung auf der Beschleunigerhardware und die Bildkombination mit IceT vorzunehmen. Das Komprimieren und Senden der Daten, was weder den Rechenbeschleuniger noch MPI benötigt, läuft aber parallel zur weiterlaufenden Simulation. In Abbildung 4.2 ist der grobe Programmablauf als Petrinetz dargestellt. Insbesondere wird der Sonderfall betrachtet, dass der Zeitschritt der Simulation schneller fertig ist als das Komprimieren und Senden von Meta- und Bilddaten. In diesem Fall wird erst auf die Fertigstellung dieses Threads gewartet.

4.1.2 ISAAC Server

Wie in den Grundlagen in Abbildung 3.1 gezeigt wurde, ist nur der Loginknoten sowohl von außen als auch von innen erreichbar. Deshalb läuft auf diesem der ISAAC Server. Dieses C++ Programm verwaltet laufende Visualisierungen und angemeldete Clients. Es kommuniziert mit dem Master einer Visualisierung über TCP/IP. Alle Nachrichten, die zwischen Master und Server ausgetauscht werden, sind im JSON Format und folgen einem bestimmten Protokoll. Auf diese Art und Weise können alle Bestandteile unabhängig voneinander ausgetauscht werden. Auch eine Nicht-ISAAC-Visualisierung könnte sich dann anmelden und ihre Bilder und Metadaten übertragen.

Der Server selbst wiederum besteht aus drei Bestandteilen, die sich um

- die Kommunikation mit den Simulationen,
- die Metadatenkommunikation mit den Clients und
- die Streamgenerierung und Übertragung

kümmern.

Für eine bessere Auslastung dieser drei Aufgaben sollen diese in unabhängigen Threads laufen und über Nachrichten kommunizieren. Der Hauptthread namens *Broker* empfängt dabei nur Nachrichten von den drei Bestandteilen, filtert sie u. U. und leitet sie dann entsprechend an einen anderen Thread weiter. Pro Aufgabe ist es möglich, mehrere Implementierungen zu definieren, damit sich ISAAC nicht bzgl. der Kommunikation mit Clients auf bestimmte Formate oder Streamingarten einschränkt. Das ermöglicht z. B. auch eine bandbreitenabhängige Auswahl der Streamingimplementierung. Ähnlich der Abstraktion der Simulation für die In-Situ Bibliothek definiert auch der Broker nur ein Interface, dass solch Implementierungen erfüllen müssen.

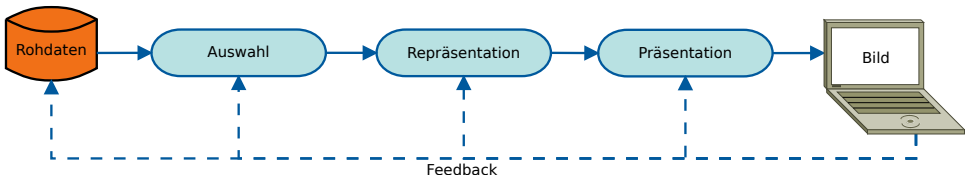

Abbildung 4.3: Klassischer Ablauf einer Visualisierung von den Rohdaten über typische Zwischenschritte hin zu einem angezeigten Bild. Jeder Zwischenschritt ist in dem Modell einzeln steuerbar.

4.1.3 ISAAC Client

So vielfältig wie die Implementierungen der Streamgenerierungen sein können, können auch Clients sein. Es gibt keinerlei Einschränkung bezüglich des Verbindungstyps, des verwendeten Datenformats, noch dem Typ der Anwendung. Im Rahmen dieser Arbeit wurde ein Referenzclient in HTML implementiert. Zum einen vereinfacht es die Arbeit mit dem JSON Format, welches direkt importiert und genutzt werden kann und zum anderen ist es jederzeit simulationsspezifisch möglich den Client anzupassen, z. B. um bestimmte übertragene Metadaten gesondert anzuzeigen. Aber auch Implementierungen als klassische Desktop- oder Terminalanwendung oder sogar als Smartphone-App wären denkbar und bedürfen keiner Anpassungen am Broker des Servers.

Die Clients bestimmen zwar, welche Art von Stream gesendet werden soll (basierend auf den möglichen Implementierungen der Streamgenerierung) und erhalten auch die Informationen, die notwendig sind, um den Stream zu erhalten (z. B. eine Stream-URL), nichtsdestotrotz ist es nicht deren zwingende Aufgabe, den Videostream auch anzuzeigen. Eine Terminalanwendung könnte zum Beispiel nur die Stream URL anzeigen, welche dann in einem Mediaplayer geöffnet werden könnte.

4.2 Einordnung im Visualisierungsprozess

Visualisierungsprozesse lassen sich, wie in Abbildung 4.3 gezeigt, in verschiedene Schritte einer Pipeline unterteilen. Es wird von einer wie auch immer gearteten Rohdatengrundlage ausgegangen, von der ein Teil für die weitere Datenverarbeitung ausgewählt wird. Im nachfolgenden Repräsentationsschritt wird die Darstellung dieser Auswahl erstellt, um im Präsentationsschritt in die Gesamtpräsentation integriert zu werden. Zu guter Letzt muss die Visualisierung angezeigt werden – in dieser Darstellung als Bild auf einem Computer, aber auch andere Anzeigetechniken wie z. B. Tabletts oder Smartphones sind denkbar. Von diesem Endgerät gibt es

eine Feedbackschleife zurück zu allen Einzelschritten, um die Visualisierung bzw. die Daten zu optimieren.

Die Rohdaten stellen in ISAAC die Quellen dar, die zur Kompilierzeit von der zu Grunde liegenden Anwendung definiert werden. Deren Ausgabe ist vollkommen simulationsabhängig und da beliebige Metadaten auch zu der Simulation übertragen werden können, können auch die Rohdaten entsprechend gesteuert werden. Für die primär beleuchtete Verwendung zur Visualisierung von Simulationen ergeben sich z. B. Möglichkeiten der Unterbrechung oder des Abbruchs dieser. Aber auch für andere Datenquellen, wie z. B. hochparallele Echtzeitberechnung von Volumendaten aus Experimenten, die sich nicht pausieren lassen, bieten sich mannigfaltige Steuermöglichkeiten an. Zum einen können experimentspezifische Steuersignale gesendet werden, die die Parameter im laufenden Experiment anpassen, aber natürlich kann das Experiment bei einem in der Visualisierung ersichtlichen Fehllauf auch gestoppt werden. Die Rohdaten solcher Experimente können des Weiteren sehr umfangreich sein, obwohl unter Umständen nur ein kurzer Zeitraum für eine komplexere Analyse interessant ist. In diesem Fall kann ISAAC genutzt werden, um auf Basis der Visualisierung im richtigen Moment die Speicherung bzw. anderwertige Weiterverarbeitung der Rohdaten in der Simulation anzustoßen.

Die Auswahl der zu visualisierenden Quellen erfolgt über ein Gewicht, das ISAAC jeder Quelle zuordnet. Für ein Gewicht von Null wird die Quelle komplett ausgefiltert. Räumlich können mit den Clippingebenen interessierende Bereiche ausgewählt werden. Aber auch die Functor-Chains bzw. in gewissen Maße die Transferfunktionen können für die Auswahl der Daten genutzt werden, insbesondere für die Auswahl der Vektorkomponenten und deren Wertebereiche. Durch eine Transferfunktion, die im maximalen Wert eine Opazität von Null hat, werden alle Werte herausgefiltert, die von der Functor-Chain über Eins gesetzt wurden. Ähnlich verhält es sich mit Werten kleiner Null. Auf diese Art und Weise kann nur das betrachtet werden, was in diesem Moment wirklich interessiert.

Die Repräsentation der Daten kann in ISAAC einerseits über die Wahl des Rendertypes, also Emissions/Absorptions-Modell oder Isoflächendarstellung, erfolgen, aber auch die Einstellung der Schrittweite, das detaillierte Verhalten von Functor-Chains, die gewählten Farben und Opazitäten in der Klassifikation mit der Transferfunktion, Interpolation und nicht zuletzt der Blickwinkel auf das Volumen tragen dazu bei.

Die Präsentation der Daten wird nicht mehr in der In-Situ Bibliothek, sondern im ISAAC Server entschieden. Ausgehend von den Starteinstellungen des Servers, z. B. welche Plugins geladen wurden, und den Streamingeinstellungen, die die Clients bei dem Beginn der Beobachtung mitübertragen, werden verschiedene Präsen-

tationsformen ausgeführt. Durch die offene, erweiterbare Struktur der Serverplug-ins sind den Präsentationsmöglichkeiten, aber auch den anzeigenden Endgeräten hier keine Grenzen gesetzt.

5 Umsetzung

In Abbildung 5.1 ist das gesamte Konzept in einer Grafik veranschaulicht. Die blau markierten Pfeile zeigen den Datenfluss über das Netzwerk oder innerhalb eines Programms. Zusätzlich ist die Art der Übertragung angegeben. Es fällt auf, dass fast ausschließlich JSON Daten verschickt werden. Insbesondere werden auch zwischen Simulation und Server nur JSON Daten geschickt. Die Bilder werden deshalb vorher in ein gültiges JSON Format gebracht.

Alle JSON Nachrichten in ISAAC besitzen dabei mindestens einen Schlüssel “type”, der den Typen der Nachricht angibt und z. B. vom Broker für die schnelle Weiterleitung genutzt wird. Die einzelnen Bestandteile dieser Grafik sollen folgend beschrieben werden.

5.1 In-Situ Bibliothek

Die Templatebibliothek bietet einer hochparallelen Anwendung mithilfe Templates und TMP die Möglichkeit eine C++ Klasse zu definieren, die sehr spezifisch auf die verschiedenen darzustellenden Quellen zugeschnitten ist. Diese Klasse kann dann eine Verbindung zu einem ISAAC Server herstellen, Visualisierungen erstellen und sowohl diese als auch beliebige Metadaten übertragen.

5.1.1 Quellenbeschreibung

Dadurch, dass ISAAC eine Template Bibliothek bereitstellt, erfolgt die Einbindung alleine durch die Headerdatei `isaac.hpp`. Nichtsdestotrotz müssen die Abhängigkeiten Boost::Fusion, IceT, Jansson, und libjpeg in den Buildprozess der Simulation eingebunden werden. Alle vier besitzen jedoch gut dokumentierte Anbindungen an klassische Buildsysteme wie Make oder CMake. Es hat sich bei der Implementierung von ISAAC für PIConGPU gezeigt, dass es Probleme bereiten kann, die CUDA Implementierung von ALPAKA in bestehenden CUDA Code einer Simulation einzuarbeiten. Da ALPAKA jedoch sehr nahe an CUDA angelehnt ist, wurde ISAAC sowohl kompatibel zu ALPAKA als auch zu reinem CUDA ohne ALPAKA-Abhängigkeit implementiert. An den wenigen unterschiedlichen Stel-

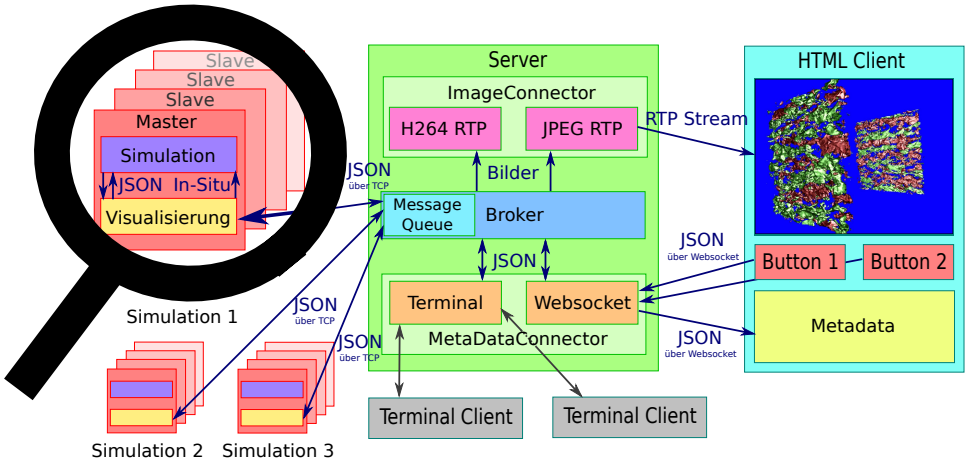

Abbildung 5.1: Konzept von ISAAC: Man erkennt die drei Hauptkomponenten Visualisierung, Server und Client. Die Visualisierung läuft verteilt mit der Simulation auf Rechenknoten. Nur ein Rechenknoten (Master) kommuniziert mit dem Server. Der Server hat verschiedene, nebenläufige Threads für die Simulationskommunikation (Broker), Streamerzeugung (ImageConnectors) und Clientkommunikation (MetaDataConnectors).

len wurden `#ifdef`s genutzt, um mit demselben Quelltext beide Möglichkeiten zu nutzen. Zugleich bietet sich so die Möglichkeit die Performance von der nativen CUDA und der abstrakten ALPAKA Implementierung zu vergleichen. Somit hat ISAAC zusätzlich entweder eine Abhängigkeit von ALPAKA oder von CUDA – je nachdem, welche Hardwarebeschleuniger genutzt werden sollen.

Die Basis der Volumenvisualisierung stellen Quellen dar. Eine Simulation kann beliebig viele Quellen definieren. Im Volume Raycasting wird für jeden Abtastschritt für jede Quelle ein Datum für die Position im Volumen gelesen, die Functor-Chain ausgeführt und das Resultat mit der Transferfunktion klassifiziert. Dann wird die Gesamtklassifikation mithilfe von quellenspezifischen Gewichten ausgerechnet und in dem im Kapitel 3.2 beschriebenen Algorithmus verwendet.

Die Quellen werden als Boost::Fusion Liste definiert. Diese Liste beinhaltet für jede Quelle eine simulationspezifische Klasse, die bestimmte Eigenschaften erfüllen muss. Zur Beschreibung muss sie vier konstante, statische Werte definieren, die dann zur Kompilierzeit direkt substituiert werden können:

- `static const std::string name` beschreibt den Namen der Quelle. Dieser wird auch an die beobachtenden Clients geschickt.

- `static const size_t feature_dim` beschreibt, ob es sich bei der Quelle um ein Skalar- oder Vektorfeld handelt, und wie groß letzteres ist. ISAAC unterstützt bis zu vierdimensionale Vektorfelder. Größere Vektorfelder können jedoch in mehrere Quellen aufgeteilt werden. Wenn ein Vektorfeld z. B. Geschwindigkeit und Krafteinwirkung beschreibt, ergibt eine Aufspaltung auch semantisch Sinn.
- `static const bool has_guard` beschreibt, ob über die Grenzen des Volumens hinaus auf Daten zugegriffen werden darf. Dabei ist es für ISAAC unerheblich, ob die Daten sinnvoll sind oder nicht, wobei der erstere Fall bessere Bilder erzeugt. Jedoch entfallen so Randüberprüfungen – insbesondere, wenn Interpolation aktiviert wird.
- `static const bool persistent` beschreibt, ob zu jedem Zeitpunkt auf die Quelle zugegriffen werden kann, oder ob das Volumen nur für eine kurze Zeit verfügbar ist. Bei PIConGPU gibt es zum Beispiel zwei Arten von Feldern. Persistent sind elektrische, magnetische und Stromstärkefelder. Diese werden auch für die Simulation genutzt und können immer angesprochen werden. Jedoch gibt es sekundär berechnete Felder, z. B. von der Elektronen- oder Ionendichte, die für Visualisierungen in ein und demselben Buffer geschrieben werden. Es wäre in diesem Beispiel nicht möglich auf Elektronen- und Ionendichten gleichzeitig zuzugreifen. Deshalb bietet ISAAC für diese Quellen die Möglichkeit an Speicher auf dem Device zu allozieren und die Volumendaten – wenn notwendig – zu sichern.

Unter anderem für den letzten Punkt muss jede Quelle deshalb eine `update` Methode `void update(bool enabled, void* pointer)` implementieren, die für jede Quelle einmal pro Frame auf dem Host aufgerufen wird. `enabled` zeigt an, ob die Quelle im Moment aktiviert ist. Es steht der Simulation frei zu entscheiden, ob in diesem Fall Vorbereitungsschritte für dem Zugriff entfallen können. Nicht persistente Quellen werden in diesem Fall nicht aktualisiert. `pointer` ist ein void pointer, der von der Simulation bei der später beschriebenen Anstoßung des Renderings mitgegeben wird. Er kann beliebig verwendet werden und wird nur durchgereicht. Für PIConGPU wird er zum Beispiel genutzt, um den momentanen Zeitschritt an die `update` Methode weiterzureichen, welcher für interne Funktionen benötigt wird. Es ist auch Aufgabe dieser Methode den Zugriff auf die Quelle bis zum `update` Aufruf der nächsten Quelle zu ermöglichen, wenn sie nicht persistent ist, damit eine ISAAC-interne Kopie angefertigt werden kann.

Weiterhin muss jede Quellenklasse den Subscriptoperator []

```
inline float_dim< feature_dim > operator[] (const int3& nIndex) const
```

```
1  Source1 source1 ( /* klassenspezifische Parameter */ );
2  Source2 source2 ( /* klassenspezifische Parameter */ );
3  using SourceList = boost::fusion::list
4  <
5      TestSource1,
6      TestSource2
7  >;
8  SourceList sources( source1, source2 );
9
10 auto visualization = new IsaacVisualization
11 <
12     SourceList, //In Zeile 3 definierte Boost::Fusion Klassenliste
13     ...         //Weitere Template Argumente
14 > (
15     sources, //In Zeile 8 instanziierte Boost::Fusion Liste von Quellen
16     ...      //Weitere Parameter für die Instanziierung
17 );
18
19 if ( visualization->init() )
20     std::cerr << "Isaac init failed" << std::endl;
```

Algorithmus 5: Vereinfachte Darstellung der Erstellung einer ISAAC Visualisierungsinstanz für CUDA. Durch die Übergabe der Quellenklassen als Template Argument entsteht eine stark simulationsspezifische Spezialisierung der generisch definierten Visualisierungsklasse.

überschreiben, damit der Volumenrenderer auf das Volumen der Quelle zugreifen kann. Der Operator wird nur auf dem Device ausgeführt, erhält als Parameter die Position im lokalen Subvolumen und gibt einen Vektor der Dimension `feature_dim` zurück. Da der Operator als `inline` definiert ist, erfolgt bei Operatornutzung kein Funktionsaufruf, sondern der Körper des Operators wird direkt an die entsprechende Stelle in der Raycastingschleife kopiert und der Kernel für die Quelle optimiert.

5.1.2 ISAAC initialisieren

Nachdem für jede Quelle eine Klasse definiert wurde, müssen diese in eine Boost::Fusion::List eingefügt, instantiiert und bei der Initialisierung von ISAAC übergeben werden. Algorithmus 5 zeigt exemplarisch die Erstellung für zwei fiktive Quellen für CUDA. Im Falle von ALPAKA kämen noch Typen und Parameter für Host, Accelerator und Streams hinzu. Diese Details sind an dieser Stelle aber nicht relevant, weshalb sich auch in zukünftigen Beispielen auf die vereinfachte Darstellung beschränkt wird. Auch wurde in der Darstellung auf weitere wichtige Parameter für die Visualisierung verzichtet, die an dieser Stelle nicht essentiell sind.

Wenn `visualization->init()` mit 0 terminiert, wurde eine Verbindung zum ISAAC Server hergestellt und `visualization` kann nun für die Visualisierung der übergebenen Quellen genutzt werden. An den Server wird eine JSON Nachricht vom Typen “register” geschickt und die Simulation so bei diesem registriert. Jeder verbundene

Client wird von der Verfügbarkeit der Visualisierung informiert werden. Auch neue Clients, die sich anmelden, nachdem eine Visualisierung schon angelaufen ist, erhalten diese Information bei der Anmeldung. Neben Namen der Simulation und der Auflösung des Bildes werden auch die verfügbaren Functors, die maximale Länge der Functor-Chain und die Namen alle Quellen übertragen. Des Weiteren wird für den Master ein Extrathread gestartet, der die ganze Zeit auf dem Host läuft, auch wenn die Simulation weiter rechnet. Dieser Thread wartet auf Nachrichten vom Server und speichert sie in einer threadsicheren verketteten Liste, damit ISAAC die Befehle abarbeiten kann, wenn die Simulation das nächste Mal den Fokus abgibt.

5.1.3 Visualisierung

Mit `visualization->doVisualization(META_MASTER,&curStep,true)` wird die Visualisierung ausgelöst. Wie schon angedeutet, blockiert diese Funktion den Programmablauf solange bis keine Aufgaben mehr vorhanden sind, die MPI oder den Rechenbeschleuniger brauchen. Es ist der Simulation überlassen, wann sie die Visualisierung anstoßen will. PIConGPU besitzt direkt ein Pluginsystem, welches eine `notify` Methode bereitstellt, die in einer einstellbaren Periode von Zeitschritten aufgerufen wird. ISAAC selbst stellt keine Vermutungen über die Frequenz seiner Aufrufe an. Jedoch hängen die Frames per Second (FPS) der Visualisierung direkt mit der Aufruffrequenz der Methode zusammen.

`doVisualization` besitzt drei Parameter. Der erste steuert, ob nur der Master oder alle Knoten Metadaten an die Clients übertragen werden. Der zweite ist der schon angesprochene Pointer mit simulationsspezifischen Daten für die `update` Methoden der Quellen. Der letzte Parameter gibt an, ob sich die Volumen geändert haben oder nicht. Im letzteren Fall wird ISAAC kein Bild übertragen, insofern auch keine Änderungen der Kamera, Functor-Chains oder Transferfunktionen erfolgten. Auch die `update` Methoden der Quellen werden in diesem Fall nicht aufgerufen. Abgesehen von der Kommunikation des Masters mit dem Server erfolgt die gesamte Kommunikation der Knoten mithilfe von MPI, um auch auf Peta- und in der Zukunft auf Exascalesystemen noch performant zu bleiben.

Die Visualisierungsmethode lässt sich in Abhängigkeit von dem ersten Parameter in sechs oder sieben Schritte einteilen:

1. Der Master arbeitet die eingegangenen Nachrichten vom Server bzw. den mit diesem verbundenen Clients ab. Sollten in dem Zeitraum seit dem letzten Bild widersprüchliche Befehle eingegangen sein, gilt nur der aktuelle. Des Weiteren werden Nachrichten bzgl. der Kameraparameter aussortiert und in die für die Computergrafik typischen *Modelview-* und *Projektionsmatrizen*

konvertiert. Wie schon erwähnt, läuft für eingehende Nachrichten am Master die ganze Zeit ein Extrathread im Hintergrund, der auf der Verbindung zum Server auf neue Nachrichten lauscht und diese in eine verkettete Liste pusht, wohingegen der Hauptthread die Liste so lange von vorne abarbeitet bis keine Nachricht mehr vorhanden ist. Da dieser Extrathread die meiste Zeit auf Nachrichten wartet, schläft er in dieser Zeit und braucht so gut wie keine Prozessorzeit. Gleichzeitig wird aber sichergestellt, dass der Netzwerkstack nicht überfüllt wird, wenn der ISAAC Server viele Nachrichten zwischen zwei Frames schickt.

2. Die gefilterten und u. U. um Modelview- oder Projektionsmatrix ergänzten JSON Objekte werden dann per MPI Broadcast an alle beteiligten Knoten geschickt und parallel ausgewertet.

3. Jeder Knoten berechnet basierend auf den Modelview- und Projektionsmatrizen seinen Abstand zur Kamera. Dann tauschen alle Knoten diese Informationen mittels MPI Allgather untereinander aus, damit alle dieselbe Knotenreihenfolge an IceT übergeben können.

4. Nun erfolgt das eigentliche Rendern des Bildes auf der Beschleunigerhardware.

5. Anschließend wird es mittels IceT zu einem Gesamtbild auf dem Master zusammengefügt.

6. Wenn die Funktion mit `META_MERGE` aufgerufen wurde, sammelt der Master in einem optionalen Schritt mittels MPI Gather von allen Slaves Metadaten und erstellt daraus ein einzelnes JSON Objekt.

7. Der letzte Schritt erfolgt in einem Extrathread, während die Simulation schon wieder anläuft (siehe Petrinetz in Abbildung 4.2). Die Metadaten werden zusammen mit ISAAC-spezifischen Informationen an den Server zur Weitergabe an die Clients geschickt. Außerdem wird das Bild komprimiert und auch als eigenes JSON Objekt an den Server geschickt. JSON erlaubt nur UTF8 Zeichen und kann nicht direkt Binärdaten beinhalten, weshalb das Bild zunächst in eine Base64-Zeichenkette umgewandelt werden muss. Dabei werden je 6 Bit des Binärdatenstreams zu einem ASCII-Zeichen umgewandelt. Aus je 3 Byte (24 Bit) entstehen so 4 Zeichen (32 Bit). Dadurch werden zwar $\approx 33\,\%$ mehr Daten übertragen, es muss aber kein zweiter Socket für Binärdaten zum Server geöffnet und kein proprietäres Protokoll etabliert werden.

Die Methode gibt einen Pointer auf ein `json_t` Objekt zurück, welches einem JSON Objekt in Jansson entspricht. Es enthält alle Daten, die seit dem letzten Visualisierungsschritt von Clients an den Server und von diesem weiter an den Master ge-

schickt wurden. Intern werden sie in dem Unterobjekt “metadata” gespeichert. Der Zugriff erfolgt mit der Jansson-API, z. B. ließe sich mit

```
json_t* meta = visualization->doVisualization(META_MASTER,&curStep,true);
json_t* js = json_object_get(meta, "exit")
if ( js && json_is_boolean( js ) && json_boolean_value( js ) )
    /* Simulation beenden */
json_decref( meta );
```

überprüfen, ob ein Client die boolsche Variable “exit” mit dem Wert *true* geschickt hat und dann die Simulation beenden. Wichtig ist, dass die Simulation den zurückgegebenen Pointer am Ende mit `json_decref` freigibt, unabhängig davon, ob die Simulation die Metadaten auswertet oder nicht.

Ähnlich lassen sich auch Metadaten von der Simulation an die Clients durchreichen. Mithilfe der Methode `visualization->getJsonMetaRoot()` erhält man einen Pointer auf `json_t` zurück, der auf das Objekt “metadata” im pro Frame geschickten JSON Objekt verweist. Auf diese Weise ließe sich z. B. jeden Frame der Zeitschritt der Simulation mitübertragen:

```
if (rank == 0)
{
    json_t* meta = visualization->getJsonMetaRoot();
    json_object_set_new( meta, "current␣step", json_integer( curStep ) );
}
visualization->doVisualization(META_MASTER,&curStep,true);
```

Alle Metadaten, die vor der Initialisierung von ISAAC mit `visualization->init()` übergeben werden, werden als Beschreibung der späteren, periodischen Metadaten interpretiert und vom ISAAC Server zusammen mit der restlichen Beschreibung der Simulation an alle Clients geschickt. Es besteht jedoch kein Zwang später genau diese Metadaten auch zu übertragen. Zum Beispiel wäre es denkbar, vor der Initialisierung `{"curStep": "Der␣momentane␣Zeitschritt␣in␣ms"}` zu übertragen, um dann periodisch in den Metadaten dem Schlüssel `"curStep"` den Zeitschritt als Fließkommazahl zuzuordnen.

Kostenlos zu der Möglichkeit die Visualisierung zu steuern, erhalten Simulationen also einen Kanal für beliebigen Meta- und Steuerdatenaustausch mit der Möglichkeit auch die Semantik der Metadaten einmalig zu beschreiben.

5.1.3.1 Functor-Chain Vorkompilierung

Eine der großen Herausforderungen an ISAAC ist es, nicht nur viele Steuer- und Analysiermöglichkeiten einzubauen, sondern gleichzeitig performant zu bleiben, um die laufende Simulation so wenig wie möglich zu beeinträchtigen. Die schon erwähnten Functor-Chains bestehen aus einer Aneinanderreihung von Functors, die

wiederum einen Namen und Parameter haben. Bisher implementiert sind Functors für Addition, Multiplikation, Länge, Potenzierung, die Summe aller Komponenten sowie ein Standardfunctor, der die Daten nicht verändert. Jeder Functor kann bis zu 4 Parameter haben, unabhängig davon, ob er alle oder überhaupt einen benutzt. Der Additionsfunctor `add(0.5,1)` würde z. B. die X-Komponente mit $0{,}5$ addieren und die Y-Komponente mit $1{,}0$. Nicht angegebene, aber von einem Functor genutzte Parameter werden auf den zuletzt angegebenen Parameter gesetzt oder auf Null, wenn keiner angegeben wurde. Die Z-Komponente würde in dem Additionsbeispiel also auch um 1 addiert werden.

Einige Functors geben niedrigdimensionalere Daten zurück als übergeben wurden. Der Rückgabewert von Länge und Summe hat z. B. immer die Dimension 1. Von dem Volumenrenderer wird prinzipiell nur die X-Komponente beachtet, unabhängig davon, ob ein Skalar- oder Vektorfeld abgetastet wird und welche Functors vorher eingesetzt wurden. Durch geschickten Einsatz der Functors lassen sich aber auch die anderen Vektorkomponenten einzeln betrachten.

Mithilfe dieser atomaren Functors lassen sich auch komplexere Funktionen wie z. B. das Filtern des Absolutwerts der Y-Komponente implementieren: `mul(0,1,0) | sum | length`. Diese Functor-Chain würde zuerst die X- und Z-Komponente nullen, dann aufsummieren, wodurch nur noch eine eindimensionale Y-Komponente übrig bleibt und zuletzt mit `length` den Betrag berechnen. Die Functor-Chain `mul(0,1,0) | length` hätte denselben Effekt, jedoch weiß der Längenfunctor nicht, dass die X- und Z-Komponente Null sind und berechnet deshalb in jedem Fall eine Wurzel. Im ersten Fall kann die Länge durch `fabs(x)` optimiert werden, da `sum` nur eindimensionale Daten ausgibt. Die Functors werden für jeden der vier möglichen Fälle an Vektordimensionen mehrfach implementiert, um solch spezifische Optimierungen durchführen zu können.

Wenn ein Client eine Functor-Chain für eine Quelle an die Simulation übergibt, wird diese in zwei Strukturen zerlegt: Einerseits eine Art Bytecode, einer Liste von Nummern, wobei jede Nummer eindeutig einem Functor zugewiesen werden kann. Für das erste Beispiel wäre das $[0{,}4{,}2]$. Außerdem wird ein Array mit den Parametern erstellt. Für dasselbe Beispiel wäre das $[[0{,}1{,}0{,}0]\,,[0{,}0{,}0{,}0]\,,[0{,}0{,}0{,}0]]$. Obwohl `sum` und `length` keine Parameter haben und `mul` keinen für die W-Komponente benutzt, werden sie trotzdem gesetzt. Es könnten nun beide Arrays auf den Hardwarebeschleuniger kopiert werden und für jede Quelle in jedem Abtastschritt der richtige Functor ausgewählt und ausgeführt werden. Das wäre jedoch nicht performant.

Eine erste Optimierung stellen Funktionspointer dar. Anstatt die Nummern der Functors hochzuladen, werden direkt die Funktionspointer auf dem Device gespei-

chert. Jedoch müssen auch hier für jede Quelle mehrere Funktionsaufrufe pro Abtastschritt vollzogen werden. Gerade bei sehr einfachen Functors wie `add` würde der Funktionsaufruf mehr Zeit beanspruchen als die eigentliche arithmetische Operation.

Deshalb muss zur Kompilierzeit feststehen, welche Functors benötigt werden, und wie viele maximal miteinander konkateniert werden sollen. Es wird eine Template Funktion definiert, die als Templateparameter eine Boost::MPL Liste mit den zu konkatenierenden Functor-Klassen erhält und zur Kompilierzeit über diese iteriert. Die Funktion gibt auch schon nur die X-Komponente des Eingabewertes zurück. Zum Programmstart wird ein Array mit Funktionspointern zu allen möglichen Kombinationen der Functors zu Functor-Chains erstellt und beim Update der Functor-Chain wird pro Quelle nur der entsprechende Funktionspointer ausgewählt.

Damit ergeben sich jedoch

$$4 \cdot c^n \tag{5.1.1}$$

Funktionen, wobei c die Anzahl der möglichen Functors ist, welche im Moment einschließlich der Idempotenz maximal 6 ist, und die maximale Länge der Functor-Chains n. Der Faktor 4 entsteht dadurch, dass von jeder Functor-Chain eine Variante für Skalarfelder und für 2-, 3- und 4-dimensionale Vektorfelder existieren muss. Verzichtet man auf den selten gebrauchten Potenzfunctor und setzt somit $c = 5$ und exemplarisch $n = 3$ ergeben sich 500 verschiedene Möglichkeiten, die aber nur zur Kompilierzeit erzeugt werden müssen. Man erkauft sich eine bessere Laufzeit durch eine längere Kompilierzeit.

Ein weiterer Vorteil dieses Ansatzes ist, dass alle Functors als `static inline` definiert sind. Dadurch ergeben sich Optimierungen, die dem Compiler nicht möglich wären, wenn erst zur Laufzeit feststehen würde, welche Functoraufrufe aufeinander folgten. Gerade die Aufeinanderfolge von Multiplikation und Addition kann u. U. durch die oft auf Rechenbeschleunigern hochoptimierte *Fused multiply-add* Operation ersetzt werden, was eine Abarbeitung beider Operationen in nur einem Takt ermöglicht. Die Genauigkeit sinkt zwar ein wenig gegenüber den einzelnen Ausführungen, aber der Effekt fällt für computergrafische Anwendungen kaum ins Gewicht.

Für die CUDA Implementierung wurde zusätzlich noch eine Optimierung eingebaut, die mithilfe von ALPAKA im Moment noch nicht realisierbar ist, wobei die Funktionalität in einer der nächsten Versionen eingebaut werden soll. Die Speicherhierarchie von Nvidias GPUs ist in vielen Punkten ähnlich der von klassischen

CPU-Systemen: Es gibt einen globalen Arbeitsspeicher, dessen Zugriffsgeschwindigkeit bei über 100 Taktzyklen liegt [Nvi16c], einen rechenhardwarenahen Cache sowie Registern, auf die instantan zugegriffen werden kann. Zusätzlich gibt es bei GPUs jedoch noch weitere optimierte Speicherbereiche, wie z. B. *Shared Memory*, der einen effizienten parallelen Zugriff für viele Kernel ermöglicht (und auch von ALPAKA unterstützt wird). Im Detail soll hier dabei nicht auf alle eingegangen werden, weil sie für ISAAC größtenteils nicht relevant sind.

Neben dem klassischen globalen Speicher, der meist mehrere GB groß ist, besitzen Nvidia GPUs jedoch zusätzlich noch *Constant Memory*, welcher zwar nur 64 KB groß ist, aber dafür besonders performant gecached werden kann. Variablen, die im Constant Memory gespeichert werden, besitzen beim ersten Zugriff eine genauso hohe Zugriffszeit wie zum globalen Speicher, danach bleiben sie jedoch im *Constant Memory Cache* und benötigen für jeden weiteren Zugriff nur noch einen Takt, sind also fast so schnell wie ein Register. Als Nachteil können sie innerhalb des Kernels jedoch nicht geschrieben, sondern nur gelesen werden. Damit eignen sie sich bevorzugt für kleine Mengen von globalen, innerhalb eines Kernelaufrufs konstanten Daten. Im Fall von ISAAC trifft das sowohl auf die Parameter der Functors als auch auf die Funktionspointer der Functor-Chains zu, weshalb sie über den Constant Memory an den Kernel übergeben werden. Weiterhin werden die später näher erläuterte inverse Matrix und die Größen und Positionen der einzelnen Simulationsvolumen, die visualisiert werden sollen, im Constant Memory gespeichert.

Das sind die einzigen Punkte, wo sich die ISAAC Implementierungen von CUDA und ALPAKA unterscheiden.

5.1.3.2 Volumenrenderkernel

Um eine hohe Performance im Volumenrendering zu erreichen, gibt es zwei wichtige Konzepte:

- Alle mehrmaligen und für einen Kernelaufruf invarianten Fallunterscheidungen erfolgen außerhalb der Abtastschleife, am besten sogar außerhalb des Kernels. Zum einen verringert es die Anzahl der genutzten Register in der Schleife bzw. im ganzen Kernel und zum anderen erzeugt es weniger auszuführenden Code pro Iteration.
- Wenn möglich, werden Kernelparameter als Templateparameter übergeben. Anstatt eines generischen Kernels entstehen so sehr viele spezifische Kernel, die speziell auf die konstanten Templateparameter hin optimiert werden können. Dadurch vergrößert sich zwar die Binärdatei und die Kompilierzeit, dafür wird die Laufzeit wesentlich kleiner. Dazu kommt, dass es in CUDA

eine Maximalgröße von Kernelparameter gibt, die durch Templateparameter nicht beansprucht wird.

Diese beiden Konzepte werden oftmals auch gleichzeitig genutzt. Wenn eine Fallunterscheidung auf einem konstanten Templateparameter arbeitet, kann sie automatisch aus den Kernel heraus optimiert werden. Auch die schon besprochene Optimierung der Functor-Chains erfüllt diese beiden Paradigmen.

Abtaststrahl berechnen Der Abtaststrahl arbeitet im Raum des lokalen Subvolumens. Modelview- und Projektionsmatrix arbeiten jedoch im Beobachtungsraum, der davon ausgeht, dass das gesamte globale Volumen einen Quader mit der maximalen Kantenlänge 2 darstellt, die jedoch mindestens einmal vorkommt.

Der klassische Weg in der Computergrafik geht von einem Punkt im Raum zu einem projizierten Punkt auf dem Bildschirm. Dazu wird zuerst die 4×4 Modelviewmatrix M, welche die Kameraposition und Rotation sowie Skalierung und Scherungen abbildet, mit dem dreidimensionalen Punkt $\overline{x}$ multipliziert. Unter anderem dafür besitzt $\overline{x}$ noch eine vierte W-Komponente, die auf 1 gesetzt ist. Ignoriert man die Z- und W-Komponente, erreicht man eine Parallelprojektion, in der nahe und ferne Objekte gleich groß erscheinen. Für eine perspektivische Projektion wird das Ergebnis wiederum mit der 4×4 Projektionsmatrix P multipliziert und anschließend der resultierende Punkt durch seine eigene W-Komponente dividiert. Folglich ergibt sich der projizierte Punkt $\overline{x}''$ mit

$$\overline{x}' = P \cdot (M \cdot \overline{x}) \overset{\text{Assoziativgesetz}}{=} (P \cdot M) \cdot \overline{x} \tag{5.1.2}$$

$$\overline{x}'' = \frac{\overline{x}'}{\overline{x}'_{\mathrm{w}}}. \tag{5.1.3}$$

Die Projektionsmatrix bildet einen Pyramidenstumpf (*View Frustum)*, der die sichtbaren Bereiche der Szene markiert, auf einen Würfel der Kantenlänge 2 ab, siehe Abbildung 5.2. Alle Bereiche außerhalb dieses Würfels sind nicht sichtbar. Sie liegen vor dem Bildschirm (z_{near}), sind zu weit entfernt (z_{far}) oder liegen nicht in der Sichtfeldpyramide.

Für das Volumenrendering ist nun der Sichtstrahl für einen gegeben Bildpunkt auszurechnen. Es werden also zwei $\overline{x}$ aus Gleichung 5.1.2 gesucht, für die $\overline{x}'_{\mathrm{x}}$ und $\overline{x}'_{\mathrm{y}}$ bekannt und für beide Punkte gleich sind. Die Projektionsmatrix P ist in ISAAC mithilfe der schon erwähnten z_{near} und z_{far} und den Dimensionen des Sichtfensters l, r, t und b definiert:

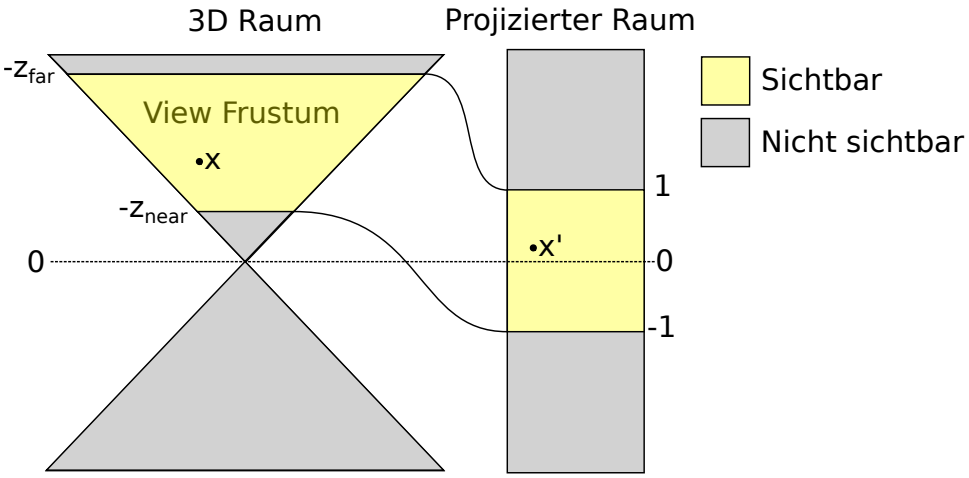

Abbildung 5.2: Darstellung des sichtbaren Pyramidenstumpfes (View Frustum) einer Szene und dessen Entsprechung als Würfel mit Kantenlänge 2 im projizierten Raum. Nur der gelb markierte Bereich ist sichtbar. Der Sichtbarkeitstest reduziert sich auf das trivial lösbare Problem, ob der Punkt im achsenparallelen Würfel liegt (nach [Len15]).

$$P = \begin{pmatrix} \frac{2 \cdot z_{\text{near}}}{r-l} & 0 & \frac{l+r}{r-l} & 0 \\ 0 & \frac{2 \cdot z_{\text{near}}}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{z_{\text{near}}+z_{\text{far}}}{z_{\text{near}}-z_{\text{far}}} & \frac{2 \cdot z_{\text{near}} \cdot z_{\text{far}}}{z_{\text{near}}-z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Unabhängig von den restlichen Komponenten eines schon transponierten Punktes $\overline{x}$ ergibt sich damit die W-Komponente $\overline{x}'_{\text{w}}$ mit

$$\overline{x}'_{\text{w}} = -\overline{x}_{\text{z}}. \tag{5.1.4}$$

Der Einfachheit halber soll nun für die zwei gesuchten Punkte $\overline{x}_1$ und $\overline{x}_2$ gelten, dass einer direkt auf der z_{near} Ebene liegt und der andere auf der z_{far} Ebene. Damit ergibt sich die maximale Strecke im View Frustum zur Berechnung des Blickstrahles. Zudem bildet $\overline{x}_1$ gleichzeitig den Startpunkt des Strahles. In Abbildung 5.2 ist ersichtlich, dass $\overline{x}''_1 = \begin{pmatrix} \overline{x}''_{1_{\text{x}}} & \overline{x}''_{1_{\text{y}}} & -1 & 1 \end{pmatrix}^{\text{T}}$ und $\overline{x}''_2 = \begin{pmatrix} \overline{x}''_{2_{\text{x}}} & \overline{x}''_{2_{\text{y}}} & 1 & 1 \end{pmatrix}^{\text{T}}$ mit gegebenen X- und Y-Komponenten und $\forall \overline{x} : \overline{x}''_{\text{w}} \overset{\text{nach (5.1.3)}}{=} 1$.

Da $\overline{x}_1$ auf der z_{near} Ebene liegt, gilt $\overline{x}_{1_{\text{z}}} = -z_{\text{near}}$ und analog $\overline{x}_{2_{\text{z}}} = -z_{\text{far}}$. Damit gilt laut Formel 5.1.4 $\overline{x}'_{1_{\text{w}}} = z_{\text{near}}$ und $\overline{x}'_{2_{\text{w}}} = z_{\text{far}}$. Somit ergeben sich $\overline{x}'_1$ und $\overline{x}'_2$ nach Gleichung 5.1.3 zu

$$\overline{x}'_1 = \begin{pmatrix} \overline{x}''_{1_x} \cdot z_{\text{near}} & \overline{x}''_{1_y} \cdot z_{\text{near}} & -z_{\text{near}} & z_{\text{near}} \end{pmatrix}^{\text{T}} \tag{5.1.5}$$

$$\overline{x}'_2 = \begin{pmatrix} \overline{x}''_{2_x} \cdot z_{\text{far}} & \overline{x}''_{2_y} \cdot z_{\text{far}} & z_{\text{far}} & z_{\text{far}} \end{pmatrix}^{\text{T}} \tag{5.1.6}$$

und folglich berechnet sich $\overline{x}$ nach Gleichung 5.1.2 durch

$$\overline{x} = (P \cdot M)^{-1} \overline{x}', \tag{5.1.7}$$

da $\overline{x}' = ((P \cdot M) \cdot \overline{x}) \overset{\text{nach (5.1.7)}}{=} \Big((P \cdot M) \cdot (P \cdot M)^{-1 \cdot} \overline{x}'\Big) = I \cdot \overline{x}' = \overline{x}'$.
Mithilfe der Inversen von $P \cdot M$ und den in Gleichung 5.1.5 und 5.1.6 definierten Punkten lässt sich somit der Abtaststrahl im sichtbaren Raum berechnen. Sowohl der Raum des globalen Volumens als auch der sichtbare Raum besitzen dasselbe Zentrum $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^{\text{T}}$ und sind nicht zueinander rotiert, sondern nur skaliert. Damit ergeben sich die globalen Volumenkoordinaten mittels der Mächtigkeit der größten Dimension max_g zu

$$\overline{x}_\text{g} = \overline{x} \cdot \text{max}_\text{g}$$

und die Verschiebung im lokalen Subvolumen mittels der Position $\overline{p}_\text{l}$ zu

$$\overline{x}_\text{l} = \overline{x}_\text{g} + \frac{\text{max}_\text{g}}{2} - \overline{p}_\text{l}, \tag{5.1.8}$$

da innerhalb des lokalen Subvolumens die linke, obere, vordere Ecke den Nullpunkt darstellt und alle gültigen Koordinaten positiv sind. Daraus ergibt sich aus $\overline{x}_1$ im Beobachtungsraum der Startpunkt $\overline{x}_\text{s}$ im lokalen Volumenraum und aus $\overline{x}_2$ analog der Endpunkt $\overline{x}_\text{e}$. Daraus wiederum ergibt sich der Vektor $\vec{v} = \overline{x}_\text{e} - \overline{x}_\text{s}$ sowie der Vektor mit der Schrittlänge d zu $\vec{v}_\text{d} = \frac{\vec{v}}{|\vec{v}|} \cdot d$ und schlussendlich der Sichtstrahl

$$g(i) = \overline{x}_\text{s} + \vec{v}_\text{d} \cdot i \tag{5.1.9}$$

Es ergäbe nun keinen Sinn, von $\overline{x}_\text{s}$ aus mit $\vec{v}_\text{d}$ zu iterieren, bis $\overline{x}_\text{e}$ erreicht ist, da für gewöhnlich große Bereiche ohne Werte sind. Wenn man den Eintritts- und Austrittspunkt mithilfe der Größe des lokalen Volumens berechnet, werden nicht nur wesentlich weniger Schritte gebraucht, sondern es kann auch eine Bereichsüberprüfung pro Abtastschritt entfallen, da sichergestellt ist, dass alle Zugriffe einen gültigen Bereich betreffen.

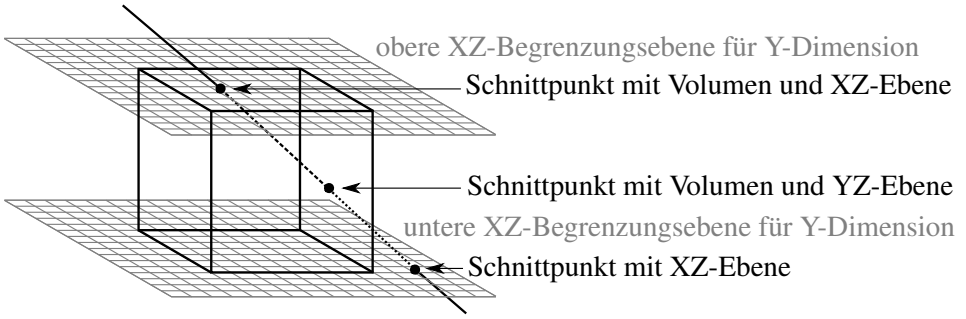

Abbildung 5.3: Schnittpunkte eines Sichtstrahles mit dem lokalen Volumen und den begrenzenden XZ-Ebenen, die orthogonal zur Y-Dimension aufgespannt werden. Durch Betrachtung aller sechs Ebenen aller drei Dimensionen kann die Abtaststrecke bestimmt werden.

Dazu wird berechnet, wie viele Schritte pro Dimension nötig sind, um in den Wertebereich des Subvolumens ein- und wieder auszudringen. Durch die Verschiebung um $\frac{\max_g}{2}$ in Gleichung 5.1.8, spannt sich das lokale Subvolumen nun von $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ bis zur lokalen Größe des Volumens $\vec{s}$ aus. Es ergibt sich für jede Dimension Δ ein Intervall D_Δ, welches alle i beschreibt, für die $g(i)$ zwischen den Ebenen liegt, die orthogonal zu dem Basisvektor $\vec{e}_\Delta$ der Δ-Dimension liegen und die Stützpunkte $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ und $\vec{s}$ haben. Abbildung 5.3 zeigt dies exemplarisch für die XZ-Ebenen, die orthogonal zu dem Basisvektor $\vec{e}_y$ der Y-Dimension aufgespannt werden.

Die Schnittmenge D aller drei Intervalle D_x, D_y und D_z ergibt dann den Eingangs- und Ausgangspunkt in und aus dem Volumen. Wenn die Schnittmenge leer ist, trifft der Strahl den von den Ebenen beschriebenen Quader gar nicht und kann verworfen werden.

Clippingebenen Kurz vor der Raycastinghauptschleife wird geprüft, ob Clippingebenen die Abtaststrecke schneiden oder ob die Abtaststrecke sogar komplett hinter einer Clippingebene liegt. Dazu wird die Geradengleichung 5.1.9 in die in der Normalform vorliegenden Ebenengleichung

$$(\overline{x} - \overline{p}) \cdot \vec{n} = 0 \tag{5.1.10}$$

eingesetzt, um den Parameter i zu bestimmen, für den die Gerade $g(i)$ die Ebene schneidet:

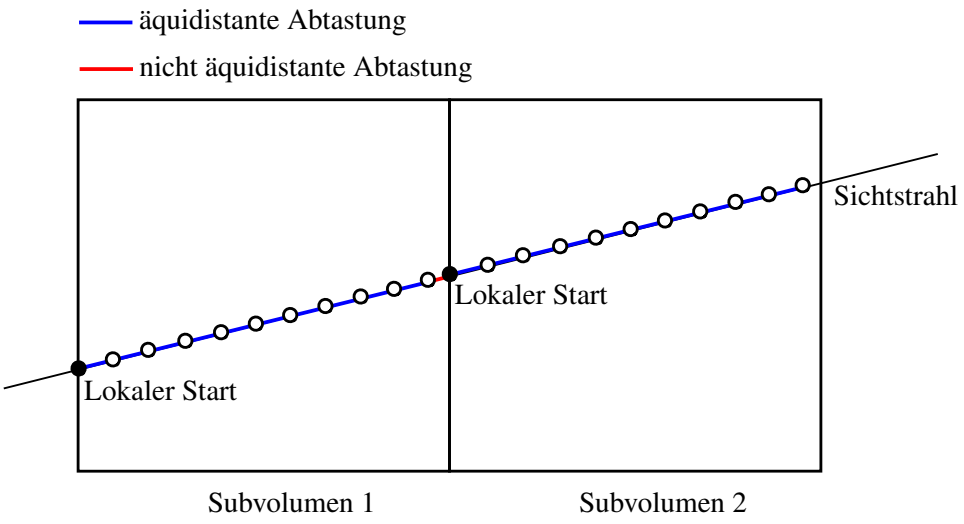

Abbildung 5.4: Problem der ungleichmäßigen Abtastabstände bei parallelem Volumeraycasting. Der rot markierte Abstand wird fehlerhaft in das Gesamtbild eingehen.

$$
\begin{aligned}
(\overline{x}_\mathrm{s} + \vec{v}_\mathrm{d} \cdot i - \overline{p}) \cdot \vec{n} &= 0 \\
\overline{x}_\mathrm{s} \cdot \vec{n} + \vec{v}_\mathrm{d} \cdot \vec{n} \cdot i - \overline{p} \cdot \vec{n} &= 0 \\
\vec{v}_\mathrm{d} \cdot \vec{n} \cdot i &= \overline{p} \cdot \vec{n} - \overline{x}_\mathrm{s} \cdot \vec{n} \\
i &= \frac{\overline{p} \cdot \vec{n} - \overline{x}_\mathrm{s} \cdot \vec{n}}{\vec{v}_\mathrm{d} \cdot \vec{n}}
\end{aligned}
$$

Das Verhalten der Clippingebenen wird derart definiert, dass die Raumhälfte, zu der der Normalenvektor $\vec{n}$ zeigt, sichtbar sein soll und die von der Normale abgewendete Seite ausgeblendet. Dann gilt: Wenn $\vec{v}_\mathrm{d} \cdot \vec{n} > 0$, der Winkel zwischen Abtaststrahl und Normale also kleiner als $90°$ ist, sind nur Intervallgrenzen von D gültig, die größer als das gefundene i sind. Ist die unteren Grenze kleiner, muss sie auf i gesetzt werden. Wenn beide Grenzen kleiner sind, liegt die komplette Abtaststrecke in einem nicht sichtbaren Bereich. Für $\vec{v}_\mathrm{d} \cdot \vec{n} < 0$ gilt entsprechend der umgekehrte Fall und die Intervallgrenzen müssen kleiner als i sein. Auf diese Weise wird die Abtaststrecke u. U. noch einmal verkleinert bzw. der Kernel mit der Hintergrundfarbe beendet.

Das eigentliche Volumeraycasting Wenn ein Sichtstrahl mehrere lokale Subvolumen betritt, muss sichergestellt werden, dass aus Sicht des globalen

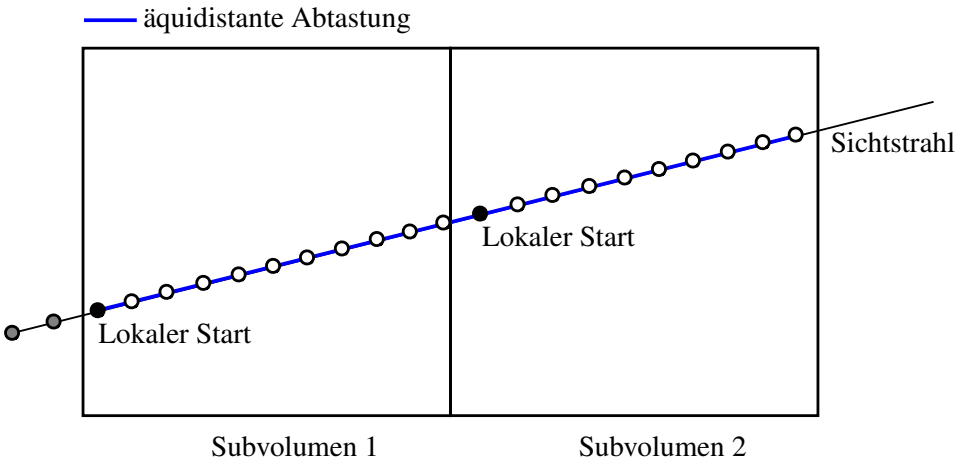

Abbildung 5.5: Lösung für äquidistantes Abtasten bei parallelem Volumeraycasting, indem nur ganzzahlige Vielfache des Abtastabstandes beginnend bei einem globalen Fixpunkt auf dem Sichtstrahl benutzt werden. Es entstehen keine Fehlabstände mehr.

Volumens die Abstände zwischen den Abtastpunkten konstant bleiben. In Abbildung 5.4 ist ein Fall zu sehen, in welchem der kleinere Wert von D als Startwert genommen wird und äquidistant mit d iteriert wurde, bis ein Punkt außerhalb der größeren Grenze von D liegt. Macht man das parallel für alle betroffenen Subvolumen, entstehen die rot markierten, zu kurzen Abtastabstände.

ISAAC löst dies, indem der Eintrittsschritt stets auf die nächste ganze Zahl auf- und der Austrittsschritt abgerundet wird (Abbildung 5.5). Auf diese Art und Weise besitzen alle Abtastpunkte zu ihren Nachbarn einen Abstand von d – unabhängig davon, auf welchem Knoten dieser Abtastpunkt liegt. Die Richtung der Rundung wird so gewählt, dass die resultierenden Punkte in jedem Fall innerhalb des lokalen Subvolumens liegen.

Dadurch ergibt sich eine feste Anzahl an Schritten, die in einer for-Schleife durchiteriert werden können. Der an C++ angelehnte, aber auf das Wesentliche reduzierte Pseudocode der Renderschleife ist in Algorithmus 6 zu sehen.

Die Initialisierung wurde schon hinreichend gezeigt und soll an dieser Stelle nicht in Pseudocode wiederholt werden. Es soll jedoch Augenmerk auf den Parameter `bool using_iso` gelegt werden, welcher nicht als Funktions- bzw. Kernelparameter, sondern als typisierter Templateparameter übergeben wird, was später relevant wird. Die Definition von `first` und `last` aus dem Intervall D wurde schon motiviert. Damit wird nun eine Schleife gestartet, die zuerst den Punkt im lokalen Volumen nach der hergeleiteten Formel 5.1.9 berechnet. Für diesen wird

```
1  template < bool using_iso > void render_kernel( ... )
2  {
3      //...
4      for (int i = first; i <= last; i++)
5      {
6          //...
7          float3 x = x_start + v_d * i; //g(i)
8          compiletime_iter( merge_iterator< using_iso >(), sources, x, ... );
9          if ( using_iso )
10         {
11             //Isoflächendarstellung
12             color = ...;
13         }
14         else
15         {
16             //Emissions-Absorptions-Modell
17             color = ...;
18         }
19     }
20     draw_pixel( ... , color );
21 }
```

Algorithmus 6: Grundkonzept der Hauptrenderschleife des Volumenraycastings von ISAAC. Je nach Templateparameter `using_iso` entstehen zwei verschiedene Kernelspezialisierungen, die entweder nur den Code von Zeile 10 bis 13 oder von Zeile 15 bis 18 enthalten.

mit `compiletime_iter( merge_iterator< using_iso >(), sources, x, value)` die kombinierte Farbe und der Alphawert aller betrachteten Quellenarten berechnet. Die genaue Funktionsweise und Besonderheit wird später erläutert. Für jetzt ist nur relevant, dass die Funktion am Ende den kombinierten Farb- und Alphawert zurückgibt.

In Zeile 9 folgt eine Fallunterscheidung, ob dieser Kernel ein Isoflächenrendering oder eine Darstellung nach dem Emissions/Absorptions-Modell generieren soll. Da `using_iso` ein konstanter Templateparameter ist, erfolgt diese Abfrage nicht in jedem Schleifendurchlauf, sondern bereits zur Kompilierzeit. So können die nie aufgerufenen Zweige herausoptimiert werden. Auf diese Art und Weise können beide Darstellungsarten große Teile des Codes gemeinsam nutzen, im Kompilat entstehen trotzdem zwei optimierte Spezialkernel für jede Darstellungsart.

Außer der Early Ray Termination ist keine andere Schleifenoptimierung, die in den Grundlagen vorgestellt wurden, implementiert worden. Das liegt daran, dass aufgrund der unbekannten Beschaffenheit der Quellen kein Empty Space Skipping implementiert werden kann. Dieses braucht für die Erkennung leerer Bereiche, ohne aktiv durch sie zu iterieren, Beschleunigungsdatenstrukuren wie z. B. Octrees, deren Generierung dem Konzept ISAACs widersprechen.

Um in Zeile 8 die Klassifikation aus den Quellen zu berechnen, wird Template Metaprogrammierung genutzt. Wie in den Grundlagen dargestellt, ist die Meta-

```
1  template < bool using_iso > struct merge_iterator
2  {
3      template < typename TSource, typename TX, ... >
4      void operator() ( TSource &source, TX &x, ... )
5      {
6          float value = source.chosen_functor_chain( source[ int3(x) ] );
7          float4 add = get_color_from_transfer( value );
8          if ( using_iso )
9          {
10             //Berechnung für Isoflächendarstellung
11         }
12         else
13         {
14             //Berechnung für das Emissions-Absorptions-Modell
15         }
16     }
17 };
```

Algorithmus 7: Zur Kompilierzeit für jedes Element einer boost::fusion::list ausgeführter Funktor. Dieser generische Funktor kann je nach Templateparamter für das Emissions/Absorptions-Modell oder Isoflächendarstellung spezialisiert werden. Der jeweils ungenutzte Code wird herausoptimiert.

programmierung in C++ funktional und wird mithilfe von Structs bzw. Klassen implementiert, die sich rekursiv selbst aufrufen bzw. instanziieren und dabei einen Templateparameter de- oder inkrementieren bis dieser einen bestimmten Wert erreicht.

Für ISAAC wurde das in Algorithmus 4 gezeigte `boost::fusion::for_each` um die Möglichkeit erweitert, zusätzliche Parameter als Referenzen zur sequentiellen Bearbeitung mit zu übergeben. Die Funktion `compiletime_iter` entspricht dieser überarbeiteten Version, welcher eine beliebige Anzahl beliebig typisierter Referenzparameter übergeben werden kann, welche wiederum an den `()` Operator der Iterationsstruct übergeben werden. Für den exemplarischen Pseudocode 6 sähe die Struct wie in Algorithmus 7 zu sehen aus.

In dem Funktor wird in Zeile 6 mithilfe des `[]` Subscriptoperators, der pro Quellenklasse definiert wurde, ein Datum aus der Quelle geladen und die Functor-Chain für diese Quelle ausgeführt. Etwaige Interpolation wird in diesem Pseudocode nicht weiter betrachtet, stattdessen wird der Gleitkommavektor `x` in einen Ganzzahlvektor vom Typ `int3` konvertiert. Auch ist der Funktionspointer der Functor-Chain kein Element der Klasse `Source`, sondern liegt zumindest für CUDA als `__constant__` Variable im globalen (Device) Speicher. Der Einfachheit halber wurden die meisten Details und Optimierungen des realen Codes aber nicht in dieses Beispiel übernommen.

In Zeile 7 wird der Wert der Quelle mit der Transferfunktion klassifiziert. ISAAC nutzt hierzu ein einfaches Array, in welchem der Wert nachgeschlagen wird. Da-

nach wird analog zu Algorithmus 6 zwischen den beiden Darstellungsarten unterschieden. Im Falle der Isoflächendarstellung wird geprüft, ob die Klassifizierung einen Alphawert größer $0{,}5$ ergibt. In diesem Fall wird die Farbe des Pixels gesetzt sowie eine Abbruchvariable auf `true` gesetzt, um anzugeben, dass die Schleife terminieren kann, weil ein Datum gefunden wurde. Da der konkrete Alphawert für die Isoflächendarstellung irrelevant ist, würde sich auch jeder andere Wert zwischen 0 und 1 anbieten. Der Wert $0{,}5$ hat jedoch den Vorteil, dass ab eben dieser Klassifizierung eine Oberfläche gezeichnet wird, die auch im Emissions/Absorptions-Modell sehr opak gezeichnet wäre.

Mithilfe des Gradienten, welcher auch über den `[]` Subscriptoperator der Quelle berechnet wird, wird die Normale der Oberfläche approximiert. Des Weiteren wird der Einfachheit halber davon ausgegangen, dass die Lichtquelle direkt hinter der Kamera liegt, womit die Lichtquellenrichtung der Schrittrichtung entspricht. Die beiden normalisierten Vektoren werden miteinander multipliziert, um den Kosinus des Winkels zwischen diesen zu erhalten, wie er in Formel 3.2.2 des Phong-Beleuchtungsmodells gebraucht wird. Die ambiente Farbe sei schwarz ($I_{\text{ambient}} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^{\text{T}}$) und die Lichtquelle rein weiß mit $I_{\text{in}} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^{\text{T}}$. k_{diffus} entspreche weiterhin der klassifizierten Farbe `add` und $k_{\text{spekular}} = {}^1\!/_3\,\pi\,k_{\text{diffus}}$. Zu guter Letzt sei $n = 4$. Dann ergibt sich für die Gesamtfarbe $I_{\text{out}} = k_{\text{diffus}} \cos\varphi + {}^1\!/_3\,\pi\,k_{\text{diffus}} \frac{6}{2\,\pi} \cos^4\theta = k_{\text{diffus}} \cos\varphi + k_{\text{diffus}} \cos^4\theta$. Diese einfache, schnelle, aber trotzdem ansehnliche Lichtberechnung wird innerhalb des Isoflächenpfades ausgeführt.

5.1.3.3 Aufruf des Renderkernels

Wie im vorherigen Abschnitt angedeutet, werden in der Realität sehr viele Parameter des Kernels als Templateparameter übergeben. Das hat einerseits den Vorteil, dass viele spezialisierte Kernelvarianten entstehen können, die schneller als generische Versionen laufen. Andererseits entsteht aber das Problem, dass diese Kernelaufrufe explizit mit konstanten Werten bzw. Typen aufgerufen werden müssen. Wenn beispielsweise viele Quellen definiert und an den Kernel übergeben werden, will man meist nicht alle auf einmal anzeigen. Das Gewicht einzelner Quellen auf 0 zu setzen, würde sie jedoch immer noch in der Berechnung mit einbeziehen und unnötig Ressourcen binden. Deshalb ist ein Templateparameter des ISAAC Kernels ein `boost::mpl::vector` von Booltypen, die angeben, ob eine Quelle aktiv ist oder nicht. Damit entstehen 2^q verschiedene Kernel, wobei q die Anzahl der Quellen beschreibt. Um den richtigen Kernel auszuwählen, wird deshalb rekursiv durch die Quellen iteriert und je nachdem, ob das Gewicht Null ist, entweder ein

`boost::mpl::true_` oder `boost::mpl::false_` zu dem MPL::Vector hinzugefügt, der dann als Templateparameter dem Kernel übergeben wird. Auf diese Art und Weise lassen sich durch einfache rekursive Schleifen zur Kompilierzeit beliebig große binäre Entscheidungsbäume für die verschiedenen Kernelvarianten erzeugen.

An dieser Stelle zeigt sich der Grund, wieso die Functor-Chains als Funktionspointer implementiert wurden und nicht auf ähnliche Art und Weise viele spezialisierte Kernel für jede Kombination der Quellen und Functor-Chains erstellt werden. Erweiterte man Formel 5.1.1 um diese Idee, entstünden so $2^q \cdot (c^n)^q = 2^q \cdot c^{nq}$ verschiedene Kernel, da nicht nur jede Kombination von Quellen Beachtung finden müsste, sondern auch jeder Kernel eine andere Functor-Chain besitzen kann und auch diese beliebig miteinander kombiniert werden müssten. Für realistische Werte von $c = 5$, $n = 3$ und nur vier Quellen $q = 4$ entstünden so $62\,500\,000\,000$ verschiedene Kernel gegenüber $2^q = 16$ Kerneln zuzüglich den schon erwähnten 500 Functor-Chains. Es wird sich später zeigen, dass schon die Generierung von 500 Functor-Chains sehr zeitintensiv in der Kompilierung sein kann. Das Kompilieren von über sechzigmilliarden Kerneln ist mit heutiger Hardware nicht in endlicher Zeit möglich.

In den Grundlagen wurde kurz erläutert, dass sowohl CUDA als auch ALPAKA zwei verschiedene Gitterdimensionen besitzen, auf denen die Kernel arbeiten. Eine Besonderheit von CUDA ist, dass immer 32 Kernel eines Blocks (also der inneren Dimension) gleichzeitig denselben Befehl ausführen müssen. Solche 32 Kernel bilden einen *Warp*. Insbesondere bei If-Verzweigungen würde ein Teil des Warps warten, wenn nicht alle 32 Kernel denselben Zweig nehmen. Die meisten Instruktionen des ISAAC Renderers laufen auf allen Kernels identisch ab. Ein Unterschied entsteht jedoch, wenn verschiedene Kernelinstanzen unterschiedlich oft durch das Volumen iterieren. Die Anzahl der Iterationsschritte hängt von der Länge der Strecke im lokalen Subvolumen ab und ob der Strahl aufgrund vieler sehr opaker Klassifikationen frühzeitig abbricht. Es kann davon ausgegangen werden, dass nebeneinander liegende Sichtstrahlen ähnliche Längen und ähnliche Abbruchswahrscheinlichkeiten besitzen. Deshalb sollte ein Warp einen möglichst kleinen Bereich des Bildschirms abdecken, optimal also ein Quadrat von $\sqrt{32} \cdot \sqrt{32}$ Pixeln. Der Abstand des linken, oberen Pixels dieses Warps zu dem rechten unteren wäre hierbei optimal. In der Realität rundet man auf Zweierpotenzen, da sie eine gute Auslastung der Speicherbandbreite und der Caches versprechen. Jedoch stellt ein Block mit 8×4 Threads – also genau einem Warp – nicht die optimale Ausführungsgeschwindigkeit sicher, da ein *Streaming Multiprozessor* (SM) einer Nvidia K20 maximal 16 Blöcke aber dafür 2048 Threads gleichzeitig aktiv haben kann. Aktiv bedeutet hierbei, dass ein Thread geladen ist und seine eigenen Register reserviert hat, aber nicht zwingend, dass er auch ausgeführt wird. Wenn jeder Block nur ge-

nau 32 Threads enthält, können pro SM also nur 512 Threads aktiv sein. Ein SM hat zwar meist nur 192 CUDA Cores für die Ausführung von Threads. Aber es ergibt Sinn so viele Threads wie möglich aktiv zu haben, damit die Wartezeit für Speicheranfragen minimiert wird (*Latency Hiding*): Wenn ein Thread ein Datum liest, kann seine Ausführung unterbrochen und ein anderer aktiver Thread weiter ausgeführt werden. Somit sollten alle 2048 Threads aktiv sein. Mit der optimalen Blockgröße 8×16 sind genau 2048 Threads pro SM aktiv.

5.1.4 Paralleles Volumenrendering

Das parallele Rendering wird komplett von IceT übernommen. Um die Zwischenbilder des Raycastings zusammenfügen zu können, ist die Reihenfolge dieser wichtig. Es ist zwar möglich, IceT dafür die Sortierung der Bilder mit `icetCompositeOrder` zu übergeben, aber nicht jede Kombinierstrategie von IceT untersützt diese Einstellung. Deshalb wird sich diese Arbeit in der Evaluation auf die folgenden Strategien beschränken:

- `ICET_STRATEGY_DIRECT` schickt Bilder direkt zu dem anzeigenden Knoten, im Fall von ISAAC dem Master. Für verteilte Systeme liefert dieser Ansatz im Allgemeinen eine schlechte Performance.
- `ICET_STRATEGY_SEQUENTIAL` und `ICET_STRATEGY_REDUCE` wenden beide optimiertere Kombinationsalgorithmen – wie z. B. Binary Swap – an und unterscheiden sich nur in der Abarbeitungsstragie der Knoten, die die Gesamtbilder erhalten sollen. Die erste Strategie arbeitet dabei sequentiell, wohingegen die zweite einen ausgefeilteren Algorithmus nutzt. Da ISAAC nur an einem Knoten ein Gesamtbild empfängt, unterscheiden sich die Strategien hier kaum. Bei beiden lassen sich die Strategien pro Bild einstellen. Zur Verfügung stehen dabei unter anderem der schon erwähnte Binary Swap Algorithmus, aber auch zwei baumbasierte Ansätze. Die Geschwindigkeit der einzelnen Strategien wird in der Evaluation gemessen und bewertet werden.

Für die Sortierung bestimmt jeder Knoten den Abstand der Mitte seines lokalen Volumens zu der Kamera, indem dieser Punkt im Beobachterraum mit der Modelviewmatrix multipliziert wird. Danach ist die Kamera am Punkt $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$, d.h. die Länge des Ortsvektors entspricht dem Abstand zu dieser. Die Abstände werden dann mit `MPI_Allgather` für alle anderen Knoten verteilt, da für Algorithmen wie Binary Swap jeder Knoten über die komplette Sortierung Bescheid wissen muss.

IceT wird nur am Masterknoten ein Gesamtbild zurückgeben, welches dann mithilfe der libjpeg-turbo [lib16] komprimiert wird. Um nur einen Kanal öffnen zu müssen, werden sowohl für die Übertragung des Gesamtbildes als auch für alle Steuer- und Metadaten JSON-Nachrichten genutzt. Auf diese Art und Weise kann auch eine andere, von ISAAC unabhängige Visualisierung den ISAAC Server mitbenutzen bzw. einfach beliebige Metadaten hinzugefügt werden. JSON unterstützt jedoch keine Binärdaten, weshalb das komprimierte Bild zunächst mithilfe von Base64 in eine Zeichenkette überführt werden muss. Andererseits sind moderne Browser in der Lage, solche Zeichenketten direkt als Bild anzuzeigen.

5.1.5 Steuerung der Simulation

Die meisten Parameter der Simulation können über spezielle JSON Nachrichten eingestellt werden; so die Gewichte aller Quellen, die Functor-Chains pro Quelle, die Clippingebenen, aber vor allem auch die Transferfunktion und die Szeneneinstellung. Die Transferfunktion wird dabei über beliebig viele Punkte beschrieben, die einem Skalarwert einen Farb- und Alphawert zuweisen. Daraus berechnet jeder Knoten die diskrete Transferfunktion durch lineare Interpolation für Werte, an denen kein Punkt definiert ist. Die Szeneneinstellung ergibt sich aus vier Parametern:

- Die Position der Kamera ist als drei-elementiges Array angegeben.
- Die Rotation der Kamera ist als 3×3 Matrix definiert. Jedoch muss ein Client oder der Server keine Rotationsmatrizen berechnen können. Es steht den Clients zwar frei, die Rotationsmatrix direkt mit einer anderen 3×3 Matrix zu ersetzen oder zu multiplizieren, nichtsdestotrotz besteht auch die Möglichkeit, eine Rotation um eine Achse mithilfe eines Vektors und eines Winkels anzugeben.
- Die Entfernung zu der Kamera ist eine einfache Gleitkommazahl.
- Zu guter Letzt braucht es noch eine Projektionsmatrix, um die Projektion der Szene zu beschreiben. Diese kann jedoch nur als komplette 4×4 Matrix übertragen werden. In den meisten Fällen wird eine Anpassung dieser Matrix jedoch nicht von Bedeutung sein.

Daneben gibt es noch wenige boolsche Variablen, die gesetzt werden können, wie die Aktivierung des Isoflächenrenderings oder die Interpolation der Datenpunkte. Aufgrund eines Bugs in IceT ist es des Weiteren möglich, die Bounding Box Überprüfung dieser Bibliothek zu deaktivieren. Wenn die Kamera sich in die Bounding Box hinein bewegt, liefert die Berechnung von IceT falsche Werte und es entstehen Darstellungsfehler. Andererseits bietet einem der Boundingbox Check für eine

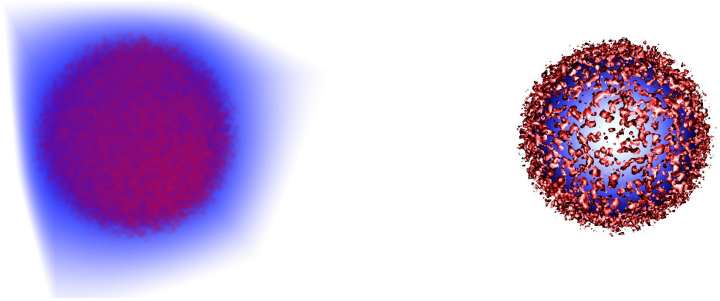

Abbildung 5.6: Testanwendung mit zwei Pseudoquellen. Links erfolgt die Darstellung nach dem Emissions/Absorptions-Modell und rechts mit einer Isoflächendarstellung sowie Interpolation. Mit dem linken Rendering ist die rote Quelle besser darstellbar, rechts die bläuliche.

Kamera innerhalb des Volumen sowieso kaum Performancegewinn, weshalb die Deaktivierung in diesem Fall nicht weiter ins Gewicht fällt.

Weiterhin änderbar ist die Hintergrundfarbe der Visualisierung. Gerade bei der Verwendung in schriftlichen Arbeiten oder auf Internetpräsenzen kann es von Vorteil sein, die Bilder vor weißem Hintergrund zu rendern. Die Schrittweite kann auch jederzeit angepasst werden. Oftmals zeigt sich erst während der Visualisierung, welche Schrittweite sich am besten eignet.

Neben diesen Einstellungen können auch explizit Werte der Simulation abgefragt werden, wie z. B. der minimale und maximale Wert einer Quelle über das gesamte Volumen nach der Anwendung der Functor-Chains. Auf diese Art und Weise können einfacher sinnvolle Functorparameter gefunden werden, damit das Resultat im Wertebereich der Transferfunktionen liegt.

5.1.6 Simulationseinbindungen

Die ISAAC In-Situ Bibliothek wurde im Rahmen der Arbeit in zwei Anwendungen eingebunden. Die eine Anwendung wird mit ISAAC mitgeliefert und stellt nur eine sehr einfache Testanwendung dar, um ISAAC zu testen und seine Möglichkeiten zu evaluieren. Die Anwendung simuliert eine Simulation, indem sie Volumen auf einem Rechenbeschleuniger auf zufällige Werte setzt, und definiert zwei Testquellen, auf die mittels ISAAC zugegriffen werden kann. Die eine Testquelle stellt dabei ein gleichmäßiges, nach außen hin schwächer werdendes Skalarfeld dar und die andere Quelle eine zufällige Verteilung von Dichten, wobei auch hier mittig höhere Werte erzielt werden. Abbildung 5.6 zeigt die beiden Quellen einmal nach

dem Emissions/Absorptions-Modell und einmal mit einer Isoflächendarstellung gerendert. Es fällt auf, dass sich die zufälligen Werte der roten Quelle nicht gut für die Isoflächendarstellung eignen, wohingegen bei der gleichmäßigen blauen Quelle eine Kugel entsteht. Später soll durch diese Anwendung simulationsunabhängig die Geschwindigkeit der Visualisierung in Abhängigkeit von Volumengröße, Bildauflösung und anderen Parametern evaluiert werden.

Des Weiteren wurde ein Plugin für PIConGPU implementiert, was auf die realen, wissenschaftlichen Daten dieser Plasmasimulation zugreift. Mithilfe dieses Plugins soll evaluiert werden, inwieweit sich ISAAC eignet, in bestehende Simulationen integriert zu werden, und welchen Einfluss das sowohl auf die Lauf-, als auch auf die Kompilierzeit der Anwendung hat. Das ISAAC-Plugin für PIConGPU kann auf das elektrische, magnetische und das Stromstärkenvektorfeld der Simulation zugreifen. Des Weiteren ist es möglich, die Dichte von Elektronen- und Ionenpartikelfeldern anzuzeigen.

Für solche Zwecke bietet PIConGPU ein Pluginsystem an. Ein Plugin muss dazu von der Klasse `ILightweightPlugin` erben und wenige Methoden überschreiben:

- Der Konstruktur kann für Initialisierungen genutzt werden, die unabhängig von Simulation und Rechenbeschleuniger sind. Außerdem muss er genutzt werden, das Plugin bei PIConGPUmittels `Environment<>::get().PluginConnector().registerPlugin(this)` zu registrieren.
- Den Namen des Plugins erhält die Simulation mit der Methode `pluginGetName()`, die einen std::string als Rückgabeparameter erwartet.
- Die eigentliche Initialisierung von ISAAC erfolgt in der Methode `pluginLoad()`. Diese wird aufgerufen nachdem das richtige CUDA-Gerät eingerichtet wurde, aber bevor PIConGPU seinen Speicher alloziert. Für eine schnellere Speicherverwaltung alloziert PIConGPU nach dem Laden aller Plugins den gesamten, restlichen Speicher des Devices und verwaltet ihn dann intern selbst. In Hinblick auf solch ein Verhalten ist ISAAC darauf hin implementiert worden, nach dem Initialisieren keinen Speicher mehr zu allozieren. In dieser Methode wird das Visualisierungsobjekt erstellt, Metadaten beschrieben und die Visualisierung initialisiert.
- Die Parameter für die Initialisierung können direkt an PIConGPU übergeben werden. Dazu füllt jedes Plugin beim Aufruf der Methode `pluginRegisterHelp(po::options_description& desc)` die Struktur `desc` mit möglichen Optionen und deren Standardwerten. ISAAC nutzt dies für die Einstellung der Server-URL, des Serverports, der Bildauflösung, dem Namen der Simulation

und der Periode, in welcher die Visualisierungsfunktion aufgerufen werden soll.

- Wenn die Simulation schließlich läuft, wird alle n Zeitschritte die Methode `notify(uint32_t curStep)` mit dem momentanen Zeitschritt als Parameter aufgerufen. Die Periode n ist frei wählbar, wird für ISAAC aber auf 1 gesetzt. Die in der vorher beschriebenen Methode definierte Periode wird innerhalb von `notify` mit einem internen Schrittzähler umgesetzt, damit die Periode nachträglich von einem Client angepasst werden kann. In dieser Methode werden die Metadaten gefüllt, `doVisualization` aufgerufen sowie eingehende Metadaten ausgewertet. Wenn ein Client die Nachricht schickt, dass die Simulation pausiert werden soll, kehrt das Plugin nicht mehr aus der Methode zurück, bis der Pausenbefehl wieder aufgehoben wurde. `doVisualization` wird trotzdem noch aufgerufen, zum einen, um auf neue Nachrichten zu überprüfen und zum anderen um weiterhin Bilder zu rendern. Auf diese Art und Weise können trotz pausierter Simulation die Szeneneinstellungen geändert und das Standbild aus allen Winkeln betrachtet werden.
- Wenn die Simulation beendet wurde, wird die Methode `pluginUnload` aufgerufen, die genutzt wird, um das ISAAC Visualisierungsobjekt zu löschen.

Die Klassen für die Quellen werden in der `pluginLoad` Methode zwar instantiiert, jedoch außerhalb definiert. Es werden drei verschiedene Typen von Klassen definiert. Eine für persistente Quellen mit sinnvollem Guard, wie die elektrischen und magnetischen Vektorfelder, eine für Quellen, die zwar persistent sind, aber keinen sinnvollen Guard besitzen, wie das Stromstärkenvektorfeld, und eine für Quellen, die nicht persistent sind und ISAAC-intern gesichert werden müssen, wie die Elektronen- oder Ionendichten. In den `update` Methoden dieser Klassen werden über global verfügbare, PIConGPU-spezifische Funktionen Pointer auf die Simulationsdaten auf den GPUs für die spätere Verwendung in den `[]` Subscriptoperatoren zwischengespeichert bzw. für nicht persistente Quellen diese Speicherbereiche überhaupt erst mit sinnvollen Daten gefüllt.

Gerade in der Laser-Kielfeld-Beschleuniger-Simulation von PIConGPU bewegt sich das interessierende Objekt durch das Volumen und für eine langfristige Beobachtung müssten sehr viele GPUs reserviert werden, wovon immer nur wenige den Laserpuls und die dahinter beschleunigten Elektronen zeigen würden. Deshalb gibt es einen *Moving Window* Modus, in welchem sich die Positionen der lokalen Subvolumen relativ zu dem globalen Volumen ändern und das Untersuchungsobjekt somit immer in der Mitte der Simulation bleibt und immer wieder hintere GPUs wieder nach vorne verlagert werden, wenn sie zu weit von dem Untersuchungsobjekt entfernt sind. Wenn dieser Modus aktiviert ist, werden in der `notify`

```
1  class MetaDataConnector : public Runable, public MessageAble<MsgContainer>
2  {
3      public:
4          //To be overwritten
5          virtual errorCode init(int port) = 0;
6          virtual errorCode run() = 0;
7          virtual std::string getName() = 0;
8          //Called from the Master
9          void setMaster(Master* master);
10     protected:
11         Master* master;
12 };
```

Algorithmus 8: `MetaDataConnector`-Klasse zur Metadatenübertragung an Clients. Durch die abstrakte Definition können beliebig viele unterschiedliche Verbindungen zum ISAAC Server implementiert werden.

Methode auch die Positionen und Größen der lokalen Volumen derartig angepasst, dass ein gleichmäßiges Betrachten möglich ist. Abgesehen von kurzen Pausen von PIConGPU, wenn hintere GPUs von hinten nach vorne geschoben und dabei neu initialisiert werden, sind diese Anpassungen nicht bemerkbar.

5.2 ISAAC Server

Der ISAAC Server hat zweierlei Aufgaben. Zum einen ist er eine vermittelnde Instanz, die Nachrichten vom Master zu Clients und umgekehrt durchreicht. Andererseits hat er die Aufgabe Videostreams aus den ankommenden Bilddaten zu generieren. Um sich nicht auf eine Art von Client festlegen zu müssen und offen für zukünftige Streamingarten und Displaytechnologien zu sein, arbeitet der Broker mit zwei abstrakten Klassenarten, die auf die wesentlichen Funktionen reduziert sind.

5.2.1 Abstraktes Pluginsystem

Die erste abstrakte Klasse `MetaDataConnector` (Algorithmus 8) bietet ein einfaches Interface für beliebige Clienttypen an. Über `init` wird eine konkrete Instanz initialisiert und zur Laufzeit `run` in einem dedizierten Thread aufgerufen. Aufgabe dieser Methode ist, auf Nachrichten vom Server zu warten und entsprechend an verbundene Clients weiterzureichen bzw. einkommende Nachrichten von Clients an den Server zu schicken. Dazu erbt diese Klasse von `MessageAble`, siehe Algorithmus 9.

Die Aufgabe dieser Klasse ist es, threadsicher Nachrichten vom Server zu Connectors und von Connectors zum Server zu senden. Inhalt dieser Nachrichten sind dabei die JSON Pakete. Diese Klasse wird auch vom `ImageConnector` (Algorithmus

```
template <typename MessageTemplate>
class MessageAble
{
    public:
        virtual ~MessageAble();
        //Called from Connector
        errorCode clientSendMessage(MessageTemplate* message);
        MessageTemplate* clientGetMessage();
        //Called from Server
        errorCode masterSendMessage(MessageTemplate* message);
        MessageTemplate* masterGetMessage();
};
```

Algorithmus 9: `MessageAble`-Klasse zur Intraklassenkommunikation im ISAAC Server. Alle Klassen, die von dieser erben, erhalten die Möglichkeit threadsicher Nachrichten mit dem Haupthread auszutauschen. Auf diese Art und Weise kann ISAAC stark nebenläufig arbeiten und Aufgaben auf viele Rechenkerne verteilen.

```
class ImageConnector : public Runable, public MessageAble<ImgContainer>
{
    public:
        ImageConnector();
        //To be overwritten
        virtual errorCode init(int minport,int maxport) = 0;
        virtual errorCode run() = 0;
        virtual std::string getName() = 0;
        //Called from the Master
        void setMaster(Master* master);
        bool showClient;
    protected:
        Master* master;
};
```

Algorithmus 10: `ImageConnector`-Klasse zur Streamübertragung an Clients. Durch die abstrakte und auf das Wesentliche reduzierte Definition können beliebige Streamtypen implementiert werden.

10) genutzt. Die Aufgabe dieser Klasse ist analog zum `MetaDataConnector` die Übertragung von Streams zu angemeldeten Clients. Genauso wie `MetaDataConnector` wartet sie auf Nachrichten vom Server und schickt sie an die Clients weiter. Nur werden diesmal keine JSON Nachrichten, sondern Bilder vom Server empfangen. Jedes Bild hat dabei einen Referenzzähler, sodass immer nur eine Instanz dieses Bildes im Speicher gehalten werden muss und alle Implementierungen von `ImageConnector` parallel lesend darauf zugreifen können.

5.2.2 Implementierung der Klasse MetaDataConnector

Im Rahmen dieser Diplomarbeit wurde nur eine Implementierung von `MetaDataConnector` für die Kommunikation mit einem Webclient vorgenommen. Durch den Webclient ist diese Referenzimplementierung unabhängig von dem eingesetz-

ten Betriebssystem. ISAAC ist als asynchroner Dienst gedacht, der Daten sofort schickt, nachdem sie entstanden sind. Deshalb ist das klassische HTTP Protokoll ungeeignet für die Kommunikation zwischen Server und Client. Stattdessen wird die HTML5 Erweiterung Websockets genutzt, die auf *HTTP UPGRADE* basiert. Mit dessen Hilfe kann eine HTTP Verbindung zu einem beliebigen anderen Protokoll geändert ("upgegraded") und aufrecht erhalten werden. Insbesondere ist es möglich, direkt mit Javascript auf eingehende Nachrichten zu warten und zu reagieren.

Da ISAAC selbst unter der GNU Lesser General Public License Version 3 steht, wurde zuerst evaluiert, welche Open Source Websocketbibliotheken für C bzw. C++ zur Verfügung stehen. In die nähere Auswahl kamen dabei Mongoose [Ces16], Nghttp2 [Tsu16] und Libwebsockets [Gre16]. Mongoose ist unter der GPL lizenziert. Das würde erfordern, dass zumindest der ISAAC Server selbst auch unter die GPL statt der LGPL gestellt wird. Nghttp2 nutzt die sehr freie MIT Lizenz, die ohne Einschränkungen von (L)GPL Programmen genutzt werden kann. Libwebsockets zu guter Letzt nutzt auch selbst die LGPL in der Version 2 oder höher. Damit bieten sich vor allem die letzten beiden Lösungen an. Am Ende wurde sich für Libwebsockets entschieden, da sie leichtgewichtiger und somit ressourcensparender für den Login Node zu sein scheint. Da das umgebene Protokoll jedoch vollständig definiert ist, wäre es auch denkbar die gleiche Funktionalität in Zukunft mit einer anderen Lösung mit relativ wenig Aufwand erneut zu implementieren, sollten unerwartete Probleme mit der gewählten Bibliothek auftreten.

Die JSON Objekte in der Darstellung der Janssonbibliothek, die vom Server an diesen `MetaDataConnector` geschickt werden, werden dabei in eine Zeichenkette konvertiert und direkt an den verbundenen Webclient übertragen. Umgekehrt werden nur JSON Zeichenketten empfangen, direkt mithilfe von Jansson umgewandelt und an den Broker, also die Hauptschleife des Servers, geschickt. In dieser Implementierung erfolgt keine Interpretation der gesendeten und empfangenen Daten.

5.2.3 Implementierungen der Klasse ImageConnector

Die Echtzeitkodiermöglichkeiten von Videos, Netzwerkgeschwindigkeiten und auch die Dekodiermöglichkeiten von Techniken wie HTML5 sind im Moment einem steten Wandel unterzogen. Deshalb war es sehr wichtig, die Klasse `ImageConnector` sehr abstrakt zu halten, um langfristig aktuelle Technologien nutzen zu können. ISAAC hat bisher fünf Implementierungen dieser abstrakten Streamingklasse. Im Rahmen der Recherche zu ISAAC wurde evaluiert, welche Möglichkeiten im HTML5 Referenzclient bestehen, Videostreams anzuzeigen.

In HTML5 übergibt man dem `<video/>` Tag dabei eine beliebige Quelle als Parameter und diese wird abgespielt. Es ist nicht standardisiert, welche Protokolle und welche Codecs unterstützt werden. Es hat sich gezeigt, dass Videostreaming über das HTTP Protokoll von allen Browsern unterstützt wird. De facto wird dazu ein Video auf einem Webserver gehostet und beim Abspielen live im Hintergrund heruntergeladen. Das oft für Streaming oder Voice over IP genutzte RTP Protokoll bzw. die Erweiterung um Steuerinformationen RTSP wird bisher von keinem der getesteten Browser nativ unterstützt – zumindest nicht im Zusammenhang mit dem `<video/>` Tag. Die meisten Browser unterstützen den Codec H.264. Gerade in Open Source Produkten muss dieser aber oft explizit aktiviert werden, da der Codec zwar schnell, durch die Belastung mit Patenten jedoch nicht frei ist. Insbesondere die kommerzielle Nutzung ist u. U. mit Gebühren verbunden. VP8 und dessen Nachfolger VP9 sind patentfreie Alternativen, die von vielen Open Source Lösungen implementiert sind, aber dafür nicht von einigen unfreien Browsern wie dem Internet Explorer, insbesondere in den Versionen 8 und älter.

Um diese Probleme zu umgehen, wurde sich für den Referenzclient von ISAAC zunächst dafür entschieden, auf das freie und für alle großen Systeme verfügbare Browserplugin des VLC Media Players [Vid16] zurückzugreifen. Zum einen ist es so möglich direkt RTP für das Streaming zu benutzen, andererseits ist man sehr frei in der Wahl der Codecs. Zu Evaluierungszwecken wurde zum einen H.264 genutzt und zum anderen ein JPEG-Stream, der aus vielen Einzelbildern besteht. Es darf hierbei nicht der Eindruck entstehen, die Wahl der Implementierung der Streamingklasse hinge direkt mit dem `MetaDataConnector`, in diesem Fall dessen Webclientimplementierung, zusammen. Metadatentransport und Streaming sind zwei unabhängige Aufgaben des ISAAC Servers. Nichtsdestotrotz ergibt es Sinn den Stream zusammen mit den Meta- und Steuerdaten anzuzeigen, weshalb beide Aspekte in die Planung einflossen. Für die Kodierung der Bilder zu einem Videostream und die Kapselung in das RTP Protokoll wurde die freie Software gStreamer [GT16] genutzt. Auch gStreamer ist unter der LGPL lizenziert. Vorteile von gStreamer sind eine umfangreiche Dokumentation und die gute Performance trotz des sehr modularen Aufbaus.

Eine weitere Methode der Streamübertragung ist es, die JPEG Daten in Form einer Base64 *Uniform Resource Identifier* (URI) Zeichenkette der Form `...` in einem JSON Objekt mithilfe eines MetaDataConnectors an den Client zu schicken, damit er diesen direkt anzeigen kann. Auch diese Idee wurde für die Diplomarbeit implementiert. Der große Vorteil liegt darin, dass kein externes Plugin für die Streamanzeige gebraucht wird. Alle modernen Browser sind dabei in der Lage die URI Zeichenketten anzuzeigen.

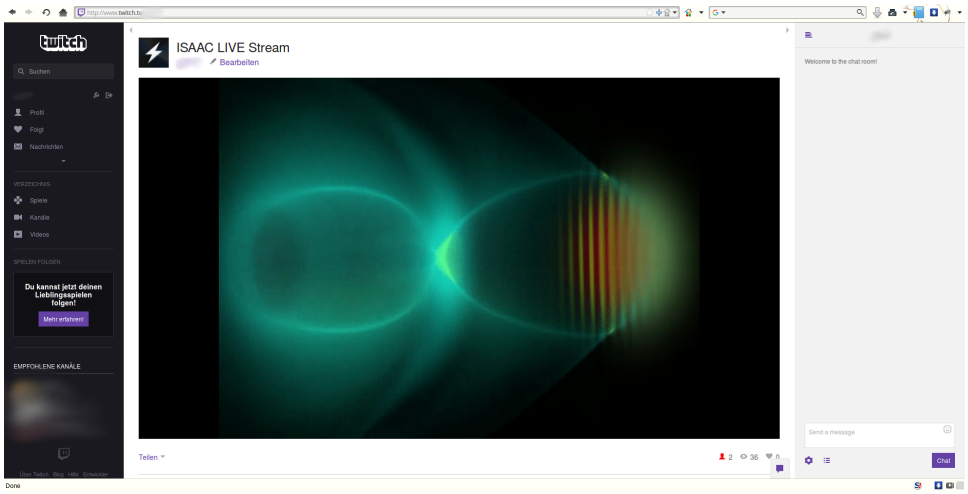

Abbildung 5.7: Direktes Übertragen der Livevisualisierung auf die Videostreamplattform Twitch.

Für Debugingzwecke wurde des Weiteren ein einfacher `ImageConnector` auf Basis von SDL [LSG16] implementiert. Diese Klasse zeigt in einem Fenster direkt auf dem Rechner, auf dem auch der ISAAC Server läuft, die älteste noch laufende Simulation an.

Um die Mächtigkeit und die Vielfältigkeit der abstrakten Klasse zu zeigen wurde zu guter Letzt noch eine Verbindung zu der Videostreamingplattform Twitch [Twi16] implementiert. Dazu konnte abermals gStreamer benutzt werden. Die gStreamer-Pipeline von dem Bitmap in ISAAC über die Videokodierung hin zu einem Netzwerkstream musste dabei nur dahingehend angepasst werden, dass statt eines RTP-Streams zu dem Client ein RTMP-Stream zu dem Twitchserver erstellt wird. Auf diese Art und Weise kann man viele Menschen an einer Livevisualisierung teilhaben lassen, ohne dass sie selbst direkt auf ISAAC zugreifen müssen können. Abbildung 5.7 zeigt solch einen Twitch Livestream.

5.2.4 Der Broker

Kern des ISAAC Servers ist eine Schleife, die zentral Nachrichten von Simulationen empfängt und an die beiden gerade beschriebenen Klassenarten weiterleitet, genannt Broker. Genauso wie es Extrathreads für jede Art von Streaming- und Metadatenklasse gibt, erfolgt auch die Kommunikation mit den Simulationen nebenläufig. Da hierfür jedoch kein erweiterbares Pluginsystem beschrieben wurde, zählt dieser *Message Thread* als Bestandteil des Brokers (siehe auch Abbildung

5.1). Dabei bekommen nicht alle Clients alle Nachrichten der Simulationen zugeschickt. Verbindet sich ein Client mit dem Server, erhält er nur eine Liste aller Simulationen mit rudimentären Informationen. Ein Client muss explizit angeben einer Simulation zu folgen, ehe er periodische Metadaten erhält und ein Videostream erstellt wird. Dazu sendet er eine JSON Nachricht mit dem Typen “observe” und der eindeutigen ID der Simulation. Gleichzeitig wählt er dabei auch aus, welche Art von Stream erzeugt werden soll, um Ressourcen auf dem Server zu sparen.

Es ist auch (die namensgebende) Aufgabe des Brokers eingehende Nachrichten an die richtige Simulation weiterzuleiten und die ankommenenden JPEG Bilder im Base64 Format zu dekodieren und an die ImageConnectors weiterzuleiten. Dabei wird ein Bild nur einmal im Speicher gehalten und über einen mutexgesicherten Referenzzähler bestimmt, wann das Bild von keinem ImageConnector mehr genutzt wird und der Speicher freigegeben werden kann.

5.3 Referenzclient

Der Client ist die wohl heterogenste aller ISAAC Komponenten. Rein technisch gesehen, ist er nicht Teil von ISAAC. Nichtsdestotrotz stellt ISAAC einen Referenzclient bereit, der mithilfe von HTML5 auf allen gängigen und modernen Betriebssystemen und Browsern lauffähig ist. Der Client ist in der Lage, alle bisher implementierten Funktionen des Servers und vor allem der In-Situ Visualisierung anzusprechen und zu nutzen. In Abbildung 5.8 ist der gesamte HTML Client zu sehen.

Alle im Abschnitt 5.1.5 beschriebenen Einstellungen haben ihre Entsprechung auf der Clientoberfläche. Oben kann die Serveradresse und der Port angegeben werden. Mit einem Klick auf den rechtsstehenden Button verbindet sich die Webseite mit dem ISAAC-Server und empfängt eine Liste aller laufenden Simulationen, die ganz unten zu sehen ist. Diese Liste enthält schon wenige, wichtige Informationen über die Simulationen. Für alle Informationen und vor allem den Empfang von Metadaten und eines Videostreams muss eine Simulation jedoch mit einem Klick auf “Observe” abonniert werden.

Erst dann entfalten sich die auf dem Screenshot zu sehenden weiteren Informationen, allen voran die simulationsspezifischen Metadaten, wie z. B. der momentane Zeitschritt, und vor allem der Livevideostream der laufenden Simulation. In den zu sehenden Rahmen zeichnet entweder das VLC Plugin den RTP Stream oder das direkt über JSON übertragene Base64 kodierte JPEG wird in ein HTML5 Canvas gezeichnet. Egal welche der beiden Streamingarten genutzt wird, besteht die

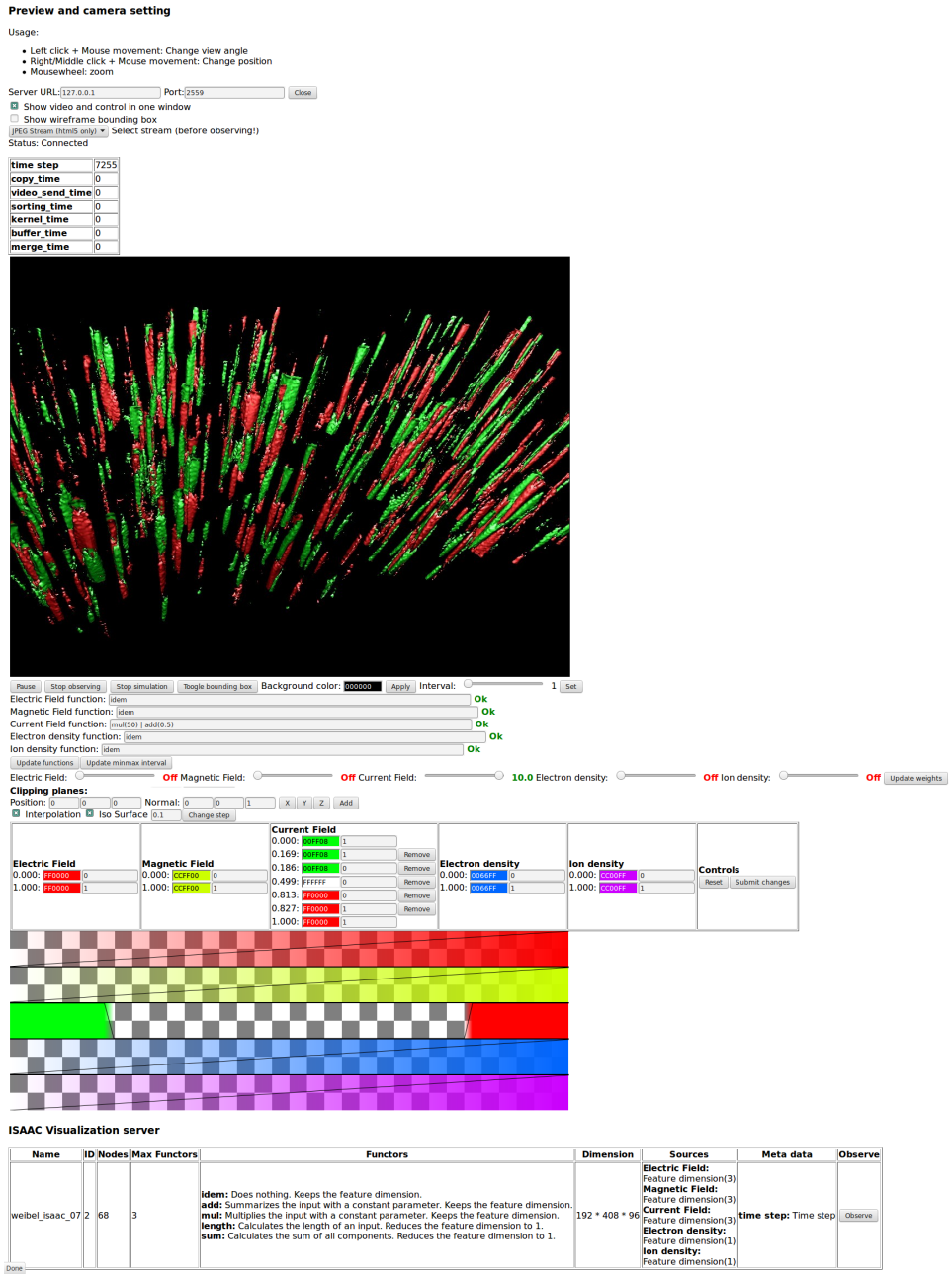

Abbildung 5.8: Komplette Ansicht des Clients mit einer laufenden PIConGPU-Simulation auf allen 68 NVidia K20 GPUs des Hypnos Clusters am HZDR. Zu sehen ist eine Simulation einer Weibel Instabilität. Nur die induzierte Stromstärke ist als Quelle aktiviert.

Möglichkeit die Boundingbox des gesamten Volumens zu zeichnen. Eine gewisse Latenz zwischen Steuerung und Bild lässt sich nicht vermeiden (mehr dazu in der Evaluation), deshalb wird die Boundingbox lokal gerendert und zeigt an, in welchen Bereich sich die Visualisierung bewegen wird. Die Steuerung von Rotation und Kameraposition erfolgt mithilfe der Maus.

Unter dem Livestream erfolgt die Einstellung der Functor-Chains, der Gewichte, der Clippingebenen und der Transferfunktionen. Die schon erwähnten Stützpunkte der Transferfunktionen können durch Klicks auf die Farbverläufe hinzugefügt werden und dann Alphawert und Farbe editiert. Für die Auswahl der Farbe nutzt dieser Referenzclient den Farbauswähler jscolor [Eas16]. Wird eine ungültige Functor-Chain angegeben, teilt einem der Client dies rechts neben dem Eingabefeld mit. Weiterhin wird rechts davon der minimale und maximale Wert einer Quelle angegeben, wenn “Update minmax Interval” ausgewählt wurde.

Generell berechnet der Client sehr wenig selbst. Bis auf die Bounding Box Position und Rotation werden alle Befehle direkt an den Server bzw. Visualisierungsmaster geschickt, welcher die geänderten Werte dann an alle beobachtenden Clients zurückschickt. Auf diese Art und Weise sind auch alle Einstellungen bei allen beobachtenden Clients synchron. Auch muss der Client die Transferfunktion nicht selbst aus den Stützpunkten interpolieren – insbesondere wenn in einer zukünftigen Version u. U. keine lineare Interpolation, sondern Kurven genutzt werden sollten.

Das Design des Clients wurde mit Absicht sehr spartanisch gehalten. Zum einen sprengt es den Rahmen dieser Arbeit eine optisch ansprechende Darstellung zu implementieren, die gleichzeitig benutzerfreundlich und intuitiv zu benutzen ist. Zum anderen bietet dieser sehr einfach gehaltene Client so die einfache Möglichkeit ihn simulations- oder gerätespezifisch zu optimieren. So wäre eine tablet- oder smartphoneoptimierte Version mit eingeschränkter Funktionalität und Steuerung durch Touchgesten denkbar. Auch möglich wäre eine Version, die auf Steuerelemente verzichtet und nur zur Anzeige von Stream- und Metadaten genutzt werden kann. Dafür müsste nicht einmal der HTML oder Javascript Code angepasst werden. Durch HTML5 können solche Designänderungen, wie die Reihenfolge bzw. das Anzeigen von Elementen, alleine durch CSS eingestellt werden.

6 Ergebnisse und Diskussion

ISAAC lässt sich nach mehreren Kriterien evaluieren. Zuallererst interessiert dabei die Geschwindigkeit, mit der Bilder erzeugt werden und hierbei nicht nur die Abhängigkeit von der Auflösung des Bildes, der Größe des Volumens und der Anzahl der verwendeten GPUs, sondern auch welche Bestandteile der ISAAC Renderchain vom Datum im Simulationsvolumen hin zu einem Bild auf der Workstation außerhalb des Rechenzentrums wie stark in die Zeit mit einfließen – auch wieder mit denselben schon erwähnten Abhängigkeiten. Insbesondere muss evaluiert werden, ob etwaige Engpässe mit der gewählten Methodik zusammenhängen und wie sich diese Probleme in Zukunft vermeiden ließen.

Weiterhin muss diskutiert werden, ob der Ansatz wie er gezeigt wurde, in der Lage wäre, auf Exascalesystemen realisiert zu werden. Es soll aus den Daten also Laufzeitverhalten von bisher nicht existenten Systemen approximiert und bewertet werden. Wichtig ist dabei auch, welche Nachteile die templatisierte In-Situ Methodik mit sich bringt und welche Lösungsstrategien es dafür in der Zukunft gibt.

All dies ist aber irrelevant, wenn die erzeugten Visualisierungen keinen Erkenntnisgewinn versprechen. Deshalb muss zu guter Letzt die Korrektheit und der Nutzen der erzeugten Streams und auch der Kommunikationsmöglichkeiten diskutiert werden. Insbesondere hierfür wird die exemplarische Implementierung in PIConGPU herangezogen und probiert zu zeigen, wie die erzeugten Bilder die vorgestellten physikalischen Phänomene aus Abschnitt 3.1.2 helfen zu verstehen und welche neuen, wissenschaftlichen Möglichkeiten ISAAC für diese Plasmasimulation und wissenschaftliche Simulationen im Allgemeinen bietet.

6.1 Geschwindigkeit

Um die Geschwindigkeit der ISAAC Bibliothek einschätzen zu können, wurde die schon beschriebene Beispielanwendung um Zeitausgaben und eine automatische Steuerung einer Testszene ergänzt. In insgesamt 1080 Zeitschritten wird das Volumen um je 1°-Schritte um die X-, dann die Y- und zu guter Letzt die Z-Achse gedreht. Der Abstand zur Kamera wird so gewählt, dass bei einer achsenparallelen Draufsicht zwar das ganze Volumen zu sehen, aber trotzdem möglichst viel

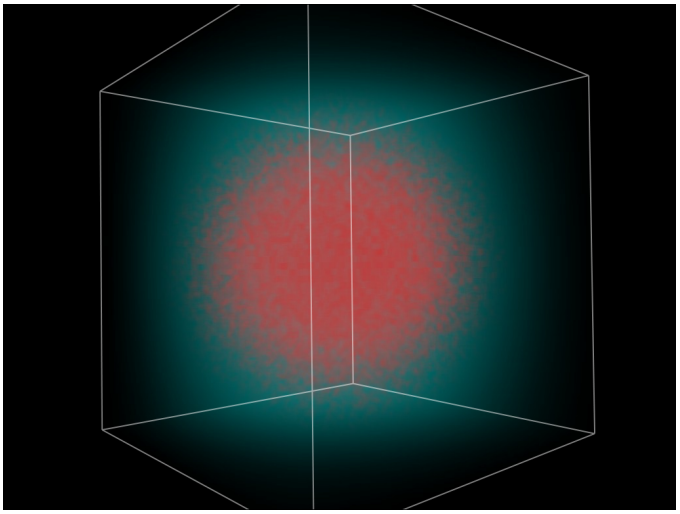

Abbildung 6.1: Testszene für Laufzeitmessungen. Es wurde probiert eine typische Szeneneinstellung zu wählen, in der die gesamte Simulation beobachtet wird.

Strategie	Direct Send	Sequential				Reduce			
		Auto	BSwap	Radixk	Tree	Auto	BSwap	Radixk	Tree
Zeit in ms	31,1	5,4	5,3	5,9	29,0	7,1	6,9	7,1	31,0

Table 6.1: Vergleich der durchschnittlichen Compositezeit der ISAAC Visualisierung mit verschiedenen IceT Strategien, aber konstanter Knotenanzahl von 64 GPUs mit je einem 128^3 Volumen bei einer Auflösung von 1024 x 768 auf dem Rechencluster Hypnos des HZDR. Sequentielles Binary Swap scheint die optimale Strategie für ISAAC darzustellen.

Bild gefüllt ist. Bei einer Rotation liegen jedoch Teile des Volumens außerhalb des sichtbaren Bereichs, wie in Abbildung 6.1 gezeigt. Dieses Szenario wird als realitätsnah angenommen. Die 1080 Zeitschritte werden zweimal hintereinander ausgeführt und nur der zweite Durchlauf ausgewertet, damit Initialisierungseffekte die Messung nicht beeinflussen. Die Messungen werden auf dem Hypnos Cluster des HZDR ausgeführt. Der für die Arbeit genutzte Teil des Clusters besteht aus 16 Rechnern mit je einem Intel Quad-Core Xeon 2,4 GHz, der dank Hyperthreading 8 Hardwarethreads besitzt, sowieso je vier Nvidia K20m Grafikkarten mit GK110 Chips mit je 2496 Streamprozessoren und 5120 MB Grafikkartenspeicher.

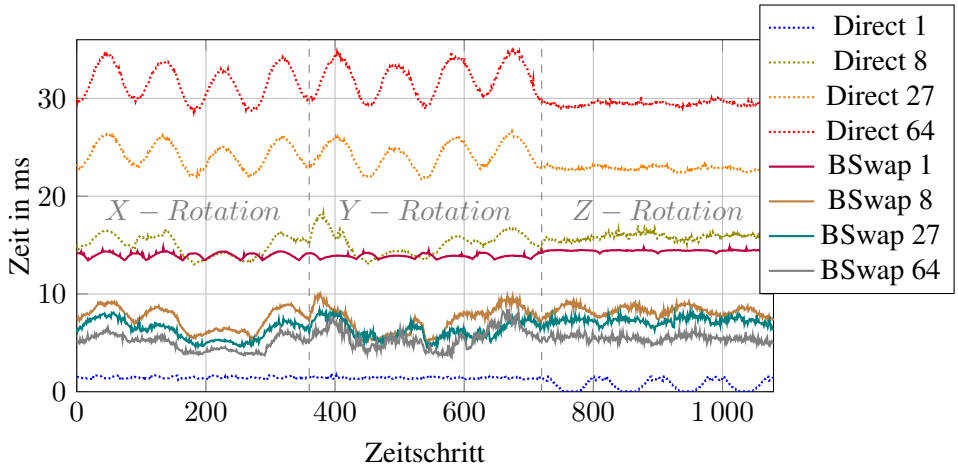

Abbildung 6.2: Vergleich der Compositezeit IceTs in Abhängigkeit der Strategien Direct Send und Binary Swap für 1, 8, 27 und 64 Rechenknoten/GPUs. Direct Send wird mit steigender Knotenanzahl langsamer, Binary Swap schneller.

6.1.1 Optimale IceT Strategie

Bevor die eigentlichen Laufzeitmessungen an ISAAC beginnen können, wird zuerst evaluiert, welche IceT-Strategie für einen typischen Simulationsaufbau auf vielen GPUs die beste Laufzeit bietet. Tabelle 6.1 zeigt die durchschnittlichen Compositezeiten für die Erzeugung eines Bildes mit 64 GPUs mit je einem Volumen von 128^3 bei einer Auflösung von 1024x768. Es zeigt sich, dass Direct Send wie erwartet eine schlechte Performance bietet. Des Weiteren sind alle Reduce-Strategien stets etwas (ca. 2 ms) schlechter als ihre sequentiellen Gegenstücke und der Tree-Ansatz in jedem Fall sogar langsamer als Direct Send. Da Binary Swap stets etwas besser als alle anderen Strategien war, wurde es für alle weiteren Messungen genutzt.

Nichtsdestotrotz kann auch die Direct Send Strategie schneller als Binary Swap sein. Abbildung 6.2 zeigt die Compositezeit pro Zeitschritt für Knotenanzahlen, mit denen sich würfelförmige Gesamtvolumen erzeugen ließen, bei einer Auflösung von 1024×768. Es zeigt sich zwar, dass Direct Send für eine sehr hohe Anzahl an Rechenknoten sehr viel schlechter skaliert als Binary Swap, trotzdem ist Direct Send bei nur einem Knoten schneller als Binary Swap. Die niedrige Netzwerkbelastung spielt hier keine Rolle und der Mehraufwand der Binary Swap Methode kann seine Stärken nicht ausspielen. Teilweise bricht die Laufzeit von Direct Send sogar auf 37 µs ein. Aber schon ab 8 MPI-Knoten übertrumpft der Binary Swap den Direct Send Algorithmus, obwohl immer noch wenig Netzwerkkommunikati-

on notwendig ist. Alleine die Verteilung des Compositings auf 8 CPUs reicht schon für eine bessere Laufzeit aus0.

Allgemein gilt, dass Direct Send für immer mehr Knoten immer langsamer wird, also sogar negativ skaliert. Binary Swap hingegen wird mit zunehmender Knotenanzahl immer etwas schneller und scheint sich einem von der Auflösung abhängigen Optimum anzunähern, was bei diesem Beispiel bei ungefähr $5\,\mathrm{ms}$ liegt. In jedem Fall zeigen die Messreihen eindeutig, dass Binary Swap für die Einsatzgebiete von ISAAC gegenüber Direct Send deutlich die bessere Wahl darstellt, insbesondere wenn man an hochparallele Peta- und Exascalesysteme denkt.

6.1.2 Kernelgeschwindigkeit

Die Geschwindigkeit der Kernelausführung hängt von vielen verschiedenen Parametern ab, die hier näher beleuchtet werden sollen. Für die folgenden Laufzeitmessungen wurde stets dasselbe Programm und dieselbe Messabfolge, die schon vorgestellt wurde, übernommen. Insofern eine Messung einen bestimmten Parameter nicht gesondert überprüft hat, gelten diese **Standardmessparameter**:

- Das lokale Volumen hat eine Größe von $128 \times 128 \times 128$.
- Der parallele Algorithmus läuft auf 64 GPUs in einer $4 \times 4 \times 4$ Aufteilung. Das globale Volumen hat also eine Gesamtausbreitung von $512 \times 512 \times 512$.
- Die Auflösung ist auf 1024×768 gesetzt. Im Durchschnitt werden also ungefähr 768×768 Pixel berechnet, da das Gesamtbild bei Draufsicht quadratisch ist.
- Die Functor-Chains sind alle `idem`. In ISAAC erfolgt immer ein Aufruf einer Chain, diese liefert in diesem Fall aber stets die X-Komponente des Eingabewerts zurück.
- Beide Testquellen sind aktiviert, da in der Praxis oft zwei Quellen und deren Korrelation zueinander betrachtet werden sollen. Das Gewicht beträgt 1.
- Die Schrittweite des Raycastingalgorithmus' beträgt $0{,}5$, um nach dem Nyquist-Shannon-Abtasttheorem keine Daten zu überspringen.
- Zur Visualisierung wird die CUDA Variante von ISAAC genutzt.
- Es erfolgt Raycasting nach dem Emissions/Absorptions-Modell und keine Interpolation.
- Die IceT-Strategie ist der sequentielle Binary Swap Algorithmus.

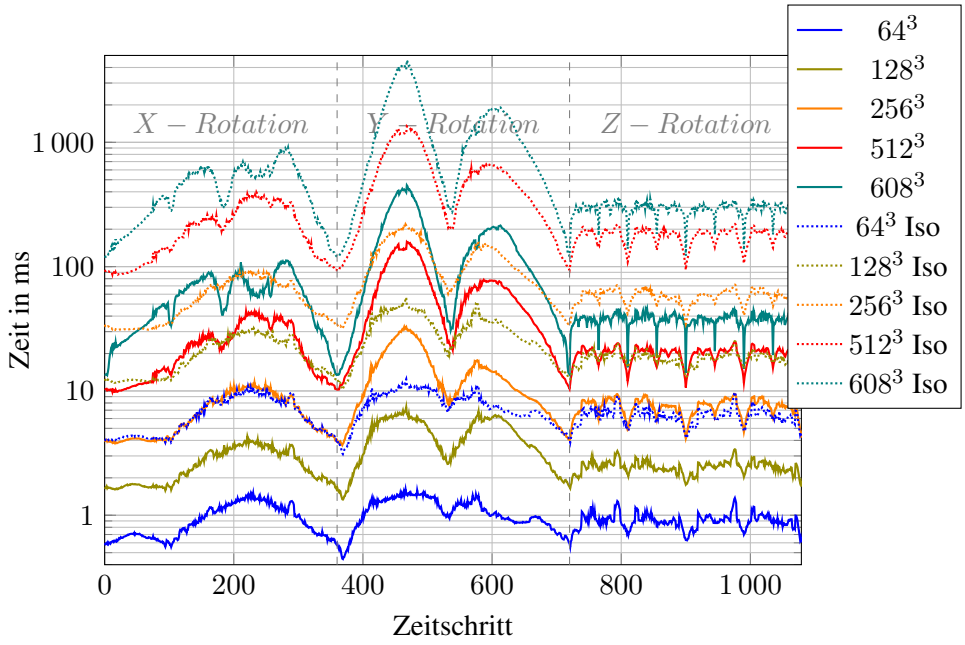

Abbildung 6.3: Kernelgeschwindigkeit pro Zeitschritt für verschiedene Volumengrößen und Renderparameter. Die restlichen Einstellungen sind wie in Abschnitt 6.1.2 beschrieben. Neben der Volumengröße hat auch der Blickwinkel einen starken Einfluss auf die Laufzeit.

Skalierung mit Volumengröße Simulationen, die für Peta- oder gar Exascalesyteme in Frage kommen, müssen skalierbar sein. Man unterscheidet zwischen *Weak* und *Strong Scaling*. Beim Weak Scaling kann ein größeres Problem in derselben Zeit durch das Hinzufügen neuer Recheneinheiten gelöst werden. Beim Strong Scaling kann dasselbe Problem in kleinerer Zeit gelöst werden. Bezogen auf ISAAC bliebe das lokale Volumen im ersten Fall gleich, im zweiten Fall würde es pro Knoten kleiner werden, aber das globale wäre konstant. Zuerst soll betrachtet werden, wie ISAAC mit schwach skalierenden Simulationen zusammenarbeitet.

In Abbildung 6.3 sieht man die Ausführungszeit des Raycasting Kernels für verschiedene Volumengrößen mit einer Kantenlänge von 64 bis hin zu 608, womit das Volumen fast den gesamten Speicher der K20m einnimmt. Des Weiteren wurde einmal nach dem Emissions/Absorptions-Modell ohne Interpolation zwischen den Volumenwerten und einmal mit Interpolation und Isoflächenrendering gemessen. Es zeigt sich, dass bei einer Verdopplung der Kantenlänge, also einer Verachtfachung des Volumens, die Laufzeit ungefähr dreimal größer wird. Bei einem Volumen von 64^3 braucht der einfache Kernel im Durchschnitt nur 1 ms, wohingegen er bei 128^3 im Durchschnitt schon 3,1 ms braucht, bei 256^3 9,2 ms usw. . Für PIConGPU sind

Werte von 64 bis 256 Normalität, für die in der Messung sehr schnelle Kernellaufzeiten erreicht werden konnten. Es zeigt sich auch, dass die Interpolation ungefähr siebenmal mehr Laufzeit benötigt gegenüber einem Rendering mit Nearest Neighbour Zugriff. Das folgt daher, da bei dieser Zugriffsart für einen Datenpunkt im Volumen, die acht umgebenden Datenpunkte gelesen und trilinear interpoliert werden müssen.

Gerade die CUDA-Variante könnte eigentlich von einem Zugriff auf die Simulationsdaten mithilfe von Texturspeicher profitieren. Dieser für Computergrafik optimierte Speicher von GPUs bietet eine Interpolation direkt beim Zugriff und würde die Laufzeit gegenüber manueller Interpolation deutlich reduzieren. Texturspeicher hat jedoch leider den Nachteil, dass er nur auf 1, 2 oder 4-elementigen Daten arbeiten kann. In wissenschaftlichen Simulationen sind jedoch Vektorfelder mit drei oder mehr als vier Dimensionen pro Datum sehr häufig, weshalb diese elementweise umkopiert und konvertiert werden müssten, was dem simulationsnahen In-Situ Ansatz ISAACs widerspricht. Des Weiteren greift ISAAC über den `[]` Subscriptoperator auf Daten zu und hat somit keinerlei Wissen über die wahre Struktur des zu Grunde liegenden Speichers. Eine Simulation könnte z. B. ein Vektorfeld mit 6 Dimensionen pro Vektor beschreiben, wovon die ersten 3 die gerichtete Geschwindigkeit und die letzten 3 einer wirkenden Kraft entsprechen. Für ISAAC ließe sich dieses Vektorfeld über zwei Quellen mit je 3 Merkmalsdimensionen beschreiben. ISAAC sieht also zwei Speicherbereiche statt dem einen realen. Auf dieser Basis kann ohne Kopieroperation kein Texturzugriff erfolgen. Weiterhin lässt sich Texturspeicher mit der abstrakten ALPAKA-Bibliothek nicht realisieren, weil er auf anderen Geräten wie Intel Xeon Phis nicht verfügbar ist.

In Abbildung 6.4 sind die Unterschiede der Darstellungsarten für PIConGPU zu sehen. Die Abbildung zeigt viermal dieselben Daten mit derselben Kamera, aber links mit einer Darstellung als leuchtendes Gas, rechts mit Isoflächendarstellung und für beide unten mit Interpolation. Es fällt auf, dass sich die linken Darstellungen kaum unterscheiden, wohingegen man bei der Isoflächendarstellung deutlich sieht, ob mit oder ohne Interpolation gezeichnet wurde. Nichtsdestotrotz sind die Darstellungen ähnlich, und es wäre denkbar im wissenschaftlichen Auswertungsworkflow bei interessanten Stellen Interpolation für ein besseres Bild zuzuschalten, aber sonst die schnellere Darstellung zu wählen. Insbesondere für Simulationen, deren Volumen sehr groß sind und für die die Kernelausführungszeit von ISAAC bei Isoflächendarstellung für ein Bild in die Sekunden gehen kann (siehe Abbildung 6.1.2), stellt dies eine gangbare Alternative dar – abgesehen von der Erhöhung der Abtastperiode bzw. einer Reskalierung des Volumens vor der Visualisierung durch den `[]` Subscriptoperator.

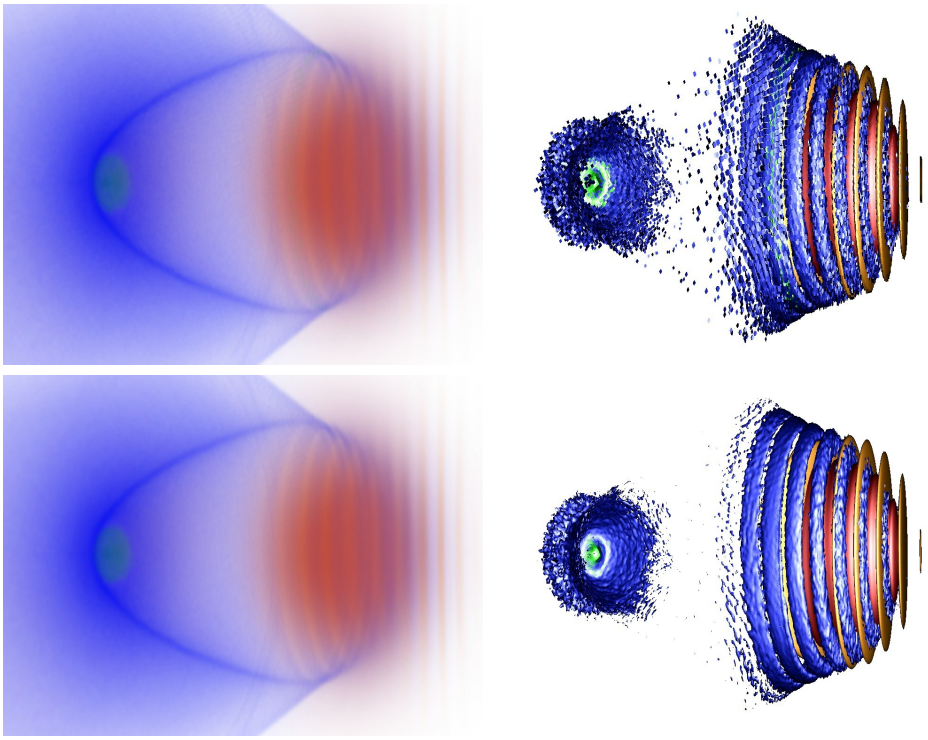

Abbildung 6.4: Darstellung der Laser-Kielfeld-Beschleuniger-Simulation in PIConGPU auf 32 GPUs mit je einem lokalen Volumen von $96 \times 128 \times 96$. Die linken Darstellungen zeigen das Emissions/Absorptions-Modell, die rechten Isoflächendarstellung, jeweils oben ohne und unten mit aktivierter Interpolation. Gerade die Isoflächendarstellung profitiert stark von aktivierter Interpolation.

In den Messwerten in Abbildung 6.3 fällt weiterhin auf, dass unabhängig von dem gewählten Renderer bzw. der Volumengröße immer ähnliche Laufzeitsprünge in Abhängigkeit des Zeitschrittes entstehen. Dieser Zeitschritt korreliert wiederum mit einer bestimmten Rotation. Rotationstechnisch runde Zeitschritte, die einer Drehung um $0°$, $90°$, $180°$ und $270°$ entsprechen, besitzen dabei oft eine zu ihren Nachbarn besonders gute Ausführungszeit, weil besonders viele Werte direkt aus einer Cacheline gelesen werden können bzw. aufeinanderfolgende Kernel aufeinanderfolgenden Speicher verwenden (Contiguous Memory Access), worauf Nvidia GPUs optimiert sind. Bei den ersten 360 Zeitschritten, also der Rotation um die X-Achse, scheint der schlechteste Fall bei einer Rotation um $225°$ und $270°$ einzutreten. Noch extremer ist dieser Effekt bei der darauf folgenden Drehung um die Y-Achse. Je größer das Volumen wird, desto schlechter wird die Ausführungszeit bei $90°$ und $270°$ verglichen mit der zu $0°$ und $180°$. Die Vermutung liegt nahe, dass bei kleinen Volumen, wie 64^3, zwar quer zu den Cachelines zugegriffen wird, aufgrund der geringen Größe aber trotzdem folgende Abtastschritte weniger Cachemisses als bei großen Volumen haben. Bei der letzten Rotation um die Z-Achse ändern sich die Laufzeiten winkelabhängig kaum. In den Blocks von 8×16 Kerneln werden unabhängig von der konkreten Rotation immer Gruppen von Kerneln auf einer Cacheline liegen und es nur wenig Cachemisses geben, da sich der gesamte Block parallel zu der XY-Ebene durch das Volumen bewegt. Jedoch zeigt sich auch hier die besonders effiziente Abarbeitung bei den Vielfachen von $90°$, wahrscheinlich durch Contiguous Memory Accesses.

Die konkrete Kernelgeschwindigkeit ist also stark abhängig von dem Blickpunkt und damit der Richtung des Raycastings. Je größer die Volumen sind, desto stärker ist der Effekt. Bei 64^3 liegt zwischen der kleinsten und der größten Laufzeit ein Faktor von $3{,}6$, wohingegen er bei 608^3 bei $33{,}6$ liegt. Um die WissenschaftlerInnen bei der Wahl eines gutes Blickwinkels zu unterstützen, wäre es denkbar in der Zukunft eine Möglichkeit zu schaffen die Rotation in die oft schnelleren Vielfachen von $45°$ oder $90°$ einrasten zu lassen. Gleichzeitig bestünde so die Möglichkeit verschiedene Simulationen einfach parallel aus demselben Blickwinkel ansehen zu können.

Einfluss der Auflösung Abbildung 6.5 zeigt die Gesamtzeichenzeit von ISAAC einschließlich der Kernelrenderzeit, dem IceT-Compositing und den notwendigen Voroperationen wie der Sortierung der lokalen Volumen in Abhängigkeit von der gewählten Auflösung. Wie erwartet, zeigt sich eine Akkumulation der IceT-Composite- und der Kernelrenderzeiten. Weiterhin fällt auf, dass die Ausführungszeit mit höherer Auflösung größer wird. In Tabelle 6.2 zeigt sich jedoch, dass die Laufzeit pro Pixel mit steigender Auflösung sinkt. Selbst wenn man nur die

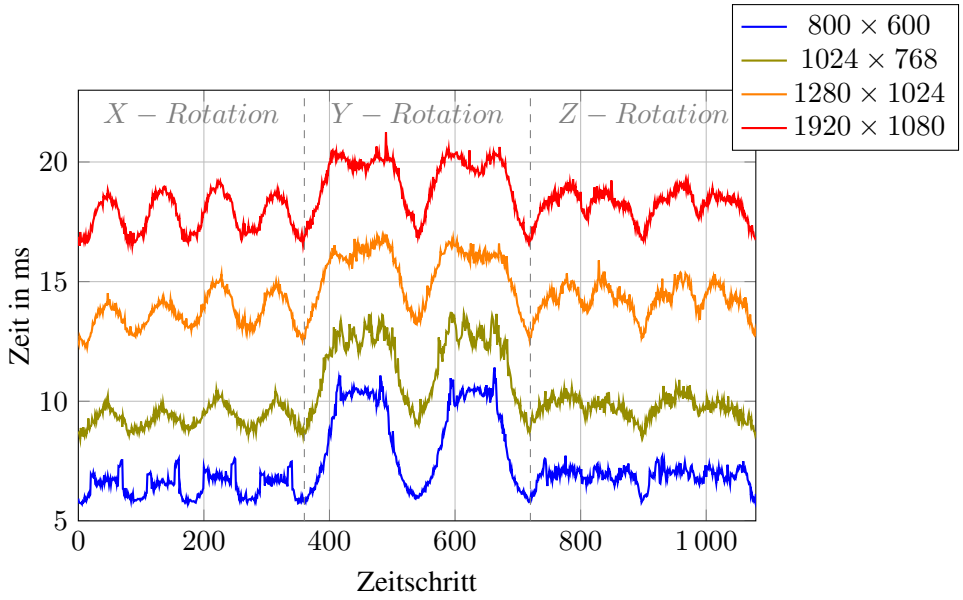

Abbildung 6.5: Gesamtrenderzeit für verschiedene, typische Auflösungen. Die restlichen Parameter sind wie in Abschnitt 6.1.2 beschrieben. Die Laufzeit steigt mit steigender Auflösung, bleibt aber auch bei Full HD echtzeitfähig.

Auflösung	800 × 600	1024 × 768	1280 × 1024	1920 × 1080
Anzahl an Pixeln	480 000	786 432	1 310 720	2 073 600
Durchschnittliche Laufzeit	7,32 ms	10,24 ms	14,46 ms	18,42 ms
Laufzeit pro Pixel	15,3 ns	13,0 ns	11,0 ns	8,9 ns
Wirklich benutzte Fläche	600 × 600	768 × 768	1024 × 1024	1080 × 1080
Anzahl an Pixeln	360 000	589 824	1 048 576	1 166 400
Laufzeit pro Pixel	20,3 ns	17,4 ns	13,4 ns	15,8 ns

Tabelle 6.2: Gesamtrenderzeit pro Bild und Gesamtrenderzeit pro Bild und Pixel für verschiedene, typische Auflösungen. Die restlichen Parameter sind wie in Abschnitt 6.1.2 beschrieben. Obwohl die Laufzeit mit steigender Auflösung mitsteigt, sinkt sie pro Pixel – meist auch, wenn man statt der Gesamtauflösung nur die wirklich gezeichnete Fläche betrachtet.

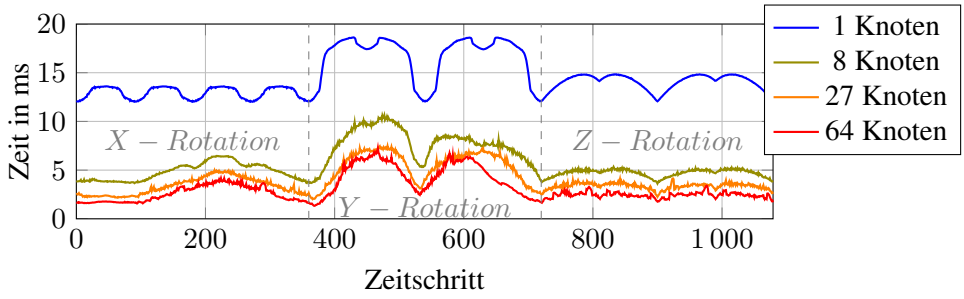

Abbildung 6.6: Laufzeitverhalten des Kernels in Abhängigkeit von Knotenanzahl und Rotation des Volumens. Die restlichen Parameter sind wie in Abschnitt 6.1.2 beschrieben. Da die zu rendernde Fläche mit steigender Knotenanzahl schrumpft, verringert sich auch die Laufzeit.

quadratische, tatsächlich genutzte Fläche betrachtet, zeigt sich eine Superskalarität bzgl. der Renderzeit pro Pixel. Nur der genutzte Full HD Ausschnitt 1080×1080 folgt nicht diesem Trend. Da fast die Hälfte des sichtbaren Bereichs ungenutzt ist, scheint sich dieser Bereich negativ auf die Gesamtlaufzeit des Renderers auszuwirken. Allgemein kann aber gezeigt werden, dass ISAAC auch für Full HD Auflösungen in Echtzeit Bilder generieren kann.

Einfluss der Knotenanzahl Es wurde schon gezeigt, dass sich IceT mit einer wachsenden Anzahl an Knoten bei gleichem Ausgabebild einer konstanten Laufzeit annähert. Nun soll betrachtet werden, wie die Kernelausführungsgeschwindigkeit mit wachsender Anzahl an Knoten skaliert. Dazu muss beachtet werden, dass mit wachsender Knotenanzahl die Größe der lokalen Volumen konstant bleibt, die zu zeichnende Fläche pro Knoten aber sinkt, da in der Messreihe stets das gesamte Volumen bildschirmfüllend dargestellt wird. Das bedeutet im Umkehrschluss aber auch, dass, wenn nur wenige oder gar nur ein Knoten involviert sind, es dem Fall entspricht, wenn in einer Visualisierung mit vielen Knoten nur ein Bereich groß dargestellt wird, weil auf diesen herangezoomt wurde.

Abbildung 6.6 zeigt dazu wieder die Ausführungszeit für 1, 8, 27 und 64 Knoten, sodass sowohl das lokale als auch das globale Volumen würfelförmig sind. Ein einzelner Knoten ist wie erwartet langsamer als mehrere Knoten, die denselben Bildbereich füllen. Für diesen Worst Case kann also von einer Kernelgeschwindigkeit von 18 ms ausgegangen werden. Die Effekte der Cache Misses sind für einen einzelnen Knoten relativ klein, der beste Wert liegt mit ungefähr 12 ms nur $1/3$ hinter dem Worst Case für einen Knoten. Für mehrere Knoten ist der Effekt der Cache Misses zwar wieder größer, trotzdem ist die Gesamtzeit stets deutlich kleiner als für

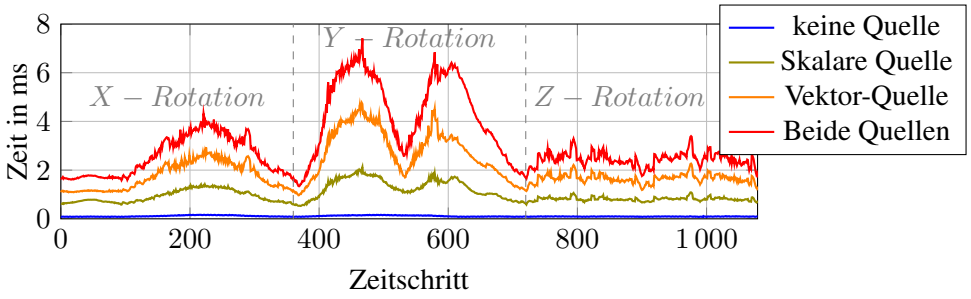

Abbildung 6.7: Laufzeitverhalten des Kernels in Abhängigkeit von der Anzahl der aktivierten Quellen und Rotation des Volumens. Die restlichen Parameter sind wie in Abschnitt 6.1.2 beschrieben. Die Laufzeit einer deaktivierten Quelle verschwindet vollständig.

einen Knoten. Da davon ausgegangen werden kann, dass die Simulation im Ganzen von Interesse ist, kann dieser Fall als Average Case angenommen werden. Bei 64 Knoten zeigt sich hier eine maximale Ausführungszeit von 7 ms.

Einfluss der Quellenanzahl ISAAC unterstützt beliebig viele Quellen. Jedoch wird es keinen Sinn ergeben, alle Quellen gleichzeitig zu beobachten; die WissenschaftlerInnen werden also für gewöhnlich eine Teilmenge der möglichen Volumendaten untersuchen wollen. Alle nicht untersuchten Quellen sollten dabei von dem Kernel nicht betrachtet werden. Um das zu erreichen, generiert ISAAC mithilfe von Templatisierung 2^n verschiedene Kernel – einen für jede mögliche Kombination von Quellen. In Abbildung 6.7 ist das Laufzeitverhalten des Kernels für verschiedene Kombinationen von aktivierten Quellen zu sehen. Wie erwartet braucht ein Kernel, der keine Quellen darstellt, sondern nur das Bild in der Hintergrundfarbe einfärbt, im Durchschnitt nur vernachlässigbare $0{,}1$ ms. Interessanter ist, dass die Geschwindigkeit des Kernels stark von der Dimensionalität des betrachteten Merkmals abhängt. Als Functor-Chain wurde `idem` gewählt, was bedeutet, dass stets nur die X-Komponente für das Rendering betrachtet wurde. Der einzige Unterschied liegt folglich darin, wie viele Daten pro Rasterschritt aus dem Volumen geladen werden können. Einerseits können hier wieder Caching-Effekte auftreten: Es fällt auf, dass die skalare Quelle weniger rotationsabhängige Laufzeitstörungen hat als die Vektorquelle. Bei der ersten liegt zwischen dem besten und dem schlechtesten Wert ungefähr Faktor 4, bei der zweiten Faktor 5. Andererseits scheint die skalare Quelle aber auch schlicht weniger Bandbreite als die Vektorquelle zu benötigen. Die Summe beider Quellen ergibt wiederum ungefähr das gleiche Verhalten wie die gleichzeitige Abtastung beider Quellen im Kernel.

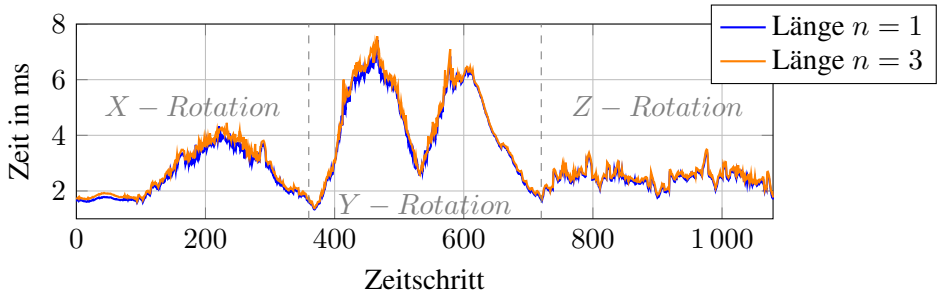

Abbildung 6.8: Laufzeitverhalten des Kernels in Abhängigkeit von der Länge der Functor-Chains. Die restlichen Parameter sind wie in Abschnitt 6.1.2 beschrieben. Da ISAAC stark Memory I/O Bound ist, ist der Unterschied hier marginal.

Damit zeigt sich, dass eine deaktivierte Quelle keine Performance des Renderers mehr verbraucht.

Einfluss der Functor-Chains Die Functor-Chains sind eine wichtige Hilfe für WissenschaftlerInnen, Daten zu analysieren bzw. für die Transferfunktion vorzubereiten. Um zu testen, wie stark diese Funktionalität die Laufzeit beeinflusst, wurde einmal die Zeit mit dem `idem` Functor gemessen und einmal mit der Functor-Chain `mul(1.1) | add(0.001) | length` für die Vektorquelle und `mul(1.1) | add(0.001) | mul(0.9)` für die Skalarquelle (da `length` hier nur den trivialen Absolutwert bestimmen würde). In Abbildung 6.8 zeigt sich, dass die Komplexität der Functor-Chains kaum Einfluss auf die Laufzeit hat, obwohl die erste Functor-Chain in `length` sogar eine relativ teure Wurzelberechnung beinhaltet. Damit zeigt sich auch sehr eindrucksvoll, dass ISAAC sehr Memory I/O Bound und nicht Compute Bound ist. Das bedeutet, dass ISAAC von einem schnelleren Rechenbeschleuniger oder mehr Speicher kaum profitieren würde, sehr jedoch von einem schnelleren Speicherzugriff.

Einfluss der Schrittweite Tabelle 6.3 zeigt die durchschnittliche minimale und maximale Laufzeit des Renderkernels für verschiedene Schrittweiten von $0{,}1$ bis $5{,}0$. Es fällt auf, dass der Kernel mit kürzeren Schrittweiten stets schneller wird. Es gibt sogar fast eine lineare Skalierung, die bei sehr großen Schrittweiten wie $5{,}0$ aber abschwächt, da die schrittweitenunabhängigen Teile des Kernels hier stärker ins Gewicht treten, die auch in dem Graphen 6.7 für keine aktivierten Quellen zu sehen waren, dort aber im Vergleich zu den anderen Messungen sehr klein. Des

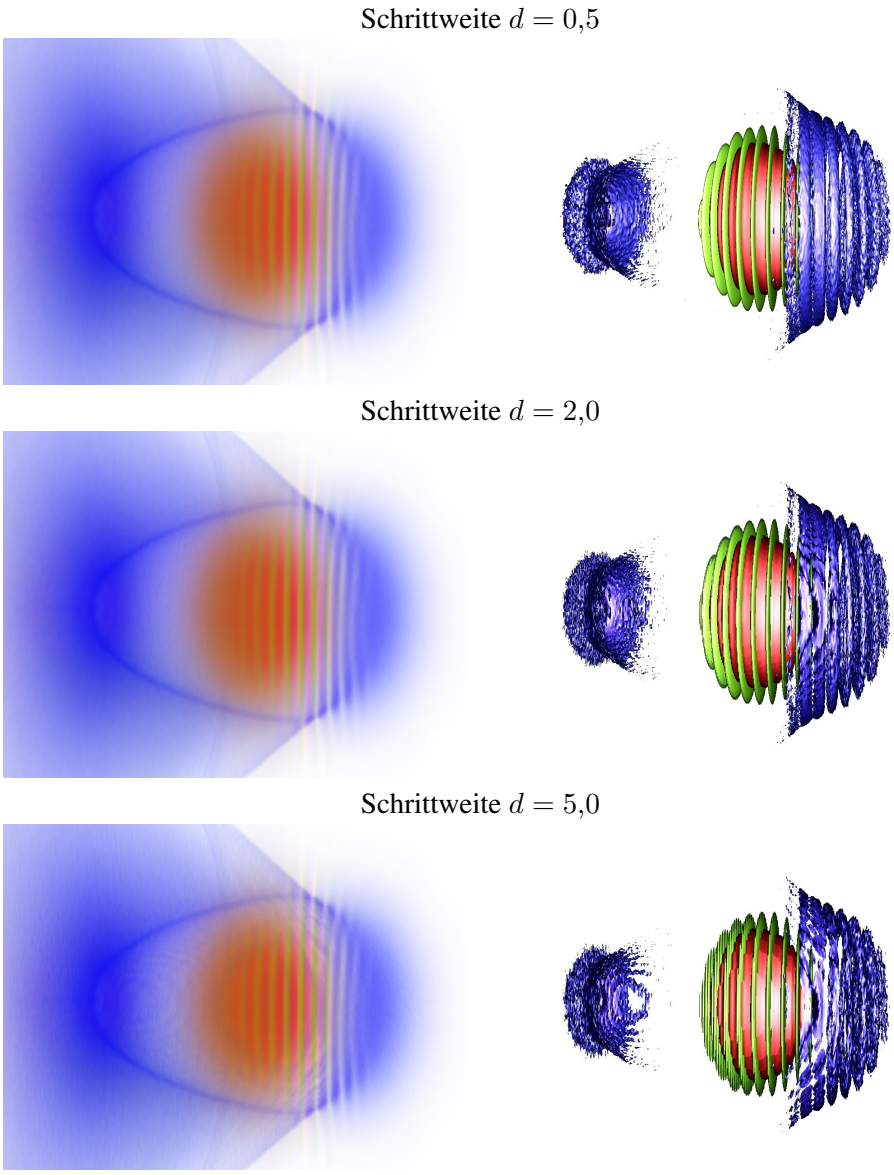

Abbildung 6.9: Einfluss von verschiedenen Schrittweiten auf die Qualität der erzeugten Visualisierung. Die nicht näher erläuterten Parameter sind wie in Abschnitt 6.1.2 beschrieben. Die Darstellung als leuchtendes Gas ist bis $d = 2$ noch ansprechend, bei der Isoflächendarstellung zeigen sich hier schon störende Fragmente.

Schrittweite	0,1	0,2	0,5	1,0	2,0	5,0
Durchschnittliche Kernellaufzeit in µs	14 116	7 214	3 104	1 724	937	395
Minimale Kernellaufzeit in µs	5 914	3 075	1 337	748	393	177
Maximale Kernellaufzeit in µs	36 491	17 927	7 409	3 597	1 894	860
Faktor zwischen minimaler und maximaler Kernellaufzeit	6,2	5,8	5,5	4,8	4,8	4,9

Tabelle 6.3: Durchschnittliche minimale und maximale Laufzeit des Kernels in Abhängigkeit von verschiedenen Schrittweiten. Die restlichen Parameter sind wie in Abschnitt 6.1.2 beschrieben. Es zeigt sich eine fast lineare Korrelation.

Weiteren verkleinert sich der relative Abstand zwischen der minimalen und maximalen Kernellaufzeit mit einer höheren Schrittweite.

Abbildung 6.9 zeigt den Einfluss der Schrittweite auf die Darstellung, in diesem Fall auf eine Laser-Kielfeld-Beschleuniger-Testszene aus PIConGPU. Für Abtastraten bis $d = 2{,}0$ sind keine wahrnehmbaren Unterschiede für das Emissions/Absorptions-Modell festzustellen gegenüber der nach dem Abtasttheorem optimalen Abtastrate von $d = 0{,}5$. Bei der Isoflächendarstellung zeigen sich jedoch fehlerhafte Kanten und die Beleuchtung bekommt eine Comic-ähnliche Fehldarstellung. Für $d = 5{,}0$ leidet die Isoflächendarstellung noch mehr und es sind deutliche, streifenförmige Fehldarstellungen zu erkennen. Aber auch in der Darstellung nach dem Emissions/Absorptions-Modell zeigen sich interferenzmusterähnliche Fehlmuster in der Darstellung.

Die Darstellungen und Fehler sind dabei sehr simulationsspezifisch. Dadurch, dass bei dieser Simulation Nachbarzellen ähnliche Werte besitzen, kann das Abtasttheorem für die Darstellung als leuchtendes Gas ignoriert und trotzdem noch ansehnliche Ergebnisse erzielt werden. Die Isoflächendarstellung könnte von dieser Charakteristik auch profitieren, indem bei einem Hit anstatt diesen Punkt zu nehmen, in höherer Abtastrate zwischen diesem und dem letzten Abtastpunkt nach einem früheren Hit gesucht wird, um die Oberfläche besser zu approximieren.

Einfluss des Rechenbeschleunigers Der Vergleich zwischen einem Rechenbeschleuniger wie Nvidia GPUs und einem Prozessor ist nur bedingt sinnvoll. Nichtsdestotrotz soll an dieser Stelle gezeigt werden, welche Beschleunigung ein Rechenbeschleuniger der Visualisierung bringt und diskutiert werden, inwiefern auch eine langsame Visualisierung auf einer CPU nützlich ist. Des Weiteren wird ein Vergleich zwischen der Ausführungszeit der nativen CUDA Implementierung und derjenigen mit ALPAKA angestellt. Bei diesem Vergleich geht es nicht darum

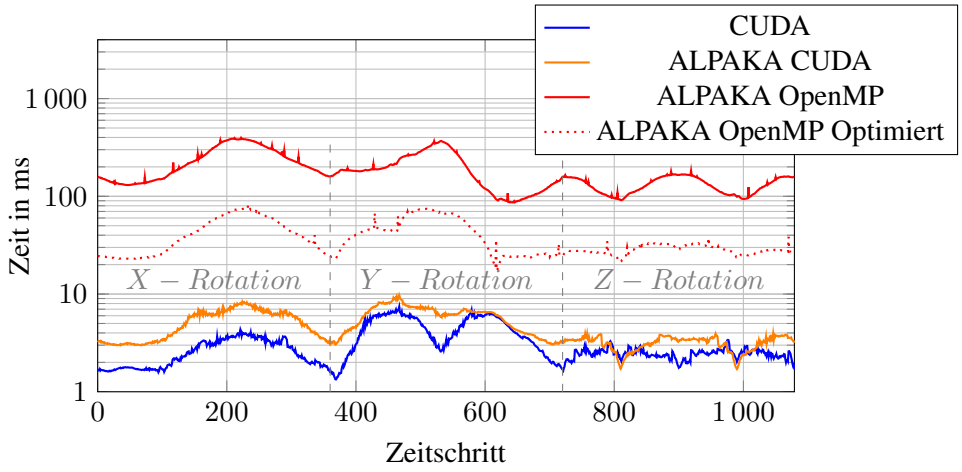

Abbildung 6.10: Laufzeitverhalten des CUDA und der beiden ALPAKA Kernel. Der optimierte, gestrichelt dargestellte OpenMP ALPAKA Kernel hat eine größere Schrittweite $d = 2{,}0$ und erzeugt ein kleineres Bild mit einer Auflösung von 800×600. Alle nicht näher spezifizierten Parameter sind wie in Abschnitt 6.1.2 beschrieben. Die CPU Variante ist wie erwartet langsamer, aber mit Qualitätseinbußen trotzdem bedingt echtzeitfähig.

zu zeigen, ob von ALPAKA erzeugter CUDA Code nativem unterlegen ist. Worpitz konnte schon zeigen, dass alle von ALPAKA abstrahierten Aspekte identischen CUDA Code erzeugen [Wor15]. Jedoch nutzt ISAAC Constant Memory, welchen ALPAKA noch nicht abstrahieren kann. Es kann also gezeigt werden, wie stark dieser Constant Memory die Laufzeit beeinflusst.

Abbildung 6.10 zeigt das Laufzeitverhalten der drei Kernel. Die CUDA Implementierung ist wie erwartet schneller als die von ALPAKA, dafür fallen die Caching Effekte nicht so stark ins Gewicht, weshalb die maximale Laufzeit kaum über der von CUDA liegt. Nichtsdestotrotz zeigt sich, dass es wichtig ist, dass auch ALPAKA in Zukunft Constant Memory unterstützen sollte. Die OpenMP Variante ist, wie zu erwarten war, Größenordnungen langsamer als beide CUDA Varianten. Nichtsdestotrotz liegt die Kernelgeschwindigkeit selbst im Worst Case unter einer halben Sekunde. Mithilfe der bereits angesprochenen Parameter ist es also auch möglich für Simulationen ohne Rechenbeschleuniger eine Echtzeitvisualisierung zu erzeugen. Alleine eine höhere Schrittweite von $d = 2{,}0$ und eine geringere Auflösung von 800×600 würde die Visualisierung schon fast um den Faktor 5 beschleunigen. Das Laufzeitverhalten dieser Optimierung zeigt die gestrichelte, rote Linie in der Grafik. Mit durchschnittlich 38 ms Laufzeit ist also auch in diesem Fall mit wenigen Einschränkungen eine Echtzeitvisualisierung möglich.

Auflösung	800×600	1024×768	1280×1024	1920×1080
Komprimierungs- und Sendezeit in ms	4,4	7,5	11,9	18,3

Tabelle 6.4: Durchschnittliche Komprimierungs- und Sendezeit vom Master zum ISAAC Server in Abhängigkeit von der Auflösung.

Bildübertragung und -kompression Für ein Full HD Bitmap müssten jeden Frame 8 MB vom Master zum ISAAC Server übertragen werden. Da nicht davon ausgegangen werden kann, dass zwischen beiden ein Hochgeschwindigkeitsnetzwerk wie Infiniband vorliegt, werden die Bilder deshalb vor dem Verschicken komprimiert. Des Weiteren muss der Binärdatenstrom durch Base64 Rekodierung in ein JSON kompatibles Format gebracht werden. Tabelle 6.4 zeigt die dafür benötigte Zeit. Da einerseits bei der Nutzung von Rechenbeschleunigern oft CPU Ressourcen brach liegen und andererseits keine mit Threading nicht kompatiblen MPI Aufrufe beim Komprimieren und Senden erfolgen, passiert die Kompression und Übertragung deshalb in einem Extrathread während die Simulation weiter läuft. Auf diese Art und Weise kann diese Laufzeit vollständig versteckt werden (*Latency Hiding*), insofern ein Zeitschritt der Simulation mehr Zeit braucht als das Verschicken der Daten. Für Full HD Bilder dürfte ein Zeitschritt also nicht schneller als $18{,}3\,\text{ms}$ sein. Aber selbst, wenn er es wäre, ergäben sich nach der Tabelle 6.1 für das Beispiel eine Gesamtrender- und sendezeit von $\sim 36{,}4\,\text{ms}$ und somit ungefähr 27 Bilder pro Sekunde.

6.1.3 Interaktionslatenz

Um ein Visualisierungsbild auf einer lokalen Workstation ansehen zu können, sind viele Computer und Netzwerke involviert. Das Bild wird verteilt auf den Compute Nodes generiert, auf dem Master zusammengesetzt, an den ISAAC Server geschickt, dort zu einem Stream umgewandelt und zu guter Letzt an den lokalen Client geschickt. Bei der reinen Beobachtung einer Visualisierung ohne Interaktion fällt die Latenz dabei nicht auf. Anders sieht es aus, wenn Simulations- oder Visualisierungsparameter geändert werden. Wie in dem Sequenzdiagramm in Abbildung 6.11 zu sehen ist, entstehen zusätzliche Latenzen von der Workstation zu der Simulation. Um die Gesamtlatenz abschätzen zu können, wurde der HTML Client um eine Funktion erweitert, die den momentanen Zeitpunkt beim Abschicken einer Änderung, z. B. der Rotation, ausgibt und außerdem, wenn ein neues Videobild im URI Format angekommen ist. Die Messungen wurden zwischen dem HZDR Rechencluster Hypnos, auf dem eine Testsimulation lief, und einem normalen Haus-

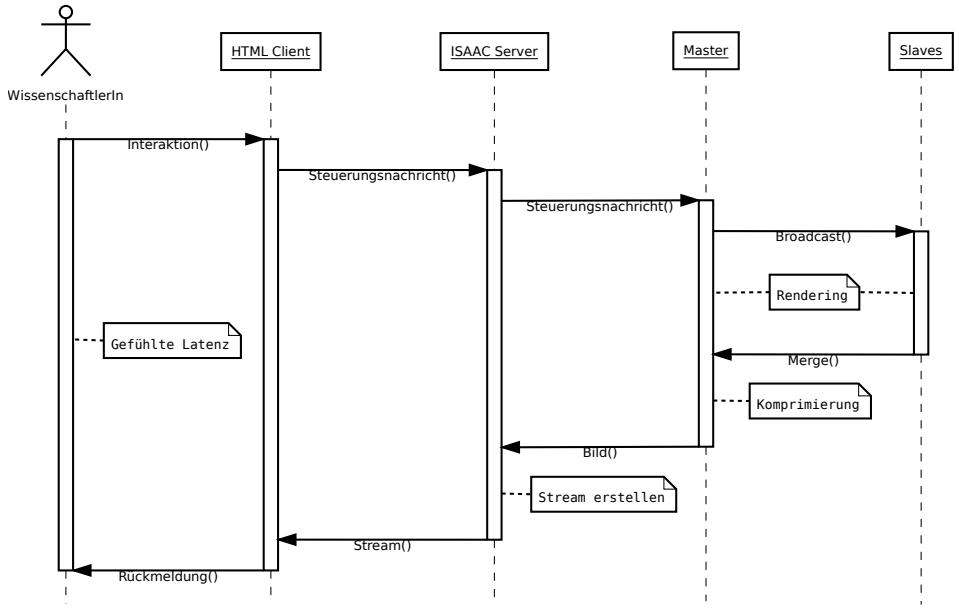

Abbildung 6.11: Zusammensetzung der gefühlten Latenz zwischen Interaktion und Rückmeldung der Simulation. Die Latenz ergibt sich in der Summe aus den vier verschiedenen Arten von Zwischenstationen von der Anfrage bis zum Bild.

haltsinternetanschluss in Dresden, durchgeführt. Bei 10 Messungen ergab sich eine Durchschnittslatenz von 175 ms, bei einem Minimal- und Maximalwert von 163 ms und 189 ms. Die durchschnittliche komplette Zeichenzeit, einschließlich Nachrichtenauswertung, Sortierung, Rendering, IceT Compositing und Bildkomprimierung, betrug rund 20 ms, nahm also nur einen kleinen Teil der Gesamtlatenz ein.

Würde statt der direkten Anzeige der URI Zeichenkette der RTP Stream mithilfe des VLC Browserplugins angezeigt werden, entstünden unabhängig von dem gewählten Codec nochmal starke Latenzen, die teilweise in die Sekunden gehen können. Dies hat sich gezeigt, als testweise direkt mit gst-launch, einem Programm aus dem gStreamer Paket, derselbe Stream wie mit VLC angezeigt wurde. Mit gst-launch ist die Latenz vergleichbar mit der des URI Streams, aber sehr stockend. Der Stream im VLC Player läuft flüssig, fügt aber nochmal wenige Sekunden Latenz hinzu. Andererseits reduzieren spezielle Videokomprimierungsalgorithmen wie H.264 die Netzwerklast gegenüber Einzelbildübertragungen wie den im JSON Format übertragenen URI Zeichenketten, welche aufgrund der Base64 Kodierung zusätzlich noch einmal 33% mehr Bandbreite benötigen.

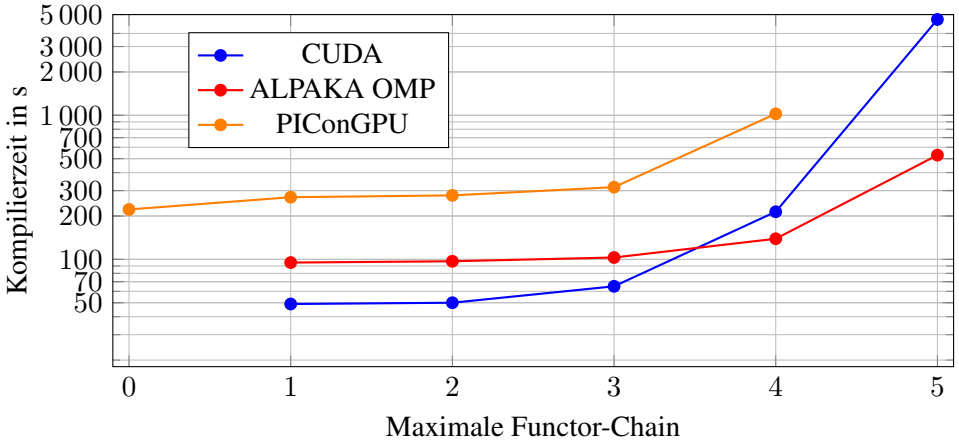

Abbildung 6.12: Benötigte Kompilierzeit für PIConGPUund die Beispielanwendung sowohl mit CUDA als auch OpenMP in Abhängigkeit von der maximalen Länge der Functor-Chain. Es zeigt sich ein superexponentielles Wachstum.

Noch höher ist die Latenz bei dem optionalen Twitch Plugin. Der Stream wird hier bis zu 10 Sekunden nach seiner Entstehung im ISAAC Server auf der Twitchwebseite angezeigt. Andererseits gibt es bei Twitch keine Feedbackmöglichkeit, womit die Latenz u. U. gar nicht auffällt. Dafür ermöglicht es tausenden von NutzerInnen, parallel die Visualisierung zu betrachten, was die anderen Streamarten von ISAAC nicht leisten können.

6.1.4 Kompiliergeschwindigkeit

Es konnte gezeigt werden, dass die Grundidee der TMP, Teile des Programmcodes schon zur Kompilierzeit auszuführen, die Laufzeit stark beschleunigen kann. Andererseits erhöht sich damit aber auch die Kompilierzeit, wie in Abbildung 6.12 zu sehen ist. Die Grafik zeigt die Zeit in Sekunden, die PIConGPU und die Beispielanwendung für CUDA und ALPAKA mit OpenMP für das Kompilieren in Abhängigkeit von der maximalen Functor-Chain Länge braucht. Als Vergleich wurde des Weiteren gemessen, wie lange PIConGPU ohne ISAAC zum Kompilieren braucht (Wert 0 des orangefarbenen Graphen). Wie erwartet, steigt die Kompilierzeit exponentiell. Zur Erinnerung: In Abhängigkeit von der maximalen Functor-Chain Länge n und der Anzahl der Functors c müssen $4 \cdot c^n$ (5.1.1) verschiedene Funktionen zur Kompilierzeit generiert werden. In der logarithmischen Darstellung der Graphen müsste dabei eine Gerade entstehen. Stattdessen zeigt sich auch hier eine exponentielle Kurve. Das bedeutet, dass die Kompilierzeit superexponentiell steigt.

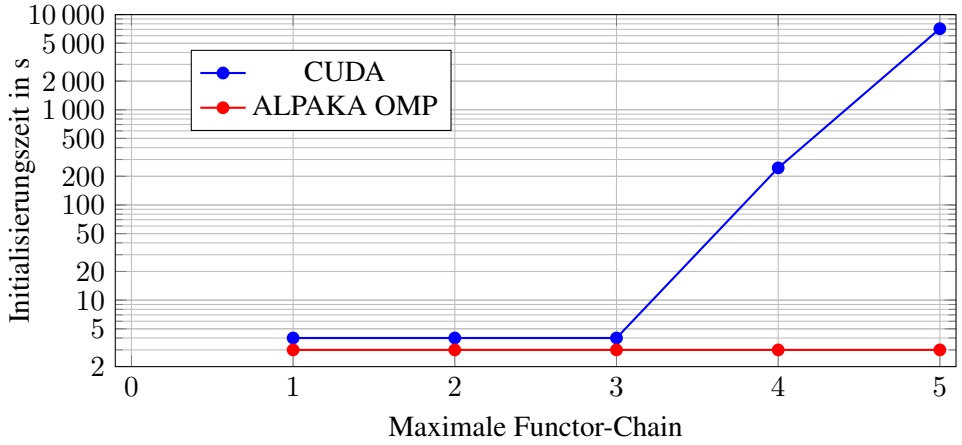

Abbildung 6.13: Benötigte Initialisierungszeit für die Beispielanwendung sowohl mit CUDA als auch OpenMP. Ab einer Functor-Chain Länge von 3 zeigt sich für CUDA ein exponentielles Wachstum.

Im Falle von PIConGPU war es nicht möglich, mehr als vier Functors zu benutzen, da der Nvidia Compiler Version 7.5 bei einer Functor-Chain Länge von 5 auf dem Testsystem nach über 3 Stunden abgestürzt ist. Der Grund für das superexponentielle Verhalten könnte sein, dass der Compiler innerhalb einer Funktion in Abhängigkeit von deren Länge in exponentieller Zeit Optimierungen vornimmt. Die Kompilierzeit ergäbe sich dann nach $4 \cdot c^n \cdot n^k \cdot t$, mit dem compilerspezifischen Exponentem k und der Zeitkonstanten t. Die Kompilierzeit von dem CUDA Beispiel und PIConGPU zeigten dabei das gleiche Wachstum. Die OpenMP Variante zeigte für kleine Werte eine langsamere Kompilierung, dafür aber ein wesentlich kleineres Wachstum für große n, besitzt also eine höhere Zeitkonstante t, aber einen kleineren Exponenten k.

Obwohl die Laufzeitunterschiede zwischen `idem` und komplexeren Functor-Chains (siehe Abschnitt 6.1.2) nur marginal sind und die Kompilierzeitoptimierungen scheinbar kaum ins Gewicht fallen, ist die Vorkompilierung trotzdem wichtig, um nur einen statt n Funktionsaufrufen im Kernel zu haben. In der Praxis hat sich gezeigt, dass drei Functors ausreichen, um die meisten Quellen hinreichend zu analysieren. Soll beispielsweise nur die Z-Komponente einer Vektorquelle betrachtet werden, deren Vorzeichen von Interesse ist und die auf die Transferfunktionsgröße skaliert werden muss, wäre die optimale Functor-Chain `mul(0,0,1000) | sum | add(0.5)` mit dem fiktiven Skalierungsfaktor 1000.

Ein unerwarteter Effekt zeigt sich jedoch noch bei der Ausführung der Simulation zusammen mit der ISAAC Visualisierung. Abbildung 6.13 zeigt die Zeit, die beim

Start des Programms benötigt wird in Abhängigkeit von der maximalen Functor-Chain Länge n. Scheinbar initialisiert die CUDA Laufzeitbibliothek bei dem ersten Zugriff bzw. der Allozierung von Grafikkartenspeicher alle in der Anwendung vorkommenden Devicefunktionspointer. Ab einer Functor-Chain Länge von $n = 4$ braucht die Laufzeitumgebung schon über vier Minuten zusätzlich – bei einer Functor-Chain Länge von $n = 5$ sogar von knapp zwei Stunden. Die CUDA Runtime scheint die initialisierten Werte jedoch für weitere Ausführungen auch zu cachen. Bei der zweiten Ausführung zeigte sich unabhängig von der Functor-Chain Länge die Bestzeit der gewählten Methode. Simulationen wie PIConGPU, die viele Parameter aus Geschwindigkeitsgründen zur Kompilierzeit festlegen, können davon jedoch nicht profitieren, wenn eine Neukompilierung obligatorisch ist. Es ergibt sich also auch deshalb eine optimale Functor-Chain Länge von $n = 3$.

6.2 Skalierbarkeit

Es konnte gezeigt werden, dass sich das Compositing von IceT für große Knotenanzahlen mit dem Binary Swap Algorithmus einer konstanten Zeit annähert. Des Weiteren ist die Laufzeit der Komprimierung und des Nachrichtenhandlings auf dem Master unabhängig von der Gesamtzahl der vorhandenen Knoten. Das bedeutet, dass nur die Skalierbarkeit des Kernels ein Bottleneck auf Exascalesystemen darstellen könnte – unter der Prämisse, dass zu Grunde liegende Technologien wie MPI ebenfalls auf solch großen Systemen skalieren.

Der ISAAC Kernel skaliert hierbei sowohl für Weak als auch Strong Scaling. Im ersteren Fall, wenn nach Belieben neue lokale Subvolumen hinzugefügt werden, können diese weiterhin in derselben Laufzeit berechnet werden. Wenn das gesamte, globale Volumen zu sehen ist, wird die Laufzeit aufgrund der Erkenntnisse aus Abbildung 6.6 sogar sinken, da das zu rendernde Bild pro Knoten kleiner wird. Der Worst Case, dass ein Knoten den alleinigen Fokus hat, bleibt identisch.

Aber auch wenn eine Simulation von Strong Scaling profitiert, wird auch der ISAAC Kernel schneller werden, wie in Abbildung 6.3 gezeigt wurde. Jedoch muss beachtet werden, dass die Laufzeit von IceT und die nicht parallelisierbaren Abarbeitungsschritte wie Nachrichtenhandling und Bildkomprimierung die Laufzeit nicht beliebig reduzieren lassen. In beiden Fällen wird sich also ISAAC einer minimalen echtzeitfähigen Ausführungszeit annähern.

Es konnte also gezeigt werden, dass ISAAC in der Lage ist, für beliebig große Systeme zu skalieren, und bietet eine zukunftssichere Analyse- und Visualisierungsmöglichkeit. Eine Skalierung auf Peta- oder gar Exascalesysteme scheint möglich.

6.3 Wissenschaftlicher Nutzen am Beispiel PIConGPU

Eine schnelle Visualisierung birgt keinen Nutzen, wenn sie dem Erkenntnisgewinn der BetrachterInnen nicht zuträglich ist. Neben der obligatorischen Möglichkeit einfach die Visualisierungsergebnisse ansehen zu können, muss die Visualisierung auch korrekt und zuverlässig sein. Das soll examplarisch für PIConGPU gezeigt werden.

Da der Referenzclient eine Webanwendung ist bzw. durch die offene Struktur beliebige, weitere Implementierungen möglich wären, ist es möglich auf verschiedenen Systemen von einer Workstation (mit beliebigem Betriebssystem) über ein Notebook bis hin zu einem Tablet oder Smartphone, die Simulation nicht nur zu beobachten, sondern auch zu steuern. Eine typische PIConGPU-Simulation braucht für einen Zeitschritt mindestens $20\,\text{ms}$. Das bedeutet, dass die Wartezeit für das Komprimieren und Senden von Bildern komplett überbrückt werden kann. Bei einer Testausführung der Laser-Kielfeld-Beschleuniger-Simulation von PIConGPU bei einer Auflösung von 1024×768 und lokalen Volumen der Größe $128 \times 64 \times 128$ mit 64 beteiligten GPUs auf Hypnos brauchte PIConGPU je nach Zeitpunkt in der Simulation einschließlich des ISAAC Renderings $35\,\text{ms}$ bis $100\,\text{ms}$. ISAAC selbst hat maximal $15\,\text{ms}$ für die Darstellung eines Bildes gebraucht. Das bedeutet, dass ISAAC die Simulation zum einen kaum beeinträchtigt – insbesondere nicht, wenn sie $85\,\text{ms}$ oder mehr Zeit pro Zeitschritt braucht – und zum anderen, dass das Komprimieren und Senden der Bilder komplett von der Simulation überlagert werden kann.

Abbildungsmächtigkeit Viele Aspekte PIConGPUs sind in ISAAC abbildbar – nicht zuletzt, weil es auch in Hinblick auf diese Referenzimplementierung designt wurde. So nutzt die Laser-Wakefield-Simulation z. B. die Möglichkeit eine Skalierung des Beobachtungsraums bzgl. des Rechenraumes anzugeben, wenn beide unterschiedliche Verhältnisse besitzen. Eine Gegenüberstellung der korrekten und simulationsinternen Darstellung ist in Abbildung 6.14 zu sehen.

Allgemein kann das Verständnis der Kielfeld-Beschleunigung mithilfe von ISAAC gesteigert werden, wie in dem Zeitraffer in Abbildung 6.15 zu sehen ist. Sowohl der Laserpuls (rot) als auch dessen Magnetfeld (grün) sind gut zu erkennen. Die Elektronendichte ist purpur dargestellt und die Stromstärke blau. Es zeig sich, wie in Abschnitt 3 beschrieben, eine Ablenkung der Elektronen von dem Puls, sowie mehrere kugelförmige, elektronenfreie Zonen und Häufungen von Elektronen hinter diesen. Auf den letzten Bildern kann sogar noch eine Selbstinjektion von Elek-

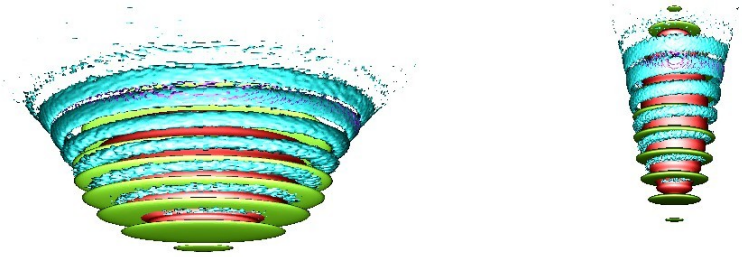

Abbildung 6.14: Vergleich der physikalisch korrekten, gestreckten Darstellung der Laser-Wakefield-Simulation (links) mit der unkorrekten Darstellung im Berechnungsraum (rechts). Zeitschritt und Blickwinkel sind identisch.

tronen in die positiv geladene Kugel direkt hinter dem Laserpuls beobachtet werden bevor sich die Kugeln komplett auflösen.

Abbildung 6.16 zeigt eine Kelvin-Helmholtz-Instabilität im 1140. (links) und 1287. Zeitschritt (rechts). Durch die rote Klassifizierung positiver Stromstärkewerte und die grüne von negativen zeigt sich das typische Ineinanderfließen an den Grenzschichten.

Für Abbildung 6.17 wurden zwei Elektronendichtequellen in Abhängigkeit der Fließrichtung definiert. Es zeigt sich, wie sich die beiden Strömungen ineinander verwirbeln. Der Effekt lässt sich sowohl mit dem Emissions/Absorptions-Modell (links) als auch mit einer Isoflächendarstellung (rechts) gut visualisieren. Rechts wurde zudem noch das elektrische Feld in blau mit angezeigt. Jedoch gerade die Darstellung als (leuchtendes) Gas zeigt die Ähnlichkeit dieser Plasmainstabilität mit der in den Grundlagen motivierten Kelvin-Helmholtz-Instabilität bei Räucherstäbchen.

Auf den Grafiken fällt bei genauem Hinsehen eine horizontale Kante exakt an der Überschneidungszone auf. Diese entsteht dadurch, dass die Quelle keinen Guard definiert, aber trotzdem mit Interpolation gerendert wird. Der Interpolationsalgorithmus geht davon aus, dass bei fehlendem Guard der Randwert weiter gilt (*Clamping*).

Abbildung 6.18 zeigt für 6 Zeitschritte einer Weibel-Instabilitäts-Simulation mit PIConGPU die Elektronendichte. Es fällt auf, dass die Instabilität zwischen Zeitschritt $7\,000$ und $8\,000$ mehr Änderungen erfährt als zwischen Zeitschritt 0 und $5\,000$. Daran lässt sich gut die Selbstverstärkung zeigen. Ungefähr zum Zeitpunkt $8\,000$ erreicht die Instabilität ihr Maximum und beginnt sich danach wieder zu vermischen. Die Instabilität besitzt zwar theoretisch ein exponentielles und unendli-

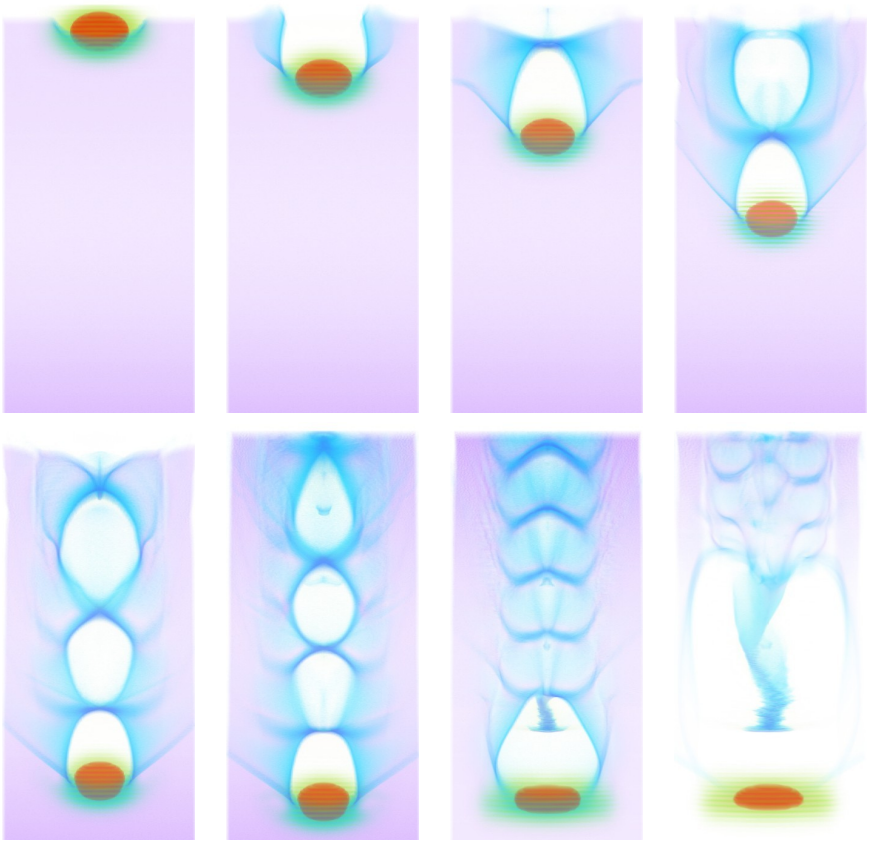

Abbildung 6.15: Zeitrafferaufnahme vom Kielfeldbeschleuniger. Die Entstehung der Kielfelder hinter dem Laserpuls ist gut zu erkennen.

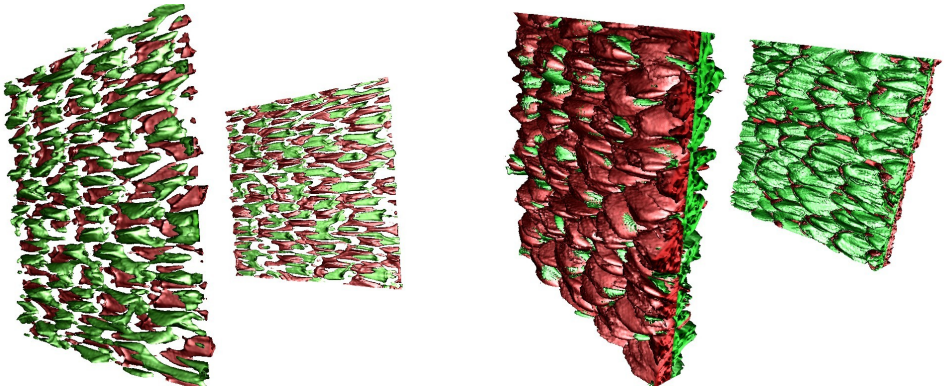

Abbildung 6.16: Isoflächendarstellungen einer Kelvin-Helmholtz-Instabilität in PIConGPU. Es wird nur die X-Komponente der Stromstärke gezeigt. Negative Werte sind rot, positive Werte grün eingefärbt. Der Betrag des Schwellwerts wurde links höher als rechts gewählt.

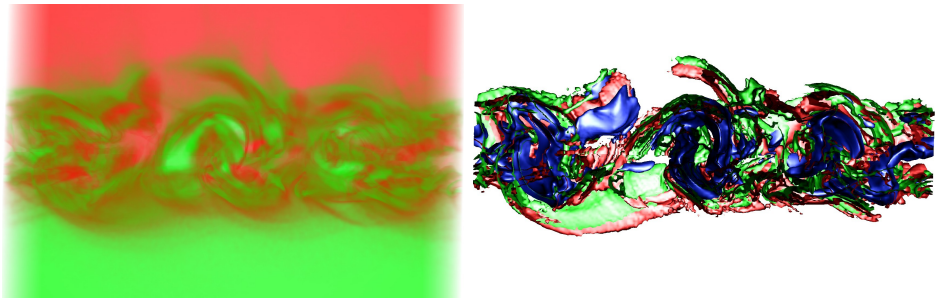

Abbildung 6.17: Darstellung einer Kelvin-Helmholtz-Instabilität mit unterschiedlicher Einfärbung der Elektronendichten in Abhängigkeit der Flussrichtung der Elektronen. Links erfolgt die Darstellung als leuchtendes Gas, rechts mit Isoflächendarstellung und zusätzlich mit dem elektrischen Feld (blau).

ches Wachstum, ist aber sowohl in der Realität als auch in dieser Simulation an die räumlichen Grenzen gebunden.

Dieselbe Instabilität lässt sich auch mit einer Isoflächendarstellung visualisieren. Dazu wurde für Abbildung 6.19 sowohl die X- als auch die Y-Komponente des Magnetfeldes als Fläche dargestellt, wobei negative Werte rot und positive grün dargestellt werden. Hier zeigen sich sehr klar die typischen, schlauchförmigen Magnetfelder, die in den Grundlagen postuliert wurden. Um die X-Komponente des Magnetfeldes darzustellen, wurde die Functor-Chain `mul(10000) | add(0.5)` und für die Y-Komponente `mul(0,10000,0) | sum | add(0.5)` genutzt. Im zweiten Fall werden die X- und Z-Komponente explizit mit 0 multipliziert, damit bei der Aufsummierung auf eine Dimension nur die Y-Komponente einfließt.

Es hat sich also gezeigt, dass ISAAC gut in der Lage ist, die verschiedenen Felder PIConGPUs für verschiedene Simulationstypen zu visualisieren. Sowohl die Möglichkeit der Darstellung mittels Emissions/Absorptions-Modell als auch mit Isoflächen, aber auch die Functor-Chains und die Möglichkeit Beobachtungsraum und Rechenraum verschieden zu skalieren, konnten helfen, schon bei der Ausführung der Simulationen die Startparameter zu evaluieren oder interessante Zeitschritte herauszufinden. Dabei konnte in den vorherigen Abschnitten auch gezeigt werden, dass ISAAC die Laufzeit von Simulationen wie PIConGPU nur wenig verlängert. Bei einer zu großen Belastung gibt es weiterhin die Möglichkeit über viele Parameter die Laufzeit pro Frame zu Lasten der Qualität zu verringern oder nicht für alle Zeitschritte ein Bild zu generieren.

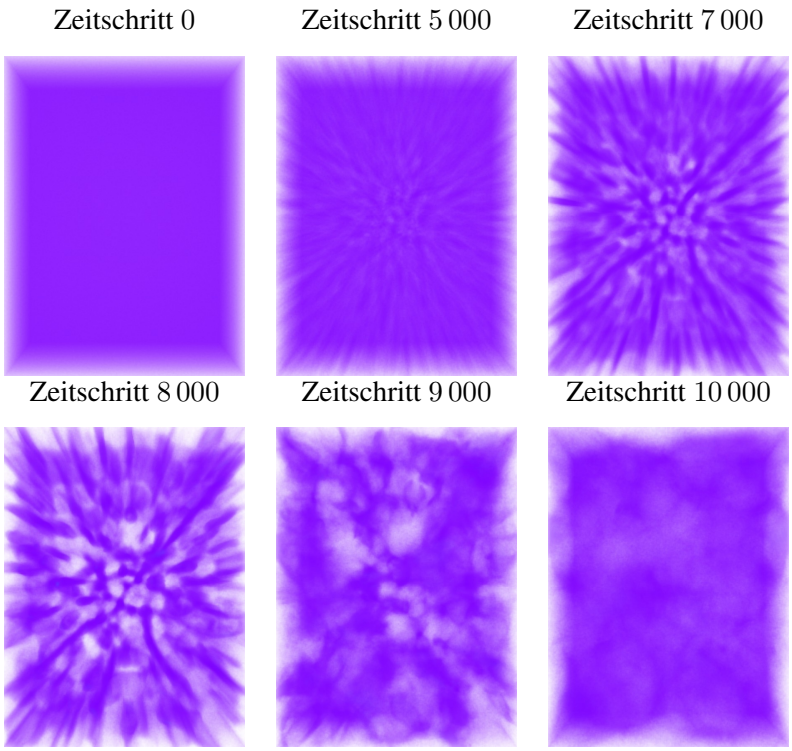

Abbildung 6.18: Weibel-Instabilität zu verschiedenen Zeitpunkten. Zu sehen ist die Elektronendichte in purpur. Mit steigendem Zeitschritt verstärkt sich die Instabilität. Ungefähr bei Zeitschritt 8 000 ist die Separation am Größten. Danach beginnen sich die Elektronen wieder mehr zu verteilen.

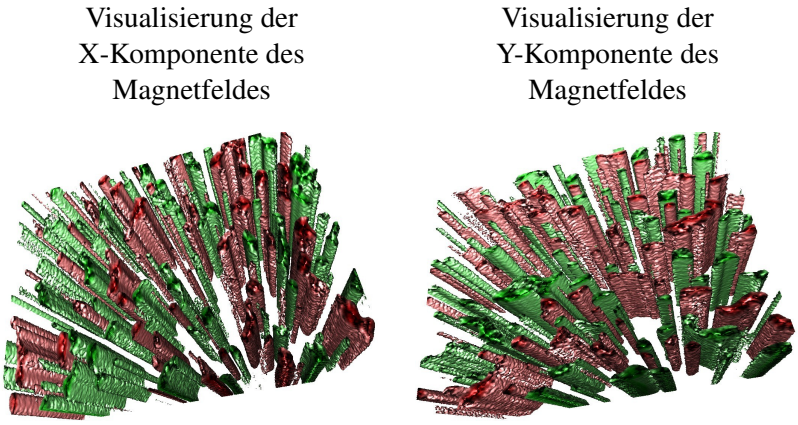

Abbildung 6.19: Darstellung der X- und Y-Komponente des Magnetfeldes einer Weibel Instabilität zum Zeitschritt 6605. Es zeigen sich die typischen Röhren.

Des Weiteren war es sehr trivial möglich, PIConGPU um die Möglichkeit einer Pausierung, aber auch der Abbrechung des Simulationsjobs direkt aus dem Webclient heraus zu erweitern.

7 Zusammenfassung

In dieser Arbeit wurde eine Lösung vorgestellt, um wissenschaftliche Simulationen und andere Hochratendatenquellen auf modernen Petascale-Hochleistungsrechern visualisieren zu können. Es konnte gezeigt werden, dass sich Anwendungen heutiger und zukünftiger HPC-Systeme aufgrund der umfangreichen Daten nicht mehr mit klassischem Postprocessing analyisieren und visualisieren lassen werden können. Auf Basis dieser Erkenntnis wurde die In-Situ Bibliothek ISAAC beschrieben, die Visualisierungen direkt auf den Rechenknoten erzeugt, auf welchen auch die zu visualisierenden Daten liegen. Um die Vorteile sehr generischer mit denen von sehr simulationsspezifischen In-Situ Lösungen zu kombinieren, wurde ein Template basiertes Interface beschrieben, was einerseits alle Simulationsdaten gleich abstrahiert, aber andererseits trotzdem anwendungsspezifisch optimiert werden kann, indem möglichst viele Aspekte mithilfe von Template Metaprogrammierung zur Kompilierzeit abgearbeitet werden.

Dank der Bibliothek ALPAKA kann ISAAC auf demselben Rechenbeschleuniger, auf dem auch die Simulation läuft, ausgeführt werden und insbesondere auf die originalen Simulationsdaten zugreifen, ohne sie kopieren zu müssen. Damit eine sinnvolle Visualisierung der Daten möglich ist, wurden Functor-Chains beschrieben, die in der Lage sind, den Wertebereich der betrachteten Felder für das Raycasting zu transformieren. ISAAC unterstützt dabei sowohl eine Darstellung mittels Emissions/Absorptions-Modells als auch Isoflächendarstellung. Die Klassifizierung der Daten erfolgt über Transferfunktionen. Des Weiteren ist es möglich, beliebige und beliebig viele Clippingebenen zu beschreiben. Neben der In-Situ Bibliothek wurde eine zentrale Serverkomponente zur Streamerzeugung und zur Verbindung mit beliebigen Clients motiviert, beschrieben und implementiert. Außerdem wurde ein HTML5 Referenzclient zur Steuerung der Visualisierung und Simulation implementiert, mit der einfachen Möglichkeit für beliebige Simulationstypen spezialisiert zu werden. Neben der Visualierungserstellung und Anzeige ist es dank ISAAC auch möglich, beliebige Metadaten zwischen Client und Simulation bzw. Visualisierung auszutauschen. Zu guter Letzt wurde ISAAC in die Plasmasimulation PIConGPU eingebunden, um den Laser-Kielfeld-Beschleuniger und sowohl die Kelvin-Helmholtz- als auch die Zweistrom-Instabilität direkt während der Simulation auf dem HZDR-Cluster Hypnos zu visualisieren.

Es konnte gezeigt werden, dass die Lösung selbst bei Full HD Auflösung echtzeitfähig bleibt, die konkrete Laufzeit aber von vielen weiteren Faktoren wie Anzahl der Quellen, Größe des Volumens pro Rechenknoten oder der Schrittweite im Raycasting abhängt. Auf diese Art und Weise kann zur Laufzeit feingranular zwischen besonders schnellen oder detailierteren Visualisierungen gewählt werden. Durch die Vorkompilierung vieler Einstellungen entstand zwar eine sehr schnelle Visualisierung, bei vielen Kombinationen von Quellen oder Functors wächst die Kompilierungszeit jedoch stetig. Auch die Initialisierungszeit auf Rechenbeschleunigern von Nvidia steigt mit der Functor-Chain Länge exponentiell. Es muss also ein Kompromiss zwischen Kompilierzeit- und Laufzeiteinstellungen gefunden werden.

8 Ausblick

Der Stand ISAACs zur Abgabe dieser Arbeit ist in vielerlei Hinsicht erweiterbar. Im Moment gibt es eine globale Option, welche Darstellungsart in der Visualisierung gewählt wird. Für die Zukunft wäre es denkbar, pro Quelle zu entscheiden, ob das Emissions/Absorptions-Modell oder eine Isoflächendarstellung genutzt wird. Die Transferfunktionen und Functor-Chains müssen außerdem immer wieder per Hand eingestellt werden. Das Speichern von tauglichen Transferfunktionen und Functor-Chains für eine Simulation oder das Erzeugen, z. B. mithilfe von Histogrammen der Simulationsdaten, wären also sinnvolle Erweiterungen. Auch der ISAAC Server könnte um weitere Funktionen, z.B. für andere Streamingformate wie HLS, RTSP, IceCast oder WebRTC – eine neue Videotelefonietechnologie moderner Webbrowser – erweitert werden.

Die Visualisierung erfolgt zwar direkt auf dem Rechenbeschleuniger, aber die Kombinierung der Einzelbilder mittels IceT muss im Moment noch auf dem Host geschehen. In Zukunft wäre es denkbar, NVIDIA GPUDirect [Nvi16b] bzw. ähnliche Konzepte für andere Arten von Rechenbeschleunigern zu nutzen, mit denen es möglich ist, direkt vom Beschleunigergerät auf die Netzwerkhardware zuzugreifen, ohne dass eine Hostkopie notwendig ist. Des Weiteren könnte ein Pixel sofort, nachdem er fertig ist, zur Weiterverarbeitung über das Netzwerk geschickt werden, anstatt wie bisher das Netzwerk während des Renderings gar nicht auszulasten. Das verlangt aber auch nach speziellen MPI-Implementierungen, die diese Möglichkeiten ausnutzen. Allgemein stellt sich die Frage, ob ISAAC weiterhin starr auf MPI setzen sollte oder stattdessen flexiblere, graphenbasierte, abstrahierende Kommunikationsbibliotheken wie z. B. Graybat [Zen16] genutzt werden sollten, die zum einen MPI im Backend nutzen können, aber zum anderen auch die Möglichkeit offenbaren andere Kommunikationsbackends zu nutzen – insbesondere, wenn diese GPUDirect besser unterstützen.

Die Volumenvisualisierung ist zwar relativ performant, aber gerade bei sehr großen Volumen oder sehr kleinen Schrittweiten kann sie die Echtzeitfähigkeit verlieren. Ein Grund ist der mehr oder minder zufällige Zugriff auf den Speicher, der es dem Rechenbeschleuniger für bestimmte Blickwinkel schwer macht, Werte zu cachen. Ein intelligentes Vorladen von bestimmten, aufeinanderfolgenden Speicherbereichen könnte die Cache-Misses reduzieren und die Rendergeschwindigkeit weiter

steigern. Auch das schon angesprochene Einrasten der Kamerablickrichtung für Winkel, die weniger Cache-Misses erzeugen, sollte in Zukunft näher untersucht werden.

Bisher wurde sich darauf konzentriert eine In-Situ Volumenvisualisierung zu generieren. Jedoch eignet sich dasselbe Konzept auch für andere Visualisierungsarten oder Datenanalyseaufgaben, die nicht zwingend Bilder erzeugen müssen. Des Weiteren sind die Functor-Chains zwar schnell und hilfreich, stoßen aufgrund der Einfachheit der Functors und der maximalen Länge der Chains schnell an ihre Grenzen. Hier ergibt sich also noch eine Menge Potential zur Verbesserung und Erweiterung.

Einige Aspekte ISAACs werden auch ohne explizite Arbeit an dieser Software in Zukunft verbessert werden. Da ISAAC auf die Rechenbeschleunigerabstraktionsbibliothek ALPAKA aufbaut, werden alle zukünftig unterstützten Backends auch für ISAAC nutzbar sein. Bisher waren AMD GPUs nur mittels OpenCL programmierbar, welches von ALPAKA aufgrund der fehlenden C++ Unterstützung nicht abstrahiert werden kann. Jedoch stellte AMD 2015 eine CUDA ähnliche Programmiermöglichkeit für seine GPUs namens HIP vor [MS16], die ALPAKA in Zukunft genauso abstrahieren könnte. Des Weiteren ist ISAAC bisher nicht in der Lage Vektorrecheneinheiten moderner CPUs, wie z.B. SSE oder AVX zu nutzen, obwohl ALPAKA explizit dabei hilft. Auch an dieser Stelle bieten sich also Optimierungsmöglichkeiten an.

Es zeigt sich also, dass ISAAC das Problem der In-Situ Visualisierung ohne jegliche Host-Kopie zwar effizient lösen könnte, aber noch viel Forschungsbedarf in dieser Richtung existiert.

Literaturverzeichnis

[AP98] AHRENS, James ; PAINTER, James: Efficient Sort-Last Rendering Using Compression-Based Image Compositing. In: *Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization* (1998)

[BAB+12] BENNETT, Janine C. ; ABBASI, Hasan ; BREMER, Peer-Timo ; GROUT, Ray ; GYULASSY, Attila ; JIN, Tong ; KLASKY, Scott ; KOLLA, Hemanth ; PARASHAR, Manish ; PASCUCCI, Valerio ; PEBAY, Philippe ; THOMPSON, David ; YU, Hongfeng ; ZHANG, Fan ; CHEN, Jacqueline: Combining In-situ and In-transit Processing to Enable Extreme-Scale Scientific Analysis. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (2012)

[BBC+13] BUSSMANN, M. ; BURAU, H. ; COWAN, T. E. ; DEBUS, A. ; HUEBL, A. ; JUCKELAND, G. ; KLUGE, T. ; NAGEL, W. E. ; PAUSCH, R. ; SCHMITT, F. ; SCHRAMM, U. ; SCHUCHART, J. ; WIDERA, R.: Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA : ACM, 2013 (SC '13). – ISBN 978–1–4503–2378–9, 5:1–5:12

[Bra16] BRAY, T.: *The JavaScript Object Notation (JSON) Data Interchange Format*. `https://tools.ietf.org/html/rfc7159`. Version: 29 Februar 2016

[Bru09] BRUCK, Nils van d.: *Techniken zur Artefaktreduktion für GPU-basiertes Ray Casting*. Zuse Institut Berlin, Freien Universität Berlin, Institut für Informatik, 16 Juli 2009

[BSH96] BATTKE, Henrik ; STALLING, Detlev ; HEGE, Hans-Christian: *Fast Line Integral Convolution for Arbitrary Surfaces in 3D*. Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1996

[BSO+11] BIDDISCOMBE, John ; SOUMAGNE, Jerome ; OGER, Guillaume ; GUIBERT, David ; PICCINALI, Jean-Guillaume: Parallel Computa-

tional Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2011)

[BWH+10] BURAU, Heiko ; WIDERA, René ; HÖNIG, Wolfgang ; JUCKELAND, Guido ; DEBUS, Alexander ; KLUGE, Thomas ; SCHRAMM, Ulrich ; COWAN, Tomas E. ; SAUERBREY, Roland ; BUSSMANN, Michael: PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. In: *IEEE TRANSACTIONS ON PLASMA SCIENCE* 38 (2010), Oktober, Nr. 10

[CBW+12] CHILDS, Hank ; BRUGGER, Eric ; WHITLOCK, Brad ; MEREDITH, Jeremy ; AHERN, Sean ; PUGMIRE, David ; BIAGAS, Kathleen ; MILLER, Mark ; CYRUS, Harrison ; WEBER, Gunther H. ; KRISHNAN, Hari ; FOGAL, Thomas ; SANDERSON, Allen ; GARTH, Christoph ; BETHEL, E. W. ; CAMP, David ; RÜBEL, Oliver ; DURANT, Marc ; FAVRE, Jean ; NÁVRATIL, Paul: VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data. (2012)

[Ces16] CESANTA: *Mongoose - Embedded Web Server*. `https://github.com/cesanta/mongoose`. Version: 29 Februar 2016

[CGM+06] CEDILNIK, Andy ; GEVECI, Berk ; MORELAND, Kenneth ; AHRENS, James ; FAVRE, Jean: Remote Large Data Visualization in the ParaView Framework. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2006)

[CL93] CABRAL, Brian ; LEEDOM, Leith (.: Imaging Vector Fields Using Line Integral Convolution. In: *SIGGRAPH* (1993)

[CM93] CORRIE, Brian ; MACKERRAS, Paul: Parallel Volume Rendering and Data Coherence. In: *Parallel Rendering Symposium* (1993)

[CNLE09] CRASSIN, Cyril ; NEYRET, Fabrice ; LEFEBVRE, Sylvain ; EISEMANN, Elmar: GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In: *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2009)

[Eas16] EAST DESIRE: *jscolor - JavaScript Color Picker (Palette) with touch support*. `http://jscolor.com/`. Version: 29 Februar 2016

[EKE01] ENGEL, Klaus ; KRAUS, Martin ; ERTL, Thomas: High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001)

[EP07a] EILEMANN, Stefan ; PAJAROLA, Renato: Direct Send Compositing for Parallel Sort-Last Rendering. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2007)

[EP07b] EILEMANN, Stefan ; PAJAROLA, Renato: The Equalizer Parallel Rendering Framework / Department of Informatics, Unviersity of Zürich. 2007. – Forschungsbericht

[FBF+15] FERNANDES, Oliver ; BLOM, David S. ; FREY, Steffen ; ZUIJLEN, Alexander H. ; BIJL, Hester ; ERTL, Thomas: ON IN-SITU VISUALIZATION FOR STRONGLY COUPLED PARTITIONED FLUID-STRUCTURE INTERACTION. In: *VI International Conference on Computational Methods for Coupled Problems in Science and Engineering* (2015)

[FCS+10] FOGAL, Thomas ; CHILDS, Hank ; SHANKAR, Siddharth ; KRÜGER, Jens ; BERGERON, R. D. ; HATCHER, Philip: Large Data Visualization on Distributed Memory Multi-GPU Clusters. In: *High Performance Graphics* (2010)

[FGP+04] FAURE, Jérôme ; GLINEC, Yannick ; PUKHOV, A ; KISELEV, S ; GORDIENKO, S ; LEFEBVRE, E ; ROUSSEAU, J-P ; BURGY, F ; MALKA, Victor: A laser–plasma accelerator producing monoenergetic electron beams. In: *Nature* 431 (2004), Nr. 7008, S. 541–544

[FMT+11] FABIAN, Nathan ; MORELAND, Kenneth ; THOMPSON, David ; BAUER, Andrew C. ; MARION, Pat ; GEVECI, Berk ; RASQUIN, Michel ; JANSEN, Kenneth E.: The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library. In: *IEEE Symposium on Large Data Analysis and Visualization* (2011)

[FVS11] FANG, Jianbin ; VARBANESCU, Ana L. ; SIPS, Henk: A Comprehensive Performance Comparison of CUDA and OpenCL. In: *International Conference on Parallel Processing* (2011)

[GA16] GURTOVOY, Aleksey ; ABRAHAMS, David: *THE BOOST MPL LIBRARY - 1.56.0.* `http://www.boost.org/doc/libs/1_56_0/libs/mpl/doc/index.html`. Version: 29 Februar 2016

[GMS16] GUZMAN, Joel de ; MARSDEN, Dan ; SCHWINGER, Tobias: *Chapter 1.1 Fusion 2.2 - 1.56.0.* `http://www.boost.org/doc/libs/1_56_0/libs/fusion/doc/html/index.html`. Version: 29 Februar 2016

[Gre16] GREEN, Andy: *libwebsockets*. `https://libwebsockets.org/index.html`. Version: 29 Februar 2016

[GT16] GSTREAMER-TEAM: *GStreamer: open source multimedia framework*. `https://gstreamer.freedesktop.org/`. Version: 29 Februar 2016

[HC11] HAGAN, R. ; CAO, Y.: Multi-GPU Load Balancing for In-situ Visualization. In: *Int. Conf. on Parallel and Distributed Processing Techniques and Applications* (2011)

[HS89] HIBBARD, William ; SANTEK, David: Interactivity is the Key. In: *Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization*. New York, NY, USA : ACM, 1989 (VVS '89), S. 39–43

[Hsu93] HSU, William M.: Segmented Ray Casting for Data Parallel Volume Rendering / Cambridge Research Lab Digital Equipment Corporation. 1993. – Forschungsbericht

[JPPR05] JENS KRÜGER ; PETER KIPFER ; POLINA KONDRATIEVA ; RÜDIGER WESTERMANN: *A Particle System for Interactive Visualization of 3D Flows*. IEEE transactions on visualization and computer graphics, 2005

[KDH10] KARIMI, Kamran ; DICKSON, Neil G. ; HAMZE, Firaz: A Performance Comparison of CUDA and OpenCL. In: *arXiv preprint arXiv:1005.2581* (2010)

[KKH02] KNISS, Joe ; KINDLMANN, Gordon ; HANSEN, Charles: Multidimensional Transfer Functions for Interactive Volume Rendering. In: *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS* 8 (2002), Nr. 3, S. 270–285

[KKW05] KONDRATIEVA, Polina ; KRÜGER, Jens ; WESTERMANN, Rüdiger: The application of GPU particle tracing to diffusion tensor field visualization. In: *Visualization, 2005. VIS 05. IEEE* (2005)

[KW03] KRÜGER, Jens ; WESTERMANN, Rürdiger: Acceleration Techniques for GPU-based Volume Rendering. In: *Visualization, 2003. VIS 2003. IEEE* (2003)

[KY14] KAGEYAMA, Akira ; YAMADA, Tomoki: An Approach to Exascale Visualization: Interactive Viewing of In-Situ Visualization. In: *Computer Physics Communications* (2014)

[Lac95] LACROUTE, Philippe G.: Fast Volume Rendering Using A Shear-Warp Factorization Of The Viewing Transformation / Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University. 1995. – Forschungsbericht

[Leh16] LEHTINEN, Petri: *Jansson - C library for working with JSON data*. `http://www.digip.org/jansson/`. Version: 29 Februar 2016

[Len15] LENGYEL, Eric: *Projection Matrix Tricks*. `http://www.terathon.com/gdc07_lengyel.pdf`. Version: 30 September 2015

[lib16] LIBJPEG-TURBO PROJECT: *libjpeg-turbo, Main / libjpeg-turbo*. `http://libjpeg-turbo.virtualgl.org/`. Version: 9 März 2016

[LRN96] LEE, Tong-Yee ; RAGHAVENDRA, C. S. ; NICHOLAS, John B.: Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. In: *IEEE Transcations on Visualization and computer graphics* 2 (1996), Nr. 3

[LSE+11] LAKSHMINARASIMHAN, Sriram ; SHAH, Neil ; ETHIER, Stephane ; KLASKY, Scott ; LATHAM, Rob ; ROSS, Rob ; SAMATOVA, Nagiza F.: Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-Temporal Data. In: *Euro-Par 2011 Parallel Processing* (2011)

[LSG16] LANTINGA, Sam ; SDL-GEMEINDE: *Simple DirectMedia Layer*. `https://www.libsdl.org/`. Version: 29 Februar 2016

[LVLRR08] LINSEN, Lars ; VAN LONG, Tran ; ROSENTHAL, Paul ; ROSSWOG, Stephan: *Surface Extraction from Multi-seld Particle Volume Data Using Multi-dimensional Cluster Visualization*. IEEE transactions on visualization and computer graphics, 2008

[LWMT97] LI, P. P. ; WHITMAN, Scott ; MENDOZA, Roberto ; TSIAO, James: ParVox: A Parallel Splatting Volume Rendering System for Distributed Visualization. In: *Parallel Rendering, 1997. (PRS '97). Proceedings., IEEE Symposium on* (1997)

[Ma95] MA, Kwan-Liu: Runtime Volume Visualization For Parallel CFD / Insitute for Computer Applications in Science and Engineering, NASA Langley Research Center. 1995. – Forschungsbericht

[Ma09] MA, Kwan-Liu: In Situ Visualization at Extreme Scale: Challenges and Opportunities. In: *IEEE Computer Graphics and Applications*

(2009)

[MAGM11] MORELAND, Kenneth ; AYACHIT, Utkarsh ; GEVECI, Berk ; MA, Kwan-Liu: Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale. In: *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on* (2011)

[MCEF94] MOLNAR, Steven ; COX, Michael ; ELLSWORTH, David ; FUCHS, Henry: A Sorting Classification of Parallel Rendering. In: *Computer Graphics and Applications, IEEE* (1994)

[MCH03] MARKUS HADWIGER ; CHRISTOPH BERGER ; HELWIG HAUSER: *High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware*. VRVis Research Center, Vienna, Austria, 2003

[MFL+02] MALKA, Victor ; FRITZLER, S ; LEFEBVRE, E ; ALEONARD, M-M ; BURGY, F ; CHAMBARET, J-P ; CHEMIN, J-F ; KRUSHELNICK, K ; MALKA, G ; MANGLES, SPD u. a.: Electron acceleration by a wake field forced by an intense ultrashort laser pulse. In: *Science* 298 (2002), Nr. 5598, S. 1596–1600

[MMD08] MARCHESIN, Stéphane ; MONGENET, Catherine ; DISCHLER, Jean-Michel: Multi-GPU Sort-Last Volume Visualization. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2008)

[MMN+04] MANGLES, SPD ; MURPHY, CD ; NAJMUDIN, Z ; THOMAS, AGR ; COLLIER, JL ; DANGOR, AE ; DIVALL, EJ ; FOSTER, PS ; GALLACHER, JG ; HOOKER, CJ u. a.: Monoenergetic beams of relativistic electrons from intense laser–plasma interactions. In: *Nature* 431 (2004), Nr. 7008, S. 535–538

[MPHK93] MA, Kwan-Liu ; PAINTER, James S. ; HANSEN, Charles D. ; KROGH, Michael F.: A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In: *Proceedings of the 1993 symposium on Parallel rendering* (1993)

[MPHK94] MA, Kwan-Liu ; PAINTER, James S. ; HANSEN, Charles D. ; KROGH, Michael F.: Parallel Volume Rendering Using Binary-Swap Image Composition. In: *IEEE GG&A* (1994)

[MPI16] MPICH: *MPI_Init_thread*. `http://www.mpich.org/static/docs/v3.1/www3/MPI_Init_thread.html`. Version: 29 Februar 2016

[MS16] MENGE-SONNENTAG, Rainald: *AMDs GPUOpen: Zahlreiche SDKs und Tools auf GitHub im Source verfügbar*. http://www.heise.de/developer/meldung/AMDs-GPUOpen-Zahlreiche-SDKs-und-Tools-auf-GitHub-im-Source-verfuegbar-3085309.html. Version: 5 März 2016

[MSE06] MÜLLER, C. ; STRENGERT, M. ; ERTL, T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2006)

[MSLP09] MARQUES, Ricardo ; SANTOS, Luis P. ; LESKOVSKY, Peter ; PALOC, Celine: *GPU Ray Casting*. Grupo Portugues de Computacao Grafica, 2009

[MWP01] MORELAND, Kenneth ; WYLIE, Brian ; PAVLAKOS, Constantine: Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays. In: *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (2001)

[MWYT07] MA, Kwan-Liu ; WANG, Chaoli ; YU, Hongfeng ; TIKHONOVA, Anna: In-situ processing and visualization for ultrascale simulations. In: *Journal of Physics: Conference Series 78* (2007)

[NGYD10] NING, Liu ; GUOXIAN, Gao ; YINGPING, Zhang ; DENGMING, Zhu: The Design and Implement of Ultra-scale Data Parallel In-situ Visualization System. In: *I nternational Conference on Audio Language and Image Processing (ICALIP)* (2010)

[NKS+04] NONAKA, Jorji ; KUKIMOTO, Nobuyuki ; SAKAMOTO, Naohisa ; HAZAMA, Hiroshi ; WATASHIBA, Yasuhiro ; LIU, Xuezhen ; OGATA, Masato ; KANAZAWA, Masanori ; KOYAMADA, Koji: Hybrid Hardware-Accelerated Image Composition for Sort-Last Parallel Rendering on Graphics Clusters with Commodity Image Compositor. In: *Volume Visualization - VVS* (2004)

[NMR+92] NAPEL, Sandy ; MARKS, Michael P. ; RUBIN, Geoffrey D. ; DAKE, Michael D. ; MCDONNELL, Charles H. ; SONG, Sanuel M. ; ENZMANN, Dieter R. ; JEFFREY, R. B.: CT Angiography with Spiral CT and Maximum Intensity Projection. In: *Radiology* (1992)

[Nvi16a] NVIDIA: *CUDA Toolkit Archive*. https://developer.nvidia.com/cuda-toolkit-archive. Version: 29 Februar 2016

[Nvi16b] NVIDIA: *NVIDIA GPUDirect | NVIDIA Developer*. https://developer.nvidia.com/gpudirect. Version: 5 März 2016

[Nvi16c] NVIDIA: *PTX ISA :: CUDA Toolkit Documentation*. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#operand-costs. Version: 29 Februar 2016

[Pho75] PHONG, Bui T.: Illumination for Computer Generated Pictures. In: *Communications of ACM* 18 (1975), Nr. 8

[PPL+03] PARKER, Steven ; PARKER, Michael ; LIVNAT, Yarden ; SLOAN, Peter-Pike ; HANSEN, Charles ; SHIRLEY, Peter: Interactive Ray Tracing for Volume Visualization. In: *IEEE TRANSACTIONS ON COMPUTER GRAPHICS AND VISUALIZATION* (2003)

[RCMS11] RIVI, Marzia ; CALORI, Luigi ; MUSCIANISI, Guiseppa ; SLAVNIC, Vladimir: In-situ Visualization: State-of-the-art and Some Use Cases. In: *Partnership for Advanced Computing in Europe* (2011)

[RS02] REZK SALAMA, Christof: *Volume Rendering Techniques for General Purpose Graphics Hardware*. Universität Erlangen-Nürnberg, 2002

[Sab88] SABELLA, Paolo: A Rendering Algorithm for Visualizing 3D Scalar Fields. In: *Computer Graphics* 22 (1988), August, Nr. 4

[San16a] SANDIA NATIONAL LABORATORIES: *Sandia National Laboratories: IceT Documentation*. http://icet.sandia.gov/documentation/index.html. Version: 29 Februar 2016

[San16b] SANDIA NATIONAL LABORATORIES: *Sandia National Laboratories: IceT Download*. http://icet.sandia.gov/download/index.html. Version: 29 Februar 2016

[Sau16] SAUTER, Marc: *Die Xeon Phi beherbergt Intels bisher größten Chip*. http://www.golem.de/news/knights-landing-die-xeon-phi-beherbergt-intels-bisher-groessten-chip-1503-113211.html. Version: 29 Februar 2016

[Sch13] SCHNEIDER, Benjamin: *In Situ Visualization of a Laser-Plasma Simulation*, Faculty of Computer Science, Institute of Software and Multimedia Technology, Chair of Computer Graphics and Visualization, Diplomarbeit, 2013

[SCL+12] SU, Ching-Lung ; CHEN, Po-Yu ; LAN, Chun-Chieh ; HUANG, Long-Sheng ; WU, Kuo-Hsuan: Overview and Comparison of OpenCL and

CUDA Technology for GPGPU. In: *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)* (2012)

[SCMO10] STUART, Jeff A. ; CHEN, Cheng-Kai ; MA, Kwan-Liu ; OWENS, John D.: Multi-GPU Volume Rendering using MapReduce. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010)

[SDSM16] STROHMAIER, Erich ; DONGARRA, Jack ; SIMON, Horst ; MEUER, Martin: *November 2015, TOP500 Supercomputer Sites*. `http://www.top500.org/lists/2015/11/`. Version: 29 Februar 2016

[SSKE05] STEGMAIER, Simon ; STRENGERT, Magnus ; KLEIN, Thomas ; ERTL, Thomas: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In: *Volume Graphics, 2005. Fourth International Workshop on* (2005)

[SVS13] STONE, John E. ; VANDIVORT, Kirby L. ; SCHULTEN, Klaus: GPU-Accelerated Molecular Visualization on Petascale Supercomputing Platforms. In: *UltraVis '13 Proceedings of the 8th International Workshop on Ultrascale Visualization* (2013)

[TSH98] TIEDE, Ulf ; SCHIEMANN, Thomas ; HÖHNE, Karl H.: *High Quality Rendering of Attributed Volume Data*. Institute of Mathematics and Computer Science in Medicine University Hospital Eppendorf, Hamburg, 1998

[Tsu16] TSUJIKAWA, Tatsuhiro: *Nghttp2: HTTP/2 C Library*. `https://nghttp2.org/`. Version: 29 Februar 2016

[Twi16] TWITCH INTERACTIVE: *Twitch*. `http://www.twitch.tv/`. Version: 29 Februar 2016

[TYRG+06] TU, Tiankai ; YU, Hongfeng ; RAMIREZ-GUZMAN, Leonardo ; BIELAK, Jacobo ; GHATTAS, Omar ; MA, Kwan-Liu ; O'HALLARON, David R.: From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In: *ACM/IEEE SC\06 Conference* (2006)

[Vel03] VELDHUIZEN, Todd L.: C++ Templates are Turing Complete / Indiana University Computer Science. 2003. – Forschungsbericht

[Vid16] VIDEOLAN NON-PROFIT ORGANIZATION: *VideoLAN - VLC: Offizielle Webseite - Freie Multimedialösungen für alle Betriebssysteme!* `http://www.videolan.org/`. Version: 29 Februar 2016

[WAF+11] WOODRING, J. ; AHRENS, J. ; FIGG, J. ; WENDELBERGER, J. ; HABIB, S. ; HEITMANN, K.: In-situ Sampling of a Large-Scale Particle Simulation for Interactive Visualization and Analysis. In: *Eurographics/ IEEE-VGTC Symposium on Visualization* (2011)

[WE98] WESTERMANN, Rüdinger ; ERTL, Thomas: Efficiently Using Graphics Hardware in Volume Rendering Applications. In: *SIGGRAPH '98* (1998)

[WFM11] WHITLOCK, Brad ; FAVRE, Jean M. ; MEREDITH, Jeremy S.: Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2011)

[Wor15] WORPITZ, Benjamin: *Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures*, Technische Universität Dresden, Masterarbeit, 2015

[Wor16] WORPITZ, Benjamin: *alpaka - Abstraction Library for Parallel Kernel Acceleration*. `https://github.com/ComputationalRadiationPhysics/alpaka`. Version: 29 Februar 2016

[WSEE05] WEISKOPF, Daniel ; SCHRAMM, Frederik ; ERLEBACHER, Gordon ; ERTL, Thomas: *Particle and Texture Based Spatiotemporal Visualization of Time-Dependent Vector Fields*. IEEE Visualization'05, 2005

[YTB+06] YU, Hongfeng ; TU, Tiankai ; BIELAK, Jacobo ; GHATTAS, Omar ; LÓPEZ, Julio C. ; MA, Kwan-Liu ; O' HALLARON, David R. ; RAMIREZ-GUZMAN, Leonardo ; STONE, Nathan ; TABORDA-RIOS, Ricardo ; URBANIC, john: Remote Runtime Steering of Integrated Terascale Simulation and Visualization. In: *ACM/IEEE Supercomputing 2006 Conference* (2006)

[YWG+09] YU, Hongfeng ; WANG, Chaoli ; GROUT, Ray W. ; CHEN, Jacqueline H. ; MA, Kwan-Liu: A Study of In-Situ Visualization for Petascale Combustion Simulations / University of California. 2009. – Forschungsbericht

[YWM08] YU, Hongfeng ; WANG, Chaoli ; MA, Kwan-Liu: Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In: *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for* (2008)

[Zen16] ZENKER, Erik: *Graybat - Graph Approach for Highly Generic Communication Schemes Based on Adaptive Topologies*. `https://github.com/ComputationalRadiationPhysics/graybat`. Version: 5 März 2016

[ZWW+16] ZENKER, Erik ; WORPITZ, Benjamin ; WIDERA, René ; HUEBL, Axel ; JUCKELAND, Guido ; KNÜPFER, Andreas ; NAGEL, Wolfgang E. ; BUSSMANN, Michael: *Alpaka - An Abstraction Library for Parallel Kernel Acceleration*. feb 2016

Danksagung

Zuallererst möchte ich an dieser Stelle meiner Familie danken, die mich in der Zeit meines Studiums und insbesondere in der Zeit meiner Diplomarbeit stark unterstützte. Allen voran meiner Lebensgefährtin Romy, für die ich während meiner Diplomarbeit viel zu wenig Zeit hatte, meiner Mutter, die mich bis zuletzt finanziell unterstützt hat, aber auch meiner Schwester Katarina und meinem Schwager Steffen – nirgends macht Literaturrecherche solch einen Spaß wie in den Schweizer Bergen.

Weiterhin danke ich der Gruppe Computergestützte Strahlenphysik am HZDR, hier allen voran meinem Betreuer Dr. Michael Bussmann, der mir die letzten fünf Jahre die Möglichkeit gab, in dieser großartigen und sehr familiären Gruppe zu arbeiten und die Grundidee meiner Arbeit legte. Besonderer Dank geht auch an René Widera und Axel Hübl, die mich bei der Konzeptionierung und der Arbeit mit PIConGPU unterstützt haben. Des Weiteren danke ich dem Space Office 325, hier vor allem Erik Zenker und Carlchristian Eckert, die mir als lebende Nachschlagewerke für viele C++-Template- und Template-Metaprogrammierungsfragen – aber auch für viele andere Kleinigkeiten – mit Rat und Tat zur Seite standen.

An der TU Dresden danke ich Dr. Sebastian Grottel und Prof. Dr. Stefan Gumhold für die Betreuung meiner Arbeit und für die vielen, kleinen Hilfestellungen und Hinweise, ohne die die Arbeit heute nicht in dieser Form existieren würde.

Erklärungen zum Urheberrecht

Sofern nicht anders angegeben, stehen alle Abbildungen, die von Alexander Matthes in dieser Arbeit veröffentlicht wurden, unter der Creative Commons Attribution-Share Alike Lizenz (CC BY-SA) Version 3.0[1].

Die entwickelte Anwendung *ISAAC* steht unter der GNU Lesser General Public License (LGPL) Version 3[2]. Eine aktuelle Version der Anwendung ist auf Gibhub zu finden:

`https://github.com/ComputationalRadiationPhysics/isaac`

Die zu der Bearbeitungszeit dieser Arbeit aktuelle Version ist zu finden unter:

`https://github.com/ComputationalRadiationPhysics/isaac/tree/master@%7B2016-03-14%7D`

[1] https://creativecommons.org/licenses/by-sa/3.0/de/
[2] http://www.gnu.org/licenses/lgpl.html