UNIVERSITY OF TECHNOLOGY DRESDEN

FACULTY OF COMPUTER SCIENCE
INSTITUTE OF SOFTWARE AND MULTIMEDIA TECHNOLOGY
CHAIR OF COMPUTER GRAPHICS AND VISUALIZATION
PROF. DR. STEFAN GUMHOLD

Diploma Thesis

for the acquisition of the academic degree
Diplom-Informatiker

# In Situ Visualization of a Laser-Plasma Simulation

Benjamin Schneider
(Born 6. Dezember 1987 in Rodewisch)

Tutor:

Prof. Dr. rer. net. Stefan Gumhold
Dr. Sebastian Grottel
Dr. Michael Bussmann

Dresden, November 1, 2013

# Task

The task of this diploma thesis is to design and implement an in situ visualization of scalar fields for the laser plasma simulation PIConGPU, developed at the Helmholtz-Zentrum Dresden-Rossendorf (HZDR). Amongst others electrical and magnetic fields are simulated and should be visualized together. Due to the high output data rate storing or transfering the raw data to a workstation or visualization cluster is not feasible.

For that reason the simulation data should be visualized on the HPC cluster itself. A distributed volume renderer will be implemented in CUDA. The use of OpenGL is not possible due to the lack of a running X Server on the compute nodes of the system. The insertion of a clipping plane and the parameterization of the 1D transfer function should be possible. Inserting up to three clipping planes and a 2-dimensional transfer function can be implemented. The compositing scheme can be implemented as maximum intensity projection (MIP) or emission absorption model.

As the simulation runs in a massively parallel manner, distributed on multiple compute nodes, the local data in each GPUs memory is rendered into a sub image. Those sub images are combined into one final image and sent to a client outside the cluster for viewing. The compute nodes communicate through the Message Passing Interface (MPI). Optionally, optimizations such as empty space skipping and progressive refinement can be employed in the volume renderer.

In detail the following tasks should be fulfilled:

- Literature research in (CUDA-)volume rendering and distributed/parallel rendering
- Training in CUDA and MPI programming
- Implementation of the volume renderer in CUDA
- Visualization of up to two scalar fields simultaneously
- Implementation of a module for PIConGPU to process and render the data
- Composition of the partial images to one final image
- Implementation of a simple client program to interactively steer the simulation and view the final image
- Evaluate the performance of the single components as well as of the complete system
- Optional: optimize the volume renderer, insertion of three clipping planes and 2D transfer functions

# Statement of authorship

I hereby certify that the diploma thesis I submitted today to the examination board of the faculty of computer science with the title:

*In Situ Visualization of a Laser-Plasma Simulation*

has been composed solely by myself and that I did not use any sources and aids other than those stated, with quotations duly marked as such.

Dresden, November 1, 2013

Benjamin Schneider

## Kurzfassung

Seit es Computer gibt sind Simulationen ein essentielles Werkzeug in Wissenschaft und Forschung. Mit zunehmender Rechenleistung wachsen auch die Datenmengen, welche von diesen Simulationen generiert werden. An der Schwelle zum Peta- und Exa-Scale Computing ist der klassische Ansatz des Post-Processing, bei welchem einfach so viele Daten wie möglich gespeichert und im Nachgang ausgewertet werden, weder praktikabel noch zeitgemäß. Um neue Erkenntnisse aus den enormen Datenmengen zu gewinnen müssen verschiedenste Probleme, wie limitierte Speicherkapazität und Netzwerkbandbreite, überwunden werden. Hier bietet die Verarbeitung und Reduktion der Rohdaten zur Simulationszeit durch In Situ Visualisierung einen skalierbaren Lösungsansatz.

Am Beispiel von PIConGPU, einem Simulationscode zur Untersuchung plasmaphysikalischer Phänomene, zeigen wir, wie eine In Situ Visualisierung mit einer Simulation verbunden wird. Dabei verfolgen wir einen eng gekoppelten Ansatz um Datenbewegung und -replikation zu minimieren. Als Visualisierungsmethode wurde Volume Ray-Casting gewählt, da dies eine effektive Darstellung der Feld- und Partikeldaten erlaubt und gleichzeitig hochparallel in CUDA implementiert werden kann. Der Renderer selbst ist ein Plug-in, welches zusammen mit der Simulation verteilt auf mehreren GPU-Knoten eines HPC Systems läuft. Unter Nutzung der Simulationsdatenstrukturen erzeugen die einzelnen (MPI) Prozesse ein Teilbild aus den lokalen Daten im Speicher der GPU. Diese Teilbilder werden durch *sort-last compositing* zu einem Gesamtbild vereint und über einen Server an einen oder mehrere Clients außerhalb des Clusters gesendet. Nutzer können mit Hilfe des Clients die erzeugten Bilder betrachten und die Visualisierung steuern, in dem sie beispielsweise den Blickpunkt der virtuellen Kamera verändern oder die Simulation pausieren. Unser System ist in der Lage interaktive Bildraten zu liefern und gibt dem Nutzer die Möglichkeit schon während des Simulationslaufs tief in die erzeugten Daten einzublicken. Die auf dem *Hypnos* Rechencluster des Helmholtz-Zentrum Dresden-Rossendorf durchgeführte Performanceanalyse verspricht eine gute Skalierung auf größere HPC Systeme.

## Abstract

Since computers exist simulations are an essential instrument in science and research. With increasing computational power the amount of data generated by simulaitons also grows. At the dawn of peta- and exa-scale computing the classical post-processing approach, in which simply as many data as possible is saved for later examination, is neither practical nor contemporary. To gain new insights from the enormous data sets various problems, like limited disk capacity and network bandwidth, have to be overcome. Here, processing and reducing the raw data during simulation time through in situ visualization provides a scalable solution.

Using the example of PIConGPU, a simulation code for investigating plasma-physical phenomena, we show how in situ visualization is incorporated with a simulation. We follow a tightly-coupled approach to minimize data movement and replication. As visualization method volume ray-casting was chosen, because it allows for an effective presentation of the field and particle data and at the same time a highly-parallelized implementation in CUDA. The renderer is a plug-in which runs together with the simulation on multiple GPU nodes of an HPC system. By utilizing the simulation data structures each individual (MPI) process generates a partial image of the local data residing in the GPUs memory. These sub images are combined to a full image by *sort-last compositing* which is then sent via a server to one or more clients outside the cluster. By means of these clients users can view the generated images and steer the visualization, e.g. by changing the viewpoint or pausing the simulation. Our system is able to deliver interactive frame rates and allows users to take deep insight into the simulation while it is running. The performance analysis conducted on the *Hypnos* cluster at the Helmholtz-Zentrum Dresden-Rossendorf promises a good scalability on larger HPC systems.

# Contents

**1 Introduction** **3**
1.1 Motivation . . . 3
1.2 Related Work . . . 4
1.2.1 Volume Rendering . . . 4
1.2.2 Parallel and Remote Rendering . . . 5
1.2.3 In Situ Visualization . . . 6
1.3 Structure of this Thesis . . . 7

**2 Fundamentals** **9**
2.1 Compute Unified Device Architecture . . . 9
2.2 Message Passing Interface . . . 11
2.3 Simulation Code PIConGPU . . . 13
2.4 Volume Rendering . . . 19
2.5 Image Compositing . . . 25

**3 A Tightly-coupled Visualization System** **29**
3.1 Requirements . . . 29
3.2 Preconditions . . . 30
3.3 Architecture Overview . . . 31

**4 The Parallel Volume Rendering Module** **33**
4.1 Integration with PIConGPU . . . 33
4.2 Initialization . . . 35
4.3 The Interactive Visualization Loop . . . 37
4.3.1 Data Retrieval . . . 37
4.3.2 CUDA Volume Ray-Casting . . . 40
4.3.3 Pre-Integrated Transfer Functions . . . 44
4.3.4 Image Compositing . . . 46
4.3.5 Steering . . . 48
4.4 Clean-Up . . . 51

**5 The Visualization Server** **53**

**6 The Client** **57**

**7 Evaluation** **63**
7.1 Rendering Performance . . . 63
7.1.1 Varying Image Size . . . 63
7.1.2 Varying Simulation Grid Size . . . 64

7.1.3 Varying Number of GPUs . . . 65
7.2 Compositing Performance . . . 65
7.3 Interactivity . . . 66
7.4 Conclusion . . . 67

**8 Summary & Outlook** **69**
8.1 Results . . . 69
8.2 Future Work . . . 70

**Bibliography** **73**

**List of Figures** **79**

**List of Tables** **83**

**A Message Protocol** **85**

**B Running an Interactive In Situ Visualization** **87**

# 1 Introduction

## 1.1 Motivation

Simulations are an important instrument of science and research. Reaching from astronomical phenomena like supernovae over weather and climate change to physical processes on the atom level and below simulations can give us unrivaled insight. It is usually a step taken before performing experiments. Experiments which are expensive or even not feasible (e.g. galaxy collision).

In our concrete case the laser-plasma simulation PIConGPU developed at the Helmholtz-Zentrum Dresden - Rossendorf (HZDR) should allow researchers to investigate the interaction of laser pulses with ionized gas. The code is based on the Particle-in-Cell Algorithm [Har55]. The heavy load of the calculations is performed on graphics processing units (GPUs), thus the name PIConGPU. It is capable of computing several time steps per second for billions of particles and a simulation grid consisting of a few million cells. To analyze the simulation output, exceeding several gigabytes per second and sometimes based in a remote location such as the Oak Ridge National Laboratory (ORNL), home of the Titan supercomputer, the raw data cannot be transfered to the scientists workstation or stored on disk in its entirety. A disproportional amount of time would be spent for I/O operations during simulation time and post processing. This is due to the growing gap between processor speed and memory bandwidth, depicted in figure 1.1.

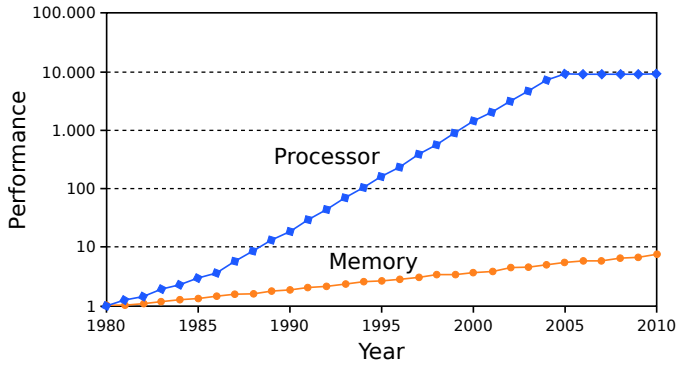



Figure 1.1: Processor-Memory Gap (after [HP12])

Additionally, commodity Desktop-PCs and even strong workstations are not able to process such a huge amount of data in an interactive manner. With the advent of exa-scale computing these issues become more and more pressing. A comparable sized machine like the HPC cluster generating the data is necessary to evaluate it. Another issue of supercomputing is energy efficiency. Every computation and every data movement costs time and energy. At its peak performance the Titan supercomputer consumes around eight Megawatts of electric power, enough to supply a small town with electricity.

For that reason it makes sense to use part of the computation capacities for reducing the data right at the moment and in the place it is generated. This procedure is called in situ processing. Visualization can reduce the amount of data from petabytes to several megabytes giving immediate feedback about the simulation run and allowing the scientists to possibly reconfigure and restart the simulation to observe the desired phenomena.

Thus, an in situ volume renderer, which visualizes the simulation data directly on the GPU reduces network traffic and storage requirements by several orders of magnitude. The renderer can utilize the simulation data structures, so no data needs to be replicated. The rendered images are then send to a server which can forward them to any client machine including desktop PCs, workstations and mobile devices like smartphones and tablets, which by themselves do not possess the computational capacity of the simulation cluster. The server decouples simulations and clients and also servers a link between the clusters private network and the public or corporate network clients reside in. Interactive steering of the simulation can also be performed through these clients by providing a two-way communication channel. This allows for adjustment of viewpoint and other visualization parameters from the client side.

## 1.2 Related Work

This section provides an overview of the most important work in the field of volume rendering, parallel and remote rendering as well as in situ visualization.

### 1.2.1 Volume Rendering

Marc Levoys paper [Lev88] is one of the principal works in the area of volume rendering. Therein, Levoy describes volume ray casting as a direct volume visualization algorithm, which many following techniques are based upon. Early volume rendering applications [DCH88] implemented solely on the CPU were not able to produce images at interactive rates. With the advent of powerful graphics processing units (GPU) the idea arose to utilize their huge potential for parallel computation. First approaches employed the texturing hardware to parallelize the rendering process [CN94], e.g. texture slicing [CCF94] and shear-warp [LL94].

After the introduction of the programmable rendering pipeline it was possible to use the GPU for other purposes than transforming and rasterizing primitives, namely to implement optimized algorithms such as ray-casting with early ray termination and empty space skipping [KW03]. So called shading languages like HLSL, GLSL and Cg enabled developers to express volume rendering algorithms in a more straight forward manner and abstracted from the cryptic assembler languages used before [RSEB$^+$00, EKE01, KW03].

The next step GPU manufacturers took was to enable programmers to write code for GPUs in a more uniform and familiar way using classical languages like C and Fortran. As that allowed developers to use the GPU for almost arbitrary computation tasks the term GPGPU (general purpose GPU) was coined. The two most important GPGPU APIs nowadays are CUDA by NVIDIA and the open standard

OpenCL. Recent works such as Volt[1] by Jato and Hinkenjann [JH11] of the Bonn-Rhein-Sieg University of Applied Sciences and Exposure Render[2] by Thomas Kroes et al. [KPB12] of the Technical University Delft show that real-time volume rendering of static data sets can be performed with CUDA.

### 1.2.2 Parallel and Remote Rendering

With the introduction of the parallel programming model on CPUs and GPUs simulations running on large distributed clusters produce an ever growing amount of data. Even the strongest workstations are not able to visualize simulation outputs of several tera to petabytes in their full extent. In the same manner simulations had to be parallelized to solve larger problems in shorter time, visualizations are now performed in parallel to allow for rendering of large data sets not fitting into the memory of a single machine.

Molnar et al. [MCEF94] classified parallel rendering as sort-first, sort-middle and sort-last, depending on which kind of primitives are sorted in which stage of the rendering pipeline. A mixture of the sort-first and sort-last approaches has been proposed by Samanta et al. [SFLS00] to form a hybrid sorting scheme. This should allow for better scaling in processor count and screen resolution.

For small to mid-sized data sets Marchesin et al. [MMD08] investigated a multi-GPU architecture on a single machine. Anyhow, this approach suffers from the same memory limitations as single GPU workstations and is not able to handle large scale simulation data sets exceeding several gigabytes.

To overcome this limitation visualization clusters dedicated to render large data sets at interactive frame rates were employed. Fogel et al. [FCS+10] demonstrated that GPU accelerated rendering is a cost-effective and promising approach concerning scalability. So called fat nodes with multiple GPUs seem to be well-suited for building larger systems.

As opposed to the system proposed by Fogel et al. which visualized static data sets Esnard et al. [ERC06] coupled their visualization cluster directly to the simulation machine. This approach is called online visualization or interactive steering as it allows scientists to immediately monitor the results and control the simulation parameters.

Furthermore, existing visualization frameworks such as the Visualization Toolkit (VTK) and ParaView have been used to parallelize data management and rendering. In [MT03] the VTK framework is extended by parallel rendering components. Cedilnik et al. [CGM+06] use ParaView to tackle the problem of visualizing large mostly immovable datasets by utilizing remote rendering. A system consisting of data server, rendering server and client controller is implemented.

To parallelize the rendering in existing OpenGL applications Chromium [HHN+02] was invented. It is an OpenGL implementation by itself, intended for the use on clusters of graphics workstations. By employing a client-server model OpenGL commands can be redirected to peers and executed locally. Multi-machine and multi-monitor scenarios are supported as well as sort-first and sort-last compositing.

A toolkit for scalable parallel rendering was introduced by Eilemann and Pajarola in [EMP09]. The Equalizer Framework leaves the data distribution problem to be solved by the application and solely fo-

[1] http://cg.inf.h-bonn-rhein-sieg.de/?page_id=2700
[2] http://code.google.com/p/exposure-render

cuses on the rendering process. The application encapsulates specific tasks such as culling and rendering while the framework is responsible for the distributed execution, synchronization and final image compositing. To allow for an optimal task decomposition several render modes are supported. In addition to sort-first and sort-last compositing, time-multiplex renders alternating frames in parallel but introduces a small latency. Stereo rendering can also be performed simultaneously for each eye. Lastly, interleaved pixel rendering (which is similar to sort-first) provides very good load balancing but does not scale with dataset size.

In [VBS+11] MapReduce – being a lightweight, scalable, general-purpose parallel data processing framework – is adopted as a visualization system, taking advantage of cloud computing resources for parallel data processing and rendering. Common visualization tasks such as mesh simplification and rendering as well as isosurface extraction are implemented and evaluated. For example, meshes of 30 GBs in size are simplified and rendered with image resolutions up to $80000^2$ pixels. As stated by the authors, MapReduce offers an easy to use and highly scalable foundation for a data processing and visualization system in a public cloud or a local commodity cluster.

### 1.2.3 In Situ Visualization

Making the leap from petascale to exascale computing and beyond sharpens the need for new approaches in data processing and analysis. The aforementioned disproportion between memory bandwidth and processor speed now makes it infeasible to continue saving every bit of information simulations produce and afterwards analyze them as a post-processing step. Thus, the not entirely new idea [McC88] of in situ processing is reintroduced. While the simulation is running and producing all kinds of output data, a fraction of the computational resources is used to perform reduction and analysis tasks. Highly scalable algorithms and a tight coupling of simulation and processing code is key to achieve the desired performance.

In [Ma09] Kwan-Liu Ma argues that in situ processing and especially in situ visualization is a promising approach to utilize exascale simulations to their full potential, extracting meaningful information from the huge amount of data. Figure 1.2 illustrates the differentiation between post processing, co-processing and in situ processing.

Recently there have been some works showing the use of in situ visualization. Whitlock et al. [WFM11] employ VisIt[3], a fully-featured visualization tool, to perform the necessary steps to transform raw data into expressive images. The VisIt architecture provides a client/server model in which clients are responsible for providing an interface users can interact with to steer the parallel server processing and rendering the data. Thus, a simulation library was implemented to resemble a VisIt server clients can connect to. The server is tightly coupled with the simulation sharing data structures to deliver quantities for visualization on demand. Identical VTK data flow networks are created to process the data in a distributed manner, each being executed on another portion of the whole dataset. Eventually an image is delivered to the client.

Making in situ visualizations accessible as a post processing resource was approached by Kageyama et

[3] https://wci.llnl.gov/codes/visit/

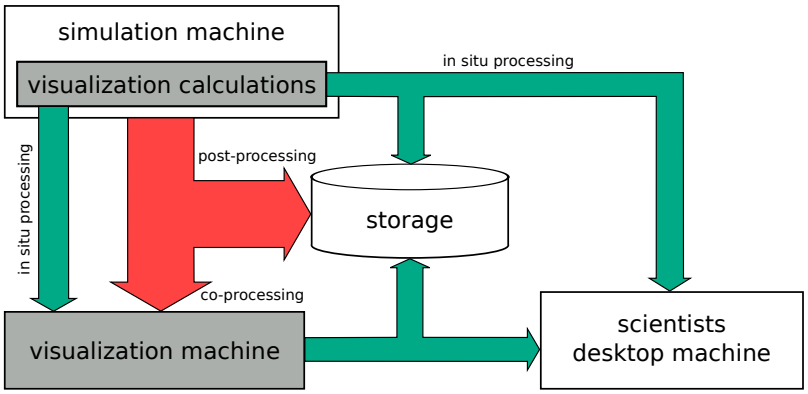



Figure 1.2: Comparison between post-, co- and in situ processing. Both, post- and co-processing involve the transfer of raw data. In situ processing provides the most scalable solution to deal with large scale simulation data (after [YWG+10]).

al. [KY13]. The key idea is to run lots of in situ visualizations at the same time – from thousands to millions – showing different configurations and different views on the data. Thereby an image database is created which can be explored interactively as a post processing step. This is particularly useful for long running simulations where watching the progress is too time consuming.

In [YWG+10] Yu et al. conduct a case study showing how to combine in situ visualization with a petascale combustion simulation. Several challenges had to be addressed, like integrating visualization into the simulation cycle, parallel rendering of particles and volume data as well as image compositing. It was shown that in situ visualization is feasible in terms of integration effort and computational cost and can allow scientists to understand their models, algorithms and data sets as a whole.

In his doctoral thesis Jérôme Soumagne presented a loosely-coupled model for in situ visualization [Sou12]. Distributed and in-memory HDF5 files storing steering, simulation and meta data are used to decouple compute and visualization resources on a supercomputer. The user generates steering data by interacting with a desktop GUI, investigates received images from the visualization and can perform various post-processing tasks.

## 1.3 Structure of this Thesis

This thesis is structured in the following way. In the first chapter we motivated the need for scientific simulations and pointed out current challenges which demand for new approaches in data analysis e.g. through in situ visualization. Related literature, showing the state of available solutions, was reviewed as well.

Chapter 2 will introduce basic methods and concepts which are necessary to understand the choices made in terms of design and implementation of the developed visualization system. Topics covered are the Compute Unified Device Architecture (CUDA) as the underlying software and hardware ecosystem, a short introduction to message passing via MPI and an overview of the instrumented simulation code PIConGPU. After that, the most common volume rendering methods are reviewed, with emphasis on volume ray casting, as this is the technique of our choice for this thesis. Lastly, we will explain how

sub images are combined into a whole. The binary swap algorithm will serve as an example of sort-last image compositing.

Chapter 3 derives the overall system design from the requirements stated in the task description. Limitations such as network bandwidth and missing OpenGL rendering support are considered as well as human factors such as user interface usability. We motivate the decomposition into three main components and describe how they collaborate to form a decoupled yet highly interactive system.

How the core of the system – the parallel volume renderer – works will be set out in Chapter 4. Its integration with the simulation code and rendering capabilities are presented. How we solved the problems arising when a single image should be rendered by multiple GPUs – like correct sampling of sub volumes and fast ordered image composition – is explained.

After introducing the rendering component in the previous chapter we will show how the image producing part of the system is decoupled from the interface users interact with – namely the client program – in Chapter 5. For that reason, a connector and dispatcher – the visualization server – was created. It is responsible for making the link between simulations running on an HPC system and clients connecting from desktop machines or other devices without having them to know about each other directly.

The aforementioned client is the third and last component of our system, described in Chapter 6. In the simplest case it is only responsible for connecting to a simulation registered at the server and displaying received images. More sophisticated implementations can interactively steer the simulation by sending commands to adjust the viewpoint and color scales or change various other parameters.

A performance evaluation is presented in Chapter 7. The characteristics of the rendering component moving through the stages of volume ray casting, image compositing and delivery are investigated. Scalability being the key for large-scale simulations is tested by means of different GPU configurations and simulation sizes. Additionally the overall system performance in terms of interactivity is assessed.

Chapter 8 concludes the thesis by pointing out which possibilities the system offers to scientists, which limitations are still present and where improvements can be made.

Appendix A contains a listing of the message protocol defined for communication among the system components. Appendix B provides a short guide on how to run an interactive in situ visualization.

# 2 Fundamentals

## 2.1 Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) was introduced in November 2006 by NVidia. As of the time of writing this thesis version 5.5 is released. CUDA is a highly parallel computing platform and programming model, which leverages the massive floating-point capacities of graphics processing units (GPU). Formed by the combination of CUDA-enabled devices of the GeForce, Quadro and Tesla series and accompanied by a software development kit (SDK) including CUDA libraries (e.g. cuBLAS, cuFFT, Thrust), compiler (NVCC) and tools (profiler and debugger) it offers a general-purpose API to program multiprocessors.

As opposed to common graphics programming APIs like OpenGL and Direct3D, which require GPU code to be written in a specific shading language (GLSL, HLSL), CUDA extends well-known programming languages such as C/C++ and Fortran. Thus the aggravation of mapping scientific problems to computer graphics is eliminated, making it much more convenient for developers (which are mostly scientists) to utilize the computational horsepower of their hardware and concentrate on the problem at hand.

This shift in the notion of the GPU from a graphics co-processor to a streaming multiprocessor which can execute arbitrary computational tasks led to the term GPGPU (for general-purpose graphics processing unit). The most obvious indication that GPGPU devices are no longer graphics processors is the lack of display connectors (see Figure 2.1).


Figure 2.1: NVidia Tesla K20

The code executed on a GPU is called a *kernel*. That a procedure is a kernel is indicated by the `__global__` declaration specifier, which is one of the CUDA C language extensions. A kernel is executed on a grid of thread-blocks by calling it with the new <<<...>>> *execution configuration* syn-

tax. The grid consists of multiple thread-blocks which in turn are made up of several threads as depicted in figure 2.2. All threads in a grid execute the same kernel code, which is also referred to as *device code* whereas code executed on the CPU is termed *host code*.

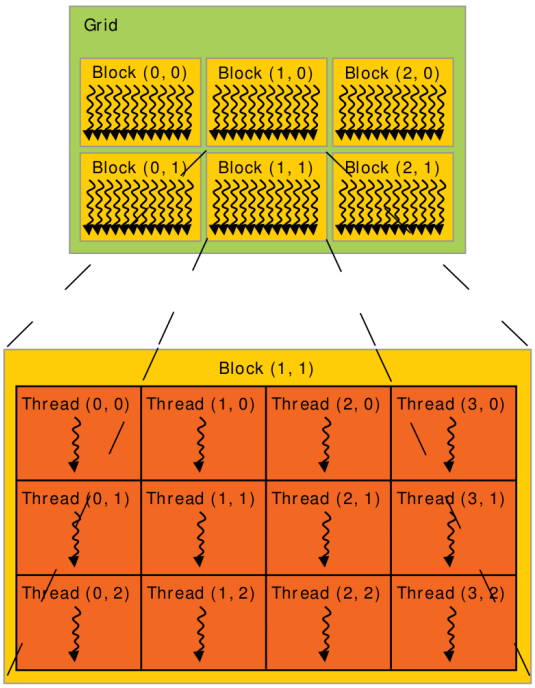



Figure 2.2: Grid of Thread-blocks (from [NVi13])

Each thread possess its own set of registers which are located *on-chip* and offer very fast reads and writes. All threads within a block collectively own a region of so called *shared memory*. As this memory is also located on-chip it exposes a very low access latency but is limited to 48 KB in size. The lifetime of registers and shared memory is bound to the runtime of a kernel execution.

The largest memory on the GPU is the global device memory. The current high-end GPU Tesla K20X offers 6 GB of GDDR5 RAM. It is the only memory accessible from host code. Data located in global memory persists between kernel calls throughout the whole application runtime and can be accessed by all threads. Being implemented as *off-chip* memory it has a much higher latency and lower bandwidth than on-chip shared memory.

The aforementioned memory types all allow read and write access. Besides them two more memory regions namely texture and constant memory exist. These are also located in the global device memory but provide read-only access. This limitation allows for optimization by caching and can shorten access latencies significantly. Textures are particularly well-suited for memory fetches which expose a high spatial locality as their cache is other than the L1 cache of global memory optimized for 2D local access. Figure 2.3 illustrates the CUDA memory hierarchy.

Memory and thread hierarchy together lead to a problem decomposition in which sub-problems can be solved by a group of threads cooperatively in a block and individual threads work on single data elements. To help identify a thread in a grid and address the data elements it is should process the built-in variables **threadIdx**, **blockIdx**, **blockDim** and **gridDim** can be used.

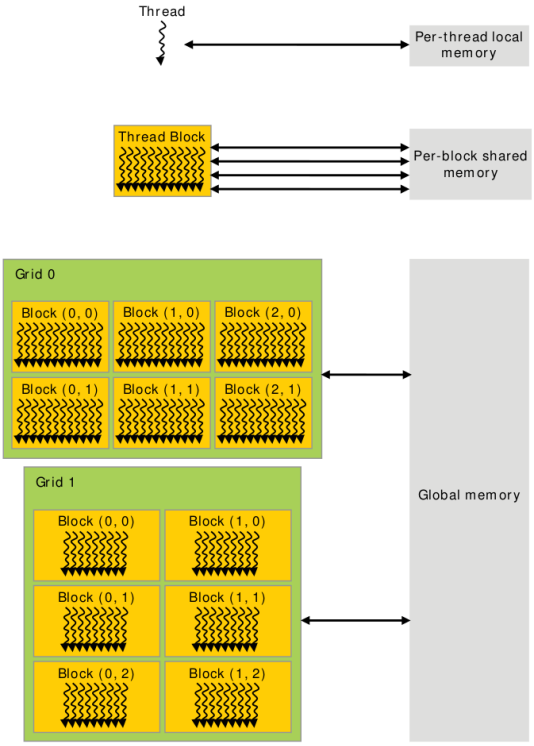



Figure 2.3: CUDA Memory Hierarchy (from [NVi13])

As it is often necessary to join the partial solutions obtained by each thread (called a *reduction*) and especially when collectively working on shared memory, execution has to be synchronized at certain points. For that purpose the intrinsic function `__syncthreads()` may be used to block the execution until all threads of a block have reached the barrier. This is not necessary if all the threads which have to be synchronized belong to the same *warp*. A warp is formed by 32 consecutively indexed threads of the same block. All threads within the same warp operate in *lockstep* mode which means that they always execute the same instruction. By exploiting this fact some synchronization overhead can be saved.

## 2.2 Message Passing Interface

Message Passing is a communication paradigm in concurrent, parallel and object-oriented programming as well as interprocess communication. A process can send messages to and receive messages from one or more other processes. Such a message can be of arbitrary size (i.e. zero or more bytes), representing a simple token or constituting a complex data structure. Synchronization among processes can be achieved by waiting for messages [Wik13].

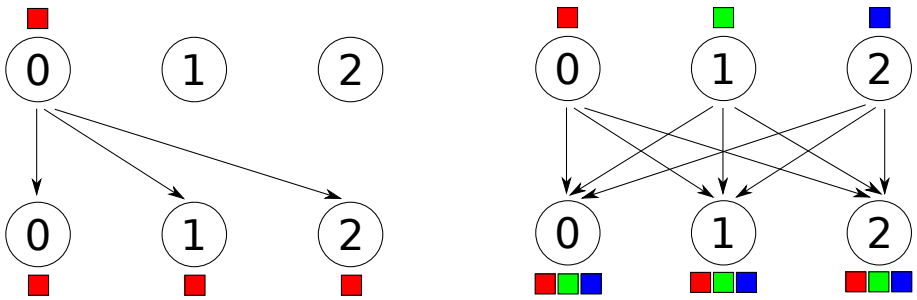
The Message Passing Interface (MPI) is a *message-passing library interface specification* and no library by itself. That means it defines abstract data types and procedure signatures which constitute the programming interface. Language bindings for Fortran, C and C++ are specified. The details of how a certain functionality is implemented is left to the concrete library (e.g. OpenMPI and MPICH2). MPI being an established industry standard is defined by a broad committee of vendors, researchers, implementors and users. Version 3.0 was released in September of 2012 although version 2.2 is due to compiler

and implementation support the most recent standard in practical use.

As opposed to remote procedure calls or remote method invocation in which *procedures* of processes in different address spaces are called, MPI primarily addresses the message passing parallel programming model, in which *data* is moved from the address space of one process to the address space of another through cooperative operations on each of the processes [For09].

The MPI specification attempts to be practical, portable, efficient and flexible. Originally it was designed for distributed memory systems, but supports shared memory and hybrid systems too.

MPI specifies operations for point-to-point as well as collective communication. A central concept is that of a *communicator*. A communicator defines a communication context by specifying which collection of processes may exchange messages. The predefined communicator `MPI_COMM_WORLD` includes all processes launched by an `mpirun` respectively `mpiexec` command. The number of processes within a communicator is referred to as its *size* and can be obtained by calling `MPI_Comm_size` with the communicator as first argument. Each process has a unique integer identifier inside a communicator called *rank*. Ranks are contiguous and start at zero. The rank of a process inside a communicator can be retrieved by calling `MPI_Comm_rank`. They can be used to identify sender and receiver in a simple point-to-point message transfer or the root process in a collective operation such as a message broadcast or reduction. Figure 2.4 illustrates two collective MPI operations which are important for this thesis. A *broadcast* sends the same data from the root process to all other processes within the specified communicator. By executing an *allgather* every process inside a communicator sends its own data to all other processes and receives the data of all other processes. Resulting in a data array of a size equal to that of the communicator, consisting of data elements ordered by process rank including each process' own data element.




(a) MPI_Bcast (b) MPI_Allgather

Figure 2.4: MPI procedures for collective communication

When writing a parallel MPI application the single program paradigm is usually employed, that means all processes run the same binary. Hence, the rank is used to control conditional program execution, e.g. to follow different code paths in sender and receiver processes. A simple example is given in Listing 2.1 to conclude this MPI introduction. Therein MPI is initialized, an integer number is sent from the process with rank 0 to the process with rank 1. After displaying rank and data MPI is shutdown and the program quit.

Listing 2.1: A simple MPI program.

```
1  int main(int argc, char ** argv)
2  {
3          MPI_Initialize(&argc, &argv);
4
5          int my_rank = MPI_Comm_rank(MPI_COMM_WORLD);
6          int send_receive_buffer;
7
8          if (my_rank == 0)
9          {
10                 send_receive_buffer = 42;
11                 MPI_Send(&send_receive_buffer, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
12         }
13         else if (my_rank == 1)
14         {
15                 MPI_Recv(&send_receive_buffer, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, nullptr);
16         }
17
18         std::cout << "My rank is: " << my_rank << " got " << send_receive_buffer << std::endl;
19
20         MPI_Finalize();
21 }
```

## 2.3 Simulation Code PIConGPU

PIConGPU is an acronym for Particle-in-Cell on Graphics Processing Units. Originating as a *Jugend forscht* project by Heiko Burau in 2008[1], it is now developed and maintained by the *Junior Group Computational Radiation Physics* at the *Institute for Radiation Physics* at HZDR. In August 2013 an open source version was released for public use[2].

PIConGPU is a fully-relativistic numerical simulation code based on the Particle-in-Cell algorithm. It allows researchers to investigate the dynamics of plasma physics without conducting real experiments. “The PIC algorithm solves the so-called Maxwell-Vlasov equation. To solve this equation, electric and magnetic fields are interpolated on a physical grid dividing the simulated volume into cells. Charged particles such as electrons and ions are modeled by macro-particles with a mass $m$ and a charge $q$. These can describe the motion of up to several hundred particles by the motion of a single spread-out particle distribution. The macro-particles motion is influenced by the electric and magnetic fields on the grid.”[Bus13] Following the Maxwell equations (see Table 2.1) and the Lorentz force (see Equation 2.1) the particle motion results in a current $J$. This current induces the magnetic field $B$ which in turn induces the electric field $E$.

$$\vec{F}_{Lorentz} = q(\vec{E} + \vec{v} \times \vec{B}) \tag{2.1}$$

The basic algorithm has four steps depicted in Figure 2.5.

PIConGPU is now one of the fastest PIC implementations in plasma physics. Depending on the chosen

[1] http://www.jugend-forscht.de/projektdatenbank/rechenwunder-grafikkarte/burau.html
[2] https://github.com/ComputationalRadiationPhysics/picongpu

| | |
|---|---|
| Gauss (electric) | $\nabla \cdot \vec{E} = \frac{\rho}{\epsilon_0}$ |
| Gauss (magnetic) | $\nabla \cdot \vec{B} = 0$ |
| Faraday | $\frac{\delta \vec{B}}{\delta t} = -\nabla \times \vec{E}$ |
| Ampere | $\nabla \times \vec{B} = \mu_0 \vec{J} + \frac{1}{c^2} \frac{\delta \vec{E}}{\delta t}$ |

Table 2.1: Maxwell Equations

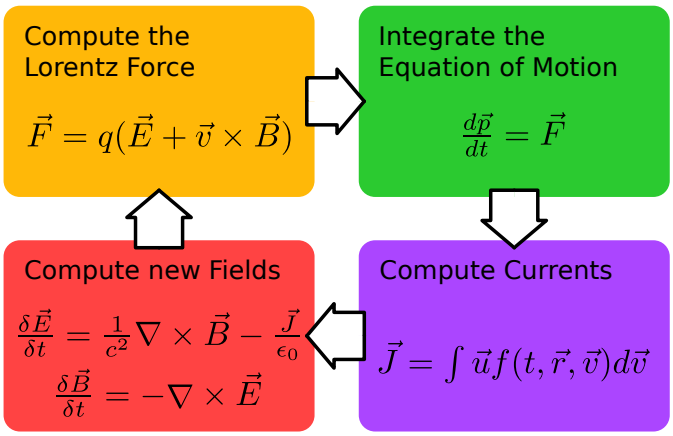



Figure 2.5: The PIConGPU simulation cycle.

parameter set, the utilized solvers and GPU configuration, several time steps can be computed per second (e.g. using the Villasenor-Buneman solver and a simulation grid size of $128 \times 1024 \times 128$ cells yields around 20 time steps per second on 64 GPUs). The key to its speed and scalability is the hybrid parallel architecture of the code. Using CUDA on the thread-level fully utilizes a single GPUs parallel computation power. To make the original single-GPU version multi-GPU enabled and scalable to a cluster of GPU nodes, MPI is employed on the process-level (see Figure 2.6).

All the heavy load is on the GPU, while the CPU handles data exchange among processes. As the memory of a single GPU is too small to simulate large physical problems, the simulation domain is decomposed into sub domains. Hence, each GPU is responsible for simulating a sub volume of the whole simulation grid and exchanging data with other GPUs. This communication is required because the schemes used to interpolate fields need data from adjacent cells. Also, particles can move from one sub volume to another. The exchanged data areas are called borders or guarding and extend the local simulation grid by cells which do not belong to the local simulation grid and are thus not computed locally (see Figure 2.7). The data held by them needs to be updated after a time step was completed and is copied from neighboring GPUs.

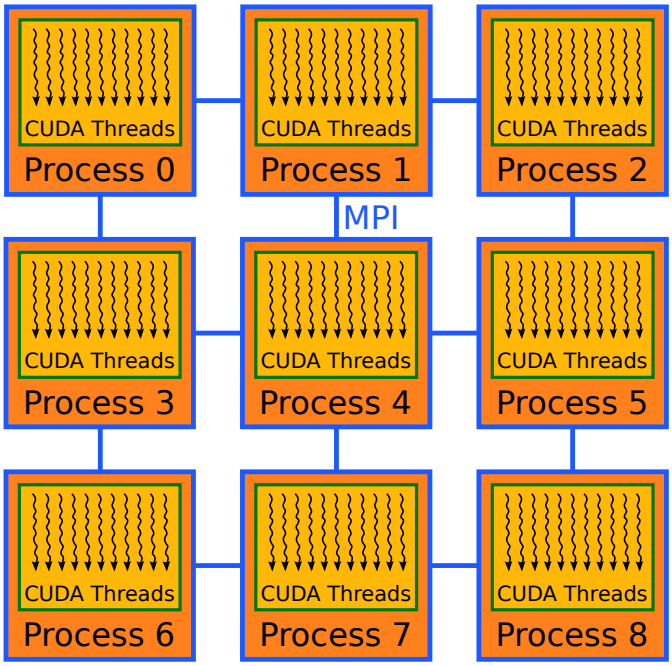



Figure 2.6: Hybrid parallelism of CUDA threads and MPI processes

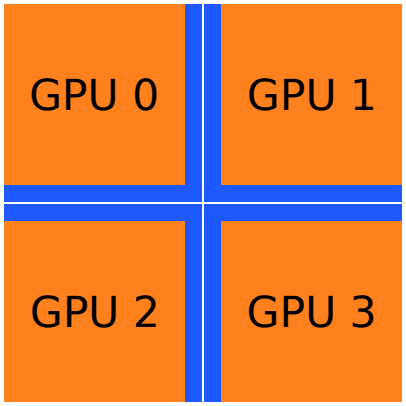



Figure 2.7: GPU Subdomains with Core (orange) and Border (blue) area

As the data transfer between nodes can be a bottleneck slowing down computation by leaving the GPU waiting for data to arrive, the code interleaves inter-process communication and computation. For example, the electric field in the border area can be computed first and sent to neighboring processes. During the transmission of that data the field in the core area can be calculated. [BWH+10]

PIConGPU offers a plug-in interface to enable developers to extend the code by custom modules (formerly called analyzers). Plug-ins have to implement the interface shown in Listing 2.2.

Listing 2.2: PIConGPU Plug-in Interface

```
class Module
{
protected:
        virtual void moduleLoad() = 0;
        virtual void moduleUnload() = 0;
};

class IPluginModule : public Module
{
public:
        virtual void setMappingDescription(MappingDesc * cellDescription) = 0;
};

class ISimulationIO
{
public:
        virtual void notify(uint32_t currentStep) = 0;
};
```

When the simulation is initialized the `moduleLoad` method is called. Here the plug-in should allocate required resources and perform its own initialization. The `moduleUnload` method is called when the simulation is shut down. The requested resources should be freed herein. If the plug-in needs access to simulation data it has to be derived from `ISimulationIO` and register itself at the `DataConnector`. This singleton class holds the simulation data of the current time step. The plug-ins `notify` method is invoked periodically every n-th time step, depending on the `notifyFrequency` specified in the `registerObserver` call. This mechanism is known as the Observer pattern [GHJV95]. The relevant interface of `DataConnector` is shown in Listing 2.3.

Listing 2.3: DataConnector Interface

```
class DataConnector
{
public:
        static DataConnector& getInstance();

        void registerObserver(ISimulationIO * observer, uint32_t notifyFrequency);

        template<class TYPE>
        TYPE& getData(uint32_t id, bool noSync = false);
};
```

To run a simulation one has to specify a configuration file containing simulation parameters as command-line arguments. These parameters determine the number of time steps computed and the GPU configuration (e.g. two GPUs in each direction resulting in the simulation area distributed over eight GPUs). Some of these parameters are prefixed with the name of a plug-in. In that manner, the initiator of the

simulation can utilize certain plug-in modules and parameterize them. An example configuration file is shown in Listing 2.4.

Listing 2.4: A PIConGPU configuration file.

```
1  TBG_wallTime="24:00:00"
2
3  TBG_gpu_x=4
4  TBG_gpu_y=4
5  TBG_gpu_z=4
6
7  TBG_gridSize="-g 128 2048 128"
8  TBG_steps="-s 10000"
9  TBG_movingWindow="-m"
10 TBG_devices="-d !TBG_gpu_x !TBG_gpu_y !TBG_gpu_z"
11
12 TBG_programParams="!TBG_devices     \
13              !TBG_gridSize    \
14              !TBG_steps       \
15              !TBG_movingWindow \
16              !TBG_analyser  | tee output"
17
18 TBG_analyser="!TBG_hdf5"
19
20 # hdf5 output
21 TBG_hdf5="--hdf5.period 100 --hdf5.compression"
22
23 # TOTAL number of GPUs
24 TBG_tasks="$(( TBG_gpu_x * TBG_gpu_y * TBG_gpu_z ))"
25
26 "$TBG_cfgPath"/submitAction.sh
```

A simulation with 64 GPUs covering a $128 \times 2048 \times 128$ grid is configured. Ten thousand time steps should be computed. A plug-in for writing compressed HDF5 files of the simulation data every 100th step is utilized (see line 21). The $-\mathrm{m}$ argument on line 9 is inserted on line 15 and enables a sliding simulation window. Because investigation of some physical phenomena may require a lengthy simulation grid but expose features worth observing only in a small frame of that area, it is possible to reorganize the GPU grid layout during the simulation run. Only a part of the whole grid is computed inside the window. If the observed feature reaches the edge of the window, GPUs from the opposing edge are detached from the "back" of the grid, reinitialized, and attached to the "front". Thereby, simulations like *Laser Wakefield Acceleration* can be run by fewer GPUs over more time steps. Figure 2.8 illustrates the mechanism.

The PIConGPU code is able to simulate a variety of plasma-physical phenomena. Exemplary, we will describe three experiments: *Laser-Wakefield Acceleration*, the *Weibel* and the *Kelvin-Helmholtz Instability*.

**Laser-Wakefield Acceleration** In Laser Wakefield-Acceleration (LWFA) an ultra-short high energy laser pulse is shot into a plasma (a low-density gas in which individual atoms are ionized). "As the light pulse travels through the plasma, its electric field causes the electrons and the atomic nuclei within the plasma to separate. Macroscopically it is as if a *bubble* of charge is moving through the plasma at nearly the speed of light, and the pulse leaves in its wake a small area of very strong electric field. This wakefield can be used to accelerate charged particles to very high energies over very short distances."

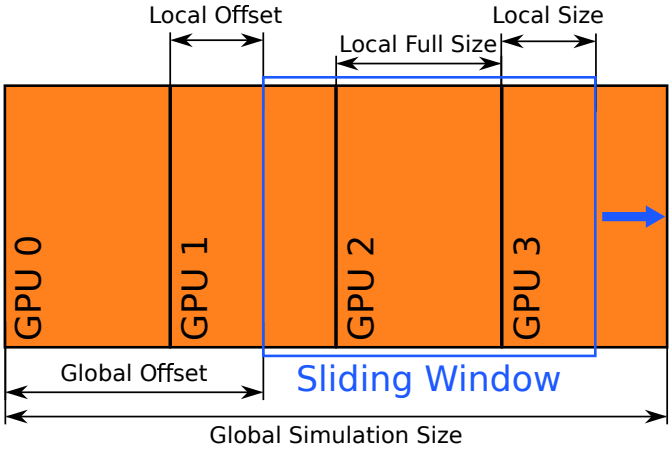



Figure 2.8: The simulation grid is distributed on four GPUs. The observed area framed in blue has the size of two GPUs but spans three. When the window hits the end of GPU three, number one will be reinitialized and attached to the end of the simulation area.

[Taj08] Laser pulses of 35 fs at 50 TW in power are typical [Taj08]. Figure 2.9 illustrates the principle of laser-driven particle acceleration.

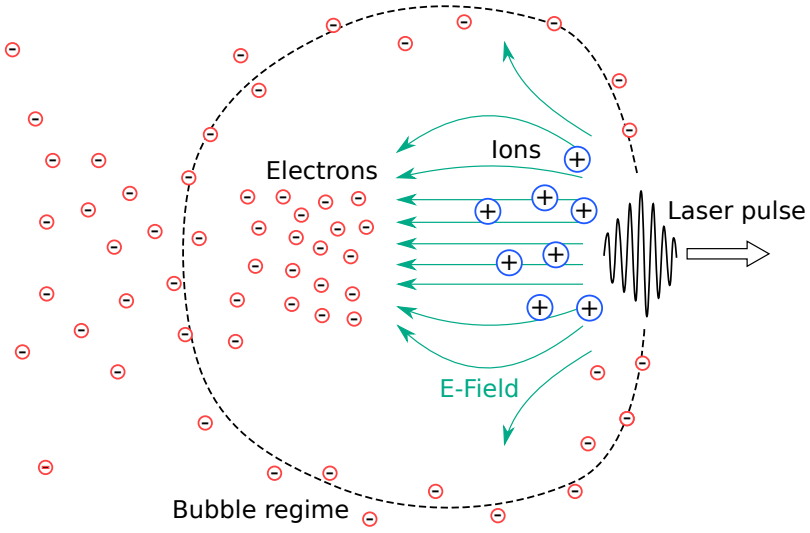



Figure 2.9: Development of a bubble regime by separation of electrons and ions after the Laser-Wakefield Acceleration principle [Züh11].

"Obtaining a high-quality electron beam using the LWFA mechanism requires careful optimization of various laser and plasma parameters, including the laser focal-spot size, plasma density, interaction length and laser pulse duration." [Taj08] With this technology very compact accelerators could deliver brilliant electron and X-Ray beams for medical application such as radiation and hadron therapy [TD79].

**Weibel Instability** The Weibel Instability [Wei59] is a plasma instability that appears in (nearly) homogeneous plasmas. It is caused by an anisotropy in the velocity space of the plasma particles which can be understood as two temperatures in different directions. Fried [Fri59] showed that the instability can also be understood as the superposition of many counter-streaming beams. It can be observed in laboratory as well as astrophysical plasmas [LSWP09].

**Kelvin-Helmholtz Instability** The Kelvin-Helmholtz Instability [Tho71] can occur in a continuous fluid if a velocity shear or on the interface of two fluids if a velocity difference between them is present. It can be observed in the sun's corona, Jupiter's Red Spot and Saturn's band[3] (see Figure 2.10)[4].

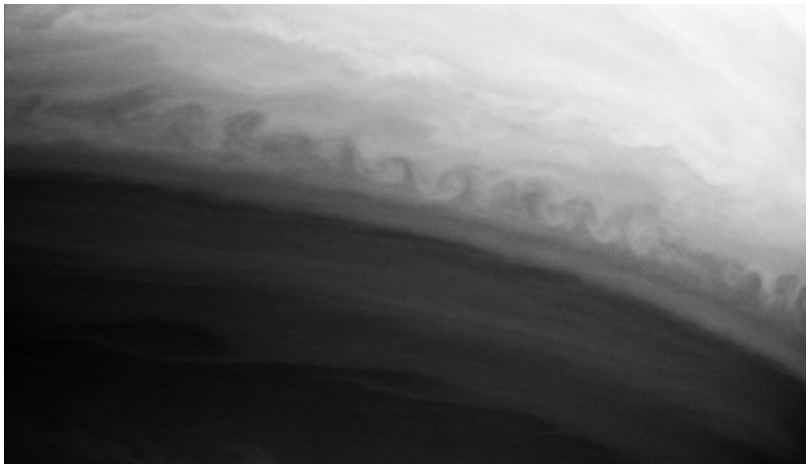


Figure 2.10: An example of the Kelvin-Helmholtz Instability in Saturn's atmosphere. At the boundary of two latitudinal bands turbulences curl repeatedly. Photo taken with Cassini spacecraft narrow image camera.

## 2.4 Volume Rendering

In this section the basics of (direct) volume rendering are recapped as necessary for the understanding of this thesis. Large parts of this chapter are derived from the book *Real-Time Volume Graphics* of Hadwiger et al. [HKRs+06]. More detailed explanations on the theory of volume rendering and practical considerations for real-time volume graphics using graphics hardware can be found there.

Volume rendering is a method to visualize 3D data sets, which typically originate from simulation (as in our case) or measurement, such as geological data and medical data form CT or MRI scans. The discretized data set is usually stored on some kind of structured or unstructured grid. In our case a uniform 3D mesh stores the vector and scalar values of electromagnetic fields and other quantities (like particle or energy density). The individual cells of the mesh are not necessarily cubic but have a constant width, height and depth[5] so that the cells are ashlar-formed. Figure 2.11 illustrates the notion of voxel and cell.

Volume rendering tries to visually extract information from a given 3D scalar field. As a scalar field is a mapping

$$\phi : \mathbb{R}^3 \to \mathbb{R}, \tag{2.2}$$

vector quantities such as the electric and magnetic field force need to be transformed into a scalar value. This can be done by extracting a single component of the field vector or calculating the intensity of the field.

[3] www.nasa.gov/mission_pages/sunearth/news/sun-surfing.html
[4] photojournal.jpl.nasa.gov/catalog/PIA06502
[5] In the code they are identified by `CELL_WIDTH`, `CELL_HEIGHT` and `CELL_DEPTH`

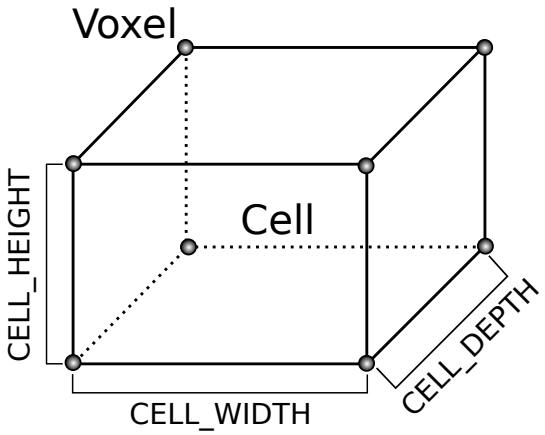



Figure 2.11: A cell formed by eight voxels.

To visualize the scalar field data the simulation volume is treated as to be filled with a gaseous material like fog or plasma. Then, the scalar values are mapped to quantities that model light interaction. This process is called classification and can be accomplished by using a transfer function which maps a scalar to a tuple of color and opacity. A physically-based model of light transport through and within the volume is employed to compute the radiation perceived by a virtual observer.

The major interactions of light with the material are modeled by:

**Emission** The material emits light by itself.

**Absorption** The material absorbs radiation it interacts with.

**Scattering** The direction of light is changed by the material. Incoming light is either scattered towards the observer (*in-scattering*) or away from the observer (*out-scattering*).

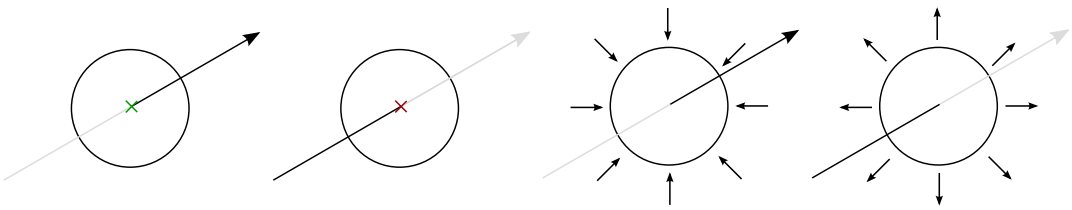


Figure 2.12: Light interactions from left to right: emission, absorption, in-scattering and out-scattering (after [HKRs+06])

Taking different combinations of these interactions into account when computing the radiation perceived by the observer leads to various optical models. Those are

**Absorption only**

Incident light is absorbed by the material, no light is scattered or emitted.

**Emission only**

The volume consists of transparent material emitting light, there is no absorption or scattering.

**Emission-Absorption**

This is the most common model. Light is emitted and absorbed by the material, but scattering and indirect illumination are neglected.

### Single-Scattering

This model extends the emission-absorption model by considering single scattering of light coming from an external light source.

### Multiple-Scattering

The complete spectrum of light interactions is taken into account including emission, absorption and scattering. This method exposes the highest computational intensity.

To determine the amount of light (emitted, absorbed or scattered) that reaches the observer the optical model provides a formula to calculate the radiation in all directions $\omega$ at a point $\mathbf{x}$. In this thesis we will employ the emission-absorption model which is described by the equation

$$\omega \cdot \nabla_x I(\mathbf{x}, \omega) = -\kappa(\mathbf{x}, \omega) I(\mathbf{x}, \omega) + q(\mathbf{x}, \omega). \tag{2.3}$$

This formula is referred to as the *volume rendering equation* in its differential form. The function $\kappa$ describes the absorption of light while the function $q$ determines the emission of light at a given point in a given direction.

To compute the radiation reaching the observer only a single direction has to be considered, thus, only the light along a single ray from a given point $\mathbf{x}$ in the volume towards the observer position has to be examined. For that single ray Equation 2.3 can be written as

$$\frac{dI(s)}{ds} = -\kappa(s) I(s) + q(s). \tag{2.4}$$

The length parameter $s$ describes positions on the ray.

By integrating along the direction of light flow the equation is solved, leading to the *volume rendering integral*

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)dt} ds. \tag{2.5}$$

Here the term $I_0$ stands for the light entering the volume from the background at the position $s = s_0$. $I(D)$ is the light leaving the volume at position $s = D$ reaching the observer.

Typically this integral cannot be evaluated analytically. Numerical methods are utilized to approximate the solution as close as possible. By splitting the integral into $n$ integration intervals a discretization is achieved. Then, the radiation $I(D)$ is computed by

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} (1 - \alpha_j). \tag{2.6}$$

Here, $c$ refers to an RGB color tuple and $\alpha$ to the opacity obtained by applying the transfer function to the sampled scalar value.

The summations and multiplications can be split into even simpler operations leading to an iterative

computation of the volume rendering integral. Its basis are so-called *compositing* schemes. Common schemes are **front-to-back** and **back-to-front** compositing. The front-to-back scheme is used when viewing rays are traversed from the observer position through the volume, as in our case. The associated iterative equations are

$$C_i = C_{i+1} + (1 - \alpha_{i+1})C_i,$$
$$\alpha_i = (1 - \alpha_{i+1})\alpha_i.$$

Renaming the variables and reindexing yields the equations in their most common form

$$C_{out} \leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \tag{2.7}$$

and

$$\alpha_{out} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src}. \tag{2.8}$$

Note that the compositing equations presented assume *associated colors* (also called *pre-multiplied colors*) as introduced by Blinn [Bli94], i.e. the RGB components of a color $C$ are already weighted by their corresponding opacity $\alpha$.

Alternative compositing schemes which are relevant for this thesis are *Maximum Intensity Projection*, which is computed by the compositing equation

$$C_{out} \leftarrow \max(C_{dst}, C_{src}) \tag{2.9}$$

and the *First* scheme which extracts iso-surfaces from the volume. Equation 2.10 shows how the compositing for this scheme can be computed ($v_s$ represents the sampled scalar value, $v_{iso}$ the desired iso surface value).

$$C_{out} = \begin{cases} C_{dst} & \text{if } v_s \neq v_{iso} \\ C_{src} & \text{if } v_s = v_{iso} \end{cases} \tag{2.10}$$

Figure 2.13 summarizes the three compositing schemes by relating the traversed volume depth to the sampled intensity which is later mapped to a color and opacity to eventually be projected on screen.

Various approaches have been suggested to compute the optical models introduced earlier in this chapter. All of them share some common components which are executed in a sequential order. This basic algorithm structure is referred to as the volume rendering pipeline and consists of the following stages:

### Data Traversal

Sampling positions are chosen inside the volume depending on the position and orientation of slices or sampling interval. The samples are needed to find a discrete approximation of the continuous volume rendering integral.

#### Interpolation

As the sampling points are usually different from the grid points, a continuous 3D scalar field has to be reconstructed from the discrete mesh forming the 3D volume to obtain the data values at the sampling points. Commonly used reconstruction filters are nearest neighbor and trilinear interpolation.

#### Gradient Computation

For computing the illumination most lighting models require the normal vector to be defined at any given point on a surface. In scalar fields the normal vector is approximated by the local gradient as it provides an estimate of surface boundaries inside the volume. Filters such as central differences can be used to estimate that gradient.

#### Classification

The classification stage maps the sampled and filtered scalar values to the optical properties color and opacity. Therefore a transfer function is usually employed.

#### Shading and Illumination

Volume shading can be added by applying a lighting model like Blinn-Phong and incorporating the obtained illumination term into the emission term of the volume rendering integral. This may give an additional cue about the spatial structure of the volume data.

#### Compositing

The final stage is the compositing of the (shaded) fragments. It is the basis of the iterative solution of the discretized volume rendering integral (see Equation 2.6).

The algorithms can be sorted into the two categories *object-order* and *image-order*. Object-order algorithms approach the problem from the side of the volume data traversing the 3D cells and projecting them to the screen. Image-order algorithms start from the 2D image plane and traverse the volume with a pixel as starting point.

The most popular volume rendering techniques are *ray-casting*, *texture-slicing*, *shear-warp*, *splatting* and *cell-projection*. A short overview shall be given here.

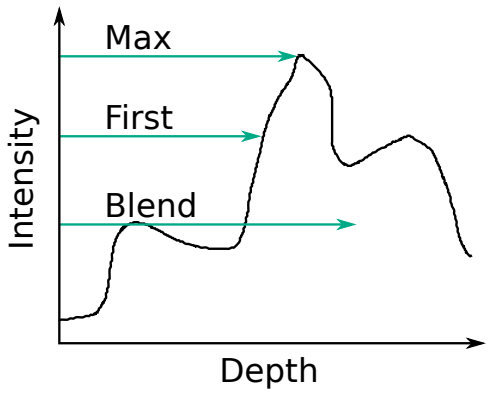



Figure 2.13: Compositing schemes Maximum Intensity Projection, Iso-Surface and Alpha Blending.

**Ray-Casting** In volume ray-casting a viewing ray is traversed for each pixel of the image plane from the observer through the volume, making it an image-order algorithm. The volume rendering integral is evaluated directly along the ray by resampling the volume data at discrete locations determined by the entry point of the ray into the volume and a (constant) sampling interval. For perspective projections the observer position is the origin of the rays whereas the image plane pixel position is the origin for rays in orthographic projections. Thus, front-to-back compositing is the natural compositing scheme used. Ray-casting is the most important method for CPU volume rendering. GPU ray-casting is a much younger development because early GPUs did not support the functionality required for this technique. But with ray-casting as an embarrassingly parallel problem and GPUs being powerful highly-parallel processors - and lastly gaining much more flexibility in their programmability (i.e. 3D texturing hardware, loops in shader programs, later CUDA/OpenCL and GPGPU) - the step towards GPU ray-casting was quite natural. Therefore GPU ray-casting has become the state of the art and will play an important role in the future. This technique is also the one of our choice for this thesis. Chapter 4 will provide a detailed description of how volume ray-casting was implemented in CUDA and MPI on a cluster of GPU nodes. Figure 2.14 illustrates the basic idea. Compared to other volume rendering approaches, ray-casting is relatively easy to implement, projection-independent and flexible in terms of parameterization (e.g. choosing different image resolutions and sampling intervals). It offers possibilities for optimizations like empty space skipping or adaptive sampling and does not require special purpose hardware (i.e. texturing hardware).

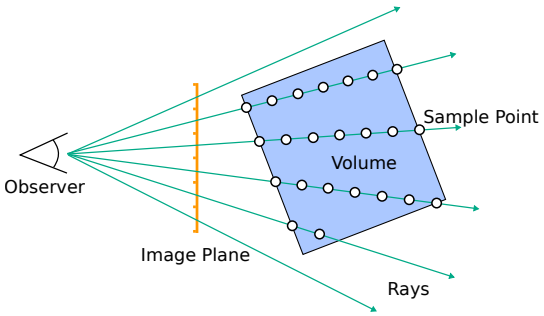
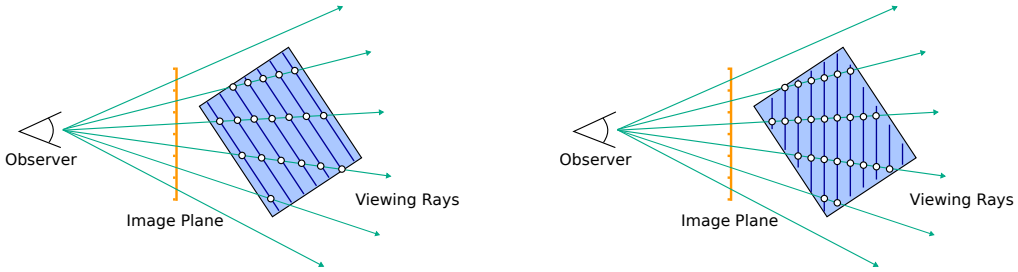



Figure 2.14: Ray-casting (after [HKRs+06])

**Texture-Slicing** In this approach 2D slices are positioned inside the 3D volume sampling the data. This makes texture-slicing an object-order algorithm. These slices are projected onto the image plane and combined according to the compositing scheme. By using the texture hardware and built-in blending operations texture-slicing is directly supported by GPUs, including older non-programmable generations and can be implemented in a very efficient way. Anyhow, it is limited to uniform grids and requires texture hardware support. Figure 2.15 depicts two possible slicing approaches.

**Shear-Warp** Shear-warp is very similar to texture-slicing and therefore also an object-order algorithm. But instead of projecting the slices on the image plane directly, they are first projected onto an intermediate image plane, called the *base plane*. The base plane is aligned with the volume, which is sheared in order to turn the projection of the 2D slices from an oblique to a perpendicular projection direction. Thus, only the image generated on the base plane needs to be warped in contrast to the




(a) Object-aligned slices (b) Image-aligned slices

Figure 2.15: Texture Slicing for perspective projection

texture-slicing method, where each slice needs to be warped. Figure 2.16 shows the basic idea.

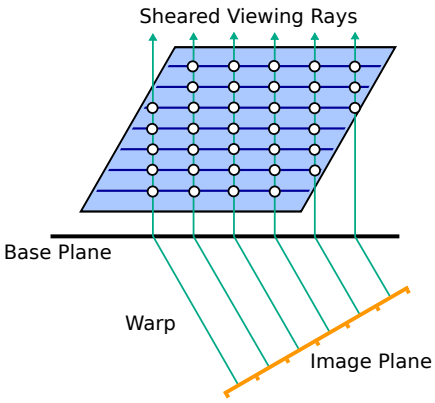



Figure 2.16: Shear-warp volume rendering.

**Splatting** Splatting as proposed by Westover [Wes91] is an object-order approach in which 3D reconstruction kernels resample the volume and are projected on the image plane. These kernels are also called footprints. The way the volume data (i.e. the voxels) is traversed can be chosen quite freely. The only restriction is that some spatial sorting is provided enabling the application of front-to-back or back-to-front compositing.

**Cell-Projection** "Cell projection is an object-order approach for the volume rendering of tetrahedral grids or even more complex unstructured meshes. The first cell projection algorithm that made efficient use of graphics hardware is the projected tetrahedra (PT) algorithm by Shirley and Tuchman [ST90]. The basic idea of the PT algorithm is to traverse the cells of the unstructured grid and project these cells onto the image plane. The projection itself leads to a collection of triangles that represent the image of the 3D cell on the image plane." [HKRs+06]

## 2.5 Image Compositing

A central issue in parallel rendering is how one should combine the subimages created by individual processes into one full image of the visualized data set. We introduced the notions of sort-first and

sort-last compositing earlier (see Section 1.2.2). As opposed to image compositing with the sort-first approach, where the image tiles are simply put together without overlap, sort-last compositing requires more sophisticated algorithms. Their goal is to utilize all available processing resources equally and minimize communication to keep the compositing time as short as possible. According to Peterka et al. [PGR+09] image compositing algorithms can be organized into the categories *direct-send* [EP07], *tree* and *parallel pipeline*.

"In direct-send, each process requests the subimages from all of those processes that have something to contribute to it [Hsu93, MI97, Neu94]. Rather than sending individual point-to-point messages, tree methods exchange data between pairs of processes, building more complete subimages at each level of the compositing tree. To improve load balance by keeping more processes busy at higher levels on the tree, Ma et al. [MPHK94] introduced binary swap, a distance doubling and vector halving algorithm. Recently, Yu et al. [YWM08] extended binary swap compositing to nonpower-of-two numbers of processors in 2-3 swap compositing. Pipeline methods are also published for image compositing, but their use is infrequent. Lee et al. [LRN96] discuss a parallel pipeline compositing algorithm for polygon rendering." [PGR+09] A more flexible algorithm called *Radix-k* was suggested by Peterka et al. [PGR+09]. It employs changing sizes of process groups which collectively compose a subimage in each compositing round until the final image is formed. These group sizes are called radices. By selecting certain radices other composting algorithms can be modeled. Kendall et al. [KPH+10] improved the Radix-k method by incorporating image encoding and runtime k-value adjustment.

To give an example of how these algorithms work, the *Binary-Swap* method for a group of four processes is illustrated in Figure 2.17.

First, each process renders it local data, splits its own partial image into two halves and sends one half to its direct neighbor. Then, each process composites the two halves into one half image. Now each process splits this image again and sends one half of it to its next but one neighbor. A quarter image is composited by each of the processes. Finally, the full image is assembled by one (or all) process(es).

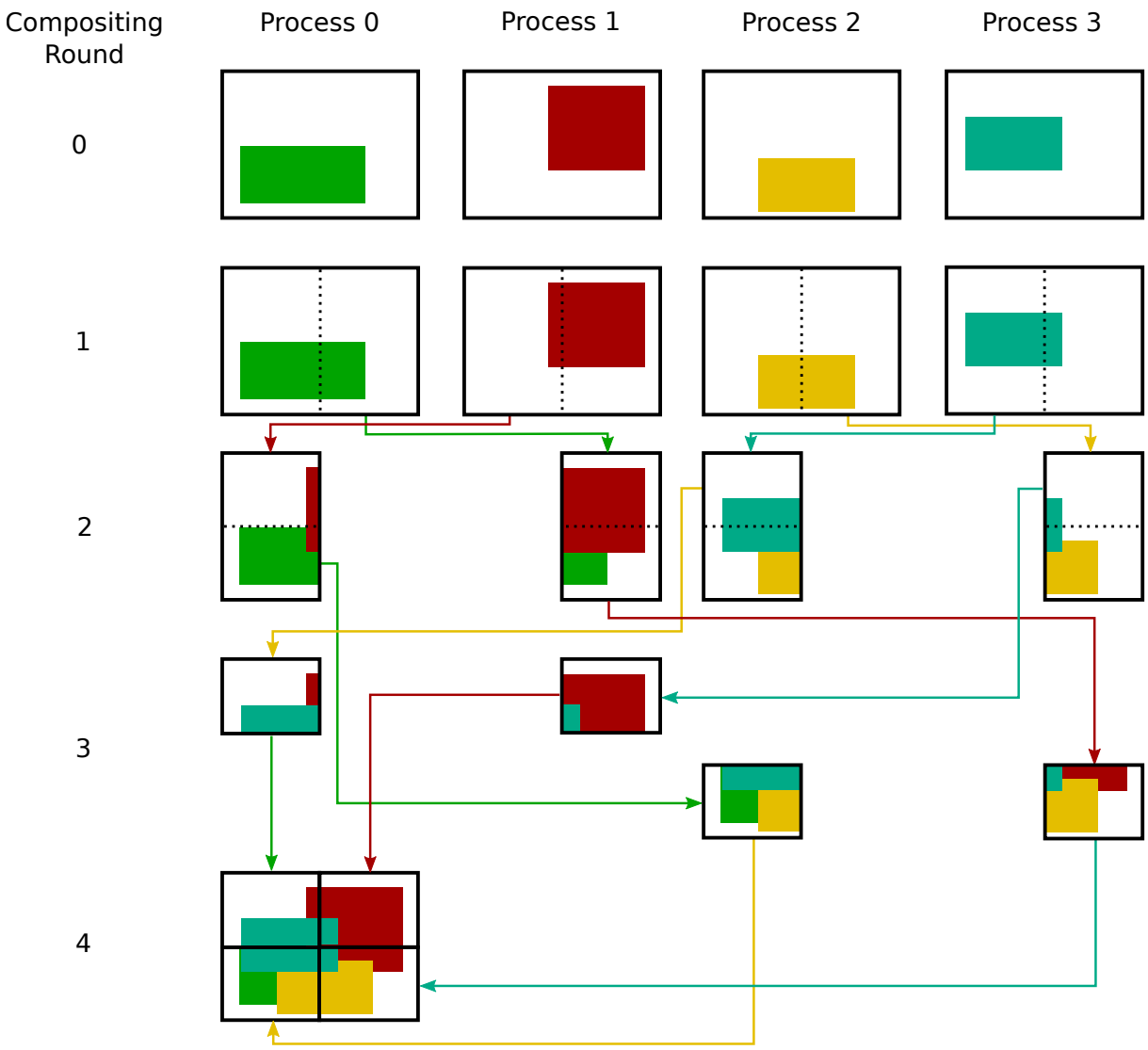



Figure 2.17: Binary-Swap image compositing with four processes.

# 3 A Tightly-coupled Visualization System

In situ visualization means performing processing and analysis of output data while the simulation producing this data is running. In a tightly-coupled approach, like the one we have chosen, the same computational resources are utilized for generating and visualizing the data. This means, the visualization uses the data structures provided by the simulation. Thus, no data movement, copying or disk storage is required eliminating network and I/O bottlenecks. At the same time the additional memory requirements are minimal. In our concrete case this means that the same HPC cluster and in particular the very same GPUs running the PIConGPU code and holding the computed data also perform the visualization. For producing meaningful images and extracting insight from the given data, several requirements had to be met.

## 3.1 Requirements

A primary goal of this thesis was to provide the researchers using PIConGPU with a more in-depth yet complete understanding of their simulation results. Thus, volume rendering (according to the emission-absorption model) was chosen as visualization technique. Insight into the spatial structure of 3D particle and field data can be obtained and hidden features may be found.

Adjusting the viewpoint from which the simulation is observed is critical to allow the investigation of interesting simulation regions. Thus, the virtual camera representing the observer should expose a sufficient degree of freedom to view any simulation area from any angle desired.

Being able to visually relate two physical quantities with each other is a way of finding relationships which are hardly detectable by looking at them separately. For that reason it is possible to render two volumes into the same image. Two different data sources can be selected and seamlessly weighted by a blending factor.

As the most interesting features might be located deep inside the volume, a means for extracting or emphasizing them must be provided. On the one hand, this can be achieved by adjusting the opacity mapping of the transfer function. Very high opacities can be assigned to a certain value range while other values can be made almost or completely transparent by mapping them to very low opacities. Various color scales are predefined to allow for good perception by all viewers (considering different color perception or even color blindness).

On the other hand, clipping can help to filter out unimportant parts of the volume and to concentrate only on the region of interest. Six axis-aligned clipping planes, each on one half-axis, should help in defining this region. By operating on spatial filtering criteria, clipping complements the information reduction performed by the opacity mapping of transfer functions.

By providing these features a set of use cases should be covered. At first, a scientist will be able to run a simulation with in situ visualization enabled. At any time he will be able to connect to the simulation and view the current progress. In fact, not only the scientist who started the simulation but anyone with the necessary network access and client software can join in. In that case, both clients are able to steer the simulation so that the users have to arrange their actions with each other. Another possible use is to present how fast the code can run. These two use cases lead to very important requirement. The simulation code is able to compute several time steps per second. Thus, the visualization is allowed to only add so much computational overhead that an interactive frame rate is preserved.

An additional use case is the production of pictures and videos for publications.

Besides the renderers features and performance, the user interface used to steer the visualization should be usable for researchers and developers. The focus should remain on the visualization, thus, the interface should be simplistic and disturb the user as little as possible.

## 3.2 Preconditions

Some characteristics of the HPC system available at HZDR had to be considered when designing the visualization system for this thesis. The network architecture depicted in Figure 3.1 requires an approach, in which network connections (e.g. TCP/UDP socket communication) that are used to transfer data between the cluster and the corporate network or the internet, are initiated by a node within the cluster. This is necessary because all cluster nodes are located in a private network, which is not addressable from any machine but the head node.

An individual node of the GPU cluster consists of two Intel Xeon E5-2609 Quad-Core CPUs clocked with 2.40 GHz and 64 GB of RAM. The lion's share of computational power is provided by four NVidia Tesla K20 GPUs. Each GPU possesses 2496 CUDA Cores clocked with 706 MHz and 5 GB GDDR5 RAM of total board memory connected to the chip thorough a 320-Bit interface. This allows a maximal memory bandwidth of 208 GB/s. There are 17 GPU nodes installed in the cluster summing up to 68 Kepler GPUs. As each GPU has a peak performance of 3.52 teraflops (single precision) the aggregate system peak performance is about 239.36 teraflops. The cluster is operated by the Linux distribution Ubuntu of version 12.04. An important limitation is the missing X server on the compute nodes. Thus, OpenGL is not supported. The cluster is fed with computational tasks by a portable batch system (PBS). These tasks (called job or batch job) are usually submitted by a script executed on the head node. The head node itself is not part of the computing nodes but responsible for managing jobs, the parallel file system, job queues and can be utilized for compilation and linking.

As the PIConGPU code is written in C/C++ and uses CUDA 5.5 and MPI 2.2 we will stick with this development environment too.

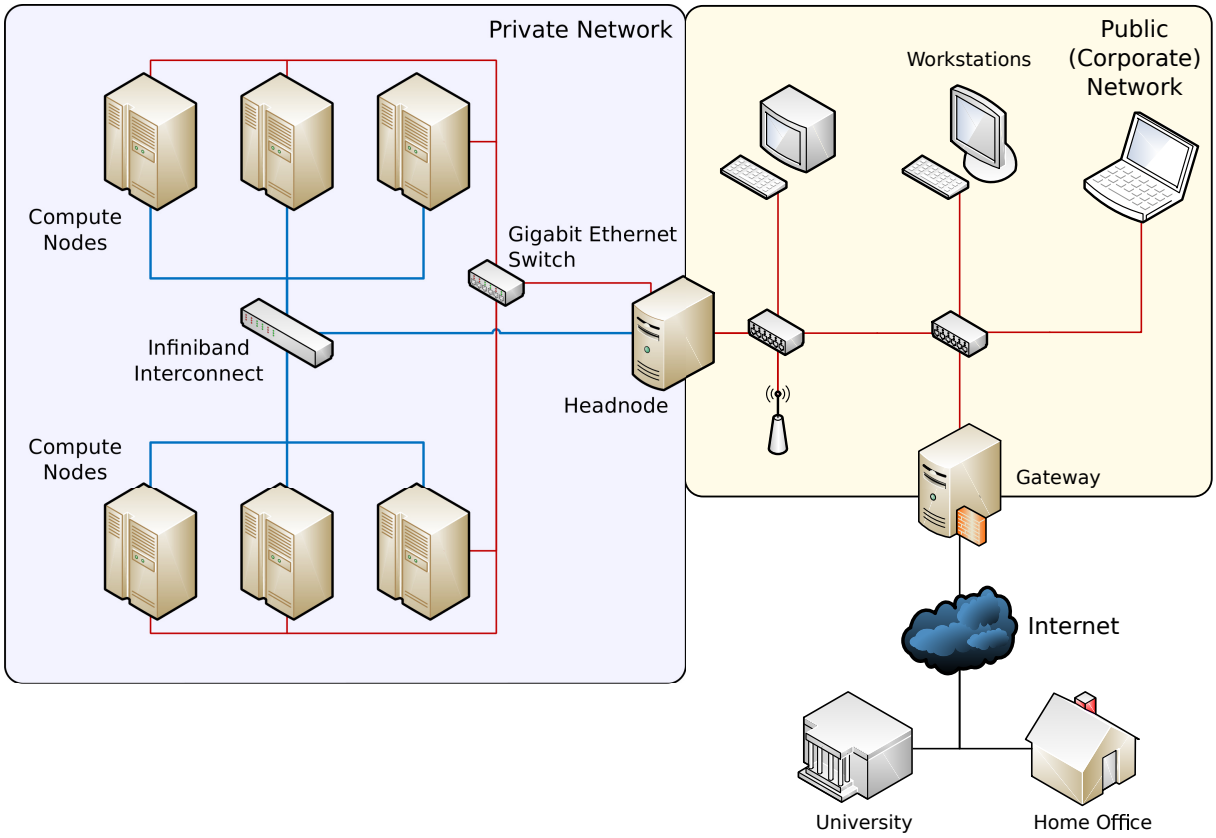



Figure 3.1: HPC cluster and network architecture

## 3.3 Architecture Overview

Below we describe the design decisions made to fulfill the requirements denoted in section 3.1 in consideration of the limitations illustrated in section 3.2. A tightly-coupled approach in which computational resources for simulation and visualization are shared was chosen. Additionally, a two way communication between user and simulation allows to steer the in situ processing (where only a one way communication would suffice for simple in situ visualization). Figure 3.2 depicts the high-level architecture of our system. It is split into three components: the PIConGPU simulation code with *visualization plug-in*, the *visualization server* and the *client*. This decomposition makes single components exchangeable. At the same time it decouples clients from simulations. The visualization server plays a central role because it accepts incoming connections from simulations and thereby solves the problem of connecting the private cluster network to the public (corporate) network where clients may reside. This allows simulations to run independently of any user observing or steering them. The visualization server runs permanently on the head node or some other machine within the corporate network.

A fixed message format defines the communication among the components. Every message is made up of an ID describing the content of the message, the length of the data in bytes and the message data itself. The way these messages are transferred between simulation and server on the one hand and between server and client on the other hand is handled differently. Messages between server and client are exchanged via the Remote Interactive Visualization Library (RIVLib)[1]. This library supports the

[1]Developed by Dr. Sebastian Grottel at TU Dresden.

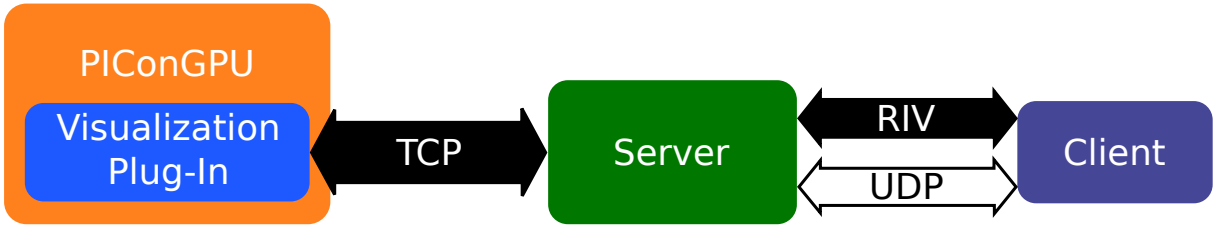
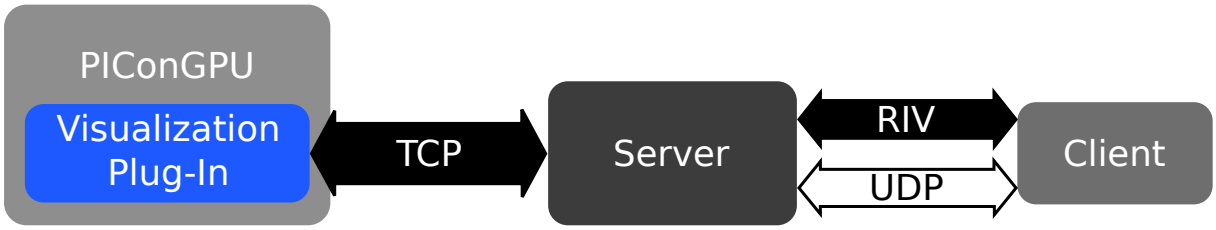



Figure 3.2: The three components and their communication protocols.

transfer of steering commands and (compressed) images. Messages between simulation and server are exchanged by simple TCP sockets. Among these two components a high-bandwidth network is available. Thus, compression is not necessary and the use of the RIVLib would introduce additional dependencies into the build process of the simulation. Through an extra communication channel clients can query the server for currently running simulations. A connectionless data transfer via UDP simply delivers a list of available simulations to the client.

A more detailed description of the single components will be given in the following. Chapter 4 describes the core of the system, the visualization plug-in. Chapter 5 will give a more detailed description of how the server handles connection attempts from simulations and clients concurrently. The concept of a client and a concrete implementation of a desktop client program is elaborated in Chapter 6.

# 4 The Parallel Volume Rendering Module




The core of our visualization system is the parallel in situ volume renderer. In short, the renderer is responsible for retrieving the simulation data of the current time step and rendering an image by volume ray-casting. This image has to be sent to the server. Finally, messages from the server are received and processed. In that way, several parameters may be set by the user interactively and the rendering process can be tweaked. The volume renderer implements all features presented in section 3.1 which are related to image generation on the given hardware and software platform described in section 3.2.

## 4.1 Integration with PIConGPU

To integrate a new module in the PIConGPU code one must implement a class that publicly inherits the `ISimulaitonIO` and `IPluginModule` interfaces (see section 2.3).

Listing 4.1: In situ renderer class definition with most important methods.

```
class InSituVolumeRenderer : public IPluginModule,
                             public ISimulationIO
{
protected:

        void moduleLoad();
        void moduleUnload();
        void moduleRegisterHelp(po::options_description& desc);
        void setMappingDescription(MappingDesc * cellDescription);
        void notify(uint32_t currentStep)
};
```

Then, the plug-in can be appended to the list of available modules. A new object is instantiated with the name `InSituVolumeRenderer` and the configuration prefix `insituvolvis`.

```
modules.push_back(new InSituVolumeRenderer("InSituVolumeRenderer", "insituvolvis"));
```

Now, the plug-in can be configured by including specific arguments into the simulation configuration file. The available parameters are:

**period *n***

The visualization plug-in will be invoked every $n$-th time step, e.g. if the period is ten, then only time steps $0, 10, 20$, etc. will be visualized.

**start *t***

The plug-in will start rendering at the specified time step $t$.

**serverip *IP***

The IP-address of the machine running the server.

**serverport *port***

The visualization will try to connect to the server on the given port.

**imagewidth *width***

The width of the final image in pixels.

**imageheight *height***

The height of the final image in pixels.

**name *name***

The name by which clients can identify a particular simulation.

An example configuration file is given in Listing 4.2. Every time step should be visualized starting at step 400 (line 20). The server is running on the machine with the IP-address `149.220.4.50` (line 23) and waits for incoming connections on port 8100 (line 24). The user intends to identify its simulation by naming it `PIConGPUInSituVis` (line 25). The resolution of the rendered images should be 800 by 800 pixels (line 21 and 22).

Listing 4.2: An example configuration file.

```
1  TBG_wallTime="1:00:00"
2
3  TBG_gpu_x=4
4  TBG_gpu_y=4
5  TBG_gpu_z=4
6
7  TBG_gridSize="-g 128 2048 128"
8  TBG_steps="-s 9096"
9  TBG_movingWindow="-m"
10 TBG_devices="-d !TBG_gpu_x !TBG_gpu_y !TBG_gpu_z"
11
12 TBG_programParams="!TBG_devices     \
13                    !TBG_gridSize    \
14                    !TBG_steps       \
15                    !TBG_movingWindow \
16                    !TBG_analyser  | tee output"
17
18 TBG_analyser="!TBG_insituvv"
19
20 TBG_insituvv="--insituvolvis.period 1 --insituvolvis.start 400 \
21               --insituvolvis.imagewidth 800 \
22               --insituvolvis.imageheight 800 \
23               --insituvolvis.serverip 149.220.4.50 \
24               --insituvolvis.serverport 8100 \
25               --insituvolvis.name PIConGPUInSituVis"
26
```

```
27 # TOTAL number of GPUs
28 TBG_tasks="$(( TBG_gpu_x * TBG_gpu_y * TBG_gpu_z ))"
29
30 "$TBG_cfgPath"/submitAction.sh
```

## 4.2 Initialization

At simulation start-up all registered plug-ins are initialized by calling their `moduleLoad` method. Tasks like allocating device memory needed by the plug-in have to be performed here, because all remaining memory will be consumed by the simulation itself.

As our plug-ins wants to be called every $n$-th time step according to the given period parameter it has to register itself at the `DataConnector` (`notifyFrequency` is the name of the member variable storing the period parameter):

```
DataConnector::getInstance().registerObserver(this, notifyFrequency);
```

After that, all data sources the user wants to observe at runtime have to be initialized. A data source may show some raw data properties like the x-component of the electric field or a computed property like field intensity or particle density.

In the next step the image resolution set by the `imagewidth` and `imageheight` parameters is cropped to a multiple of 16 in each dimension. This is done to match the CUDA thread blocks later on:

```
m_imageWidth -= m_imageWidth % 16;
m_imageHeight -= m_imageHeight % 16;
```

Each process has to store its MPI rank and the number of processes because some tasks will be performed only by the process with rank 0. For other tasks like image compositing it is necessary to know how many processes participate:

```
::MPI_Comm_rank(MPI_COMM_WORLD, &m_mpiRank);
::MPI_Comm_size(MPI_COMM_WORLD, &m_mpiSize);
```

For compositing the sub images rendered by each individual process we utilize the *Image Compositing Engine for Tiles* (IceT) library[1] [MKPH11]. Various compositing strategies like binary swap (see section 2.5), radix-k and tree reduction are implemented. The library is intended to enable OpenGL applications to perform sort-last compositing on very large displays. Such a display may be an array consisting of smaller displays and is called a *tiled display*. Even though the assumed use case for IceT is *multi-tile rendering*, single tile configurations are also possible. We configure IceT for our single tile (i.e. single image) case as follows:

```
::IceTCommunicator icetComm = ::icetCreateMPICommunicator(MPI_COMM_WORLD);
m_icetContext = ::icetCreateContext(icetComm);
```

The image data among processes is transfered via MPI. Thus, an IceT context with a communicator including all rendering processes has to be created. In our case each process produces a sub image, thus, `MPI_COMM_WORLD` is used.

[1] http://icet.sandia.gov

```
::icetStrategy(ICET_STRATEGY_SEQUENTIAL);
::icetSingleImageStrategy(ICET_SINGLE_IMAGE_STRATEGY_AUTOMATIC);
```

The sequential strategy "is recommended for the single tile case because it bypasses some of the communication necessary for the other multi-tile strategies." [Mor11] The single image compositing strategy is chosen automatically. This approach promises the best scalability.

```
::icetCompositeMode(ICET_COMPOSITE_MODE_BLEND);
::icetSetColorFormat(ICET_IMAGE_COLOR_RGBA_FLOAT);
::icetSetDepthFormat(ICET_IMAGE_DEPTH_NONE);
::icetEnable(ICET_ORDERED_COMPOSITE);
```

Our default fragment compositing scheme is alpha blending. This requires an opacity value to be stored in addition to the RGB color values. Pixel depth values are not needed at the moment. To guarantee correct blending the sub images need to be blended in the correct order. This order is computed by sorting the sub volumes by their distance to the observer and will be shown in more detail later on. For now, we just have to tell IceT that we need ordered compositing to be enabled.

```
::icetDrawCallback(InSituVolumeRenderer::drawWrapper);

::icetResetTiles();
::icetAddTile(0, 0, m_imageWidth, m_imageHeight, 0);
```

Lastly, we have to register a callback function which performs the actual rendering of a sub image and configure our single tile setup. The first four arguments of `icetAddTile` define the tile position in the global display (which is in our case made up of just one tile) and the tiles dimensions. The fifth argument identifies the associated *display process*, the process which holds the final composited image.

The color and depth information of the sub images are stored in two buffers. As the rendering process is performed on the GPU but the compositing of the final image is done on the CPU, we have to allocate the same amount of memory on the host and the device. The PIConGPU `GridBuffer` class template provides this functionality:

```
m_pPixelBuffer = new GridBuffer<ColorRGBA, DIM2>(DataSpace<DIM2>(m_imageWidth, m_imageHeight));
m_pDepthBuffer = new GridBuffer<float, DIM2>(DataSpace<DIM2>(m_imageWidth, m_imageHeight));
```

We instantiate the template for our color buffer with a 4-tuple of floats as data type (`ColorRGBA`, representing the RGBA components) and the size of our image. Our depth buffer stores the depth of a pixel in a single float. Now, enough memory for our buffers is allocated on the device and in system RAM.

```
if (m_mpiRank == 0)
{
        m_socket_connector = new TCPConnector();
        m_socket_stream = m_socket_connector->connect(m_visServerIP, m_visServerPort);

        if (m_socket_stream != NULL)
        {
                m_socket_stream->send(VisName, m_name.c_str(), m_name.size());

                uint32_t * image_size = new uint32_t[2];
                image_size[0] = m_imageWidth;
                image_size[1] = m_imageHeight;
```

```
                m_socket_stream->send(ImageSize, image_size, 2 * sizeof(uint32_t));
        }
}
```

The final step of our initialization is to connect to the visualization server. For that purpose the class `TCPConnector` was implemented. Only the process of rank 0 establishes a connection. This is our *root* process, coordinating all communication with the server. After the connection is initiated successfully we tell the server the name of our simulation and the dimensions of the final image we will transmit.

## 4.3 The Interactive Visualization Loop

The bulk of work is done inside the plug-ins `notify` method. This method is called every $n$-th time step according to the configured period. Figure 4.1 illustrates the stages run through to render an image, send it to the server and process messages received from the server.

### 4.3.1 Data Retrieval

Our first task is to get the new simulation data from the `DataConnector`. As the simulation window we are observing might have been shifted since the last time step, we have to update the offset used to address our data array. This is done by the `update` method of the data source. Here, an interesting aspect of programming in different memory spaces comes up. The update of the data source is initiated on the host but the actual access to the data will occur on the device. Thus, we have to pass the offset and the pointer to the simulation data to the device. As the observed data source is not fixed but can be changed by the user at runtime, we split the data source into two classes. The host part is responsible for initializing, updating and destroying the device part. The device part directly reads data from the simulation data structures and delivers a scalar value for any given sampling position.

The interfaces for host and device data sources are shown below:

```
class IDeviceDataSource
{
public:
    __device__ virtual ~IDeviceDataSource() { }
    __device__ virtual float valueAt(float3 pos) = 0;
};

class IHostDataSource
{
public:
    __host__ virtual ~IHostDataSource() { }
    __host__ virtual std::string name() = 0;
    __host__ virtual void initialize() = 0;
    __host__ virtual void release() = 0;
    __host__ virtual void update(DataSpace<DIM3> offset, DataSpace<DIM3> guarding) = 0;
    __host__ virtual IDeviceDataSource ** devicePtr() = 0;
};
```

The data sources to read the electric field intensity shall serve as an example. The initialization method of the host data source calls a kernel on a single CUDA thread to create the device data source object.

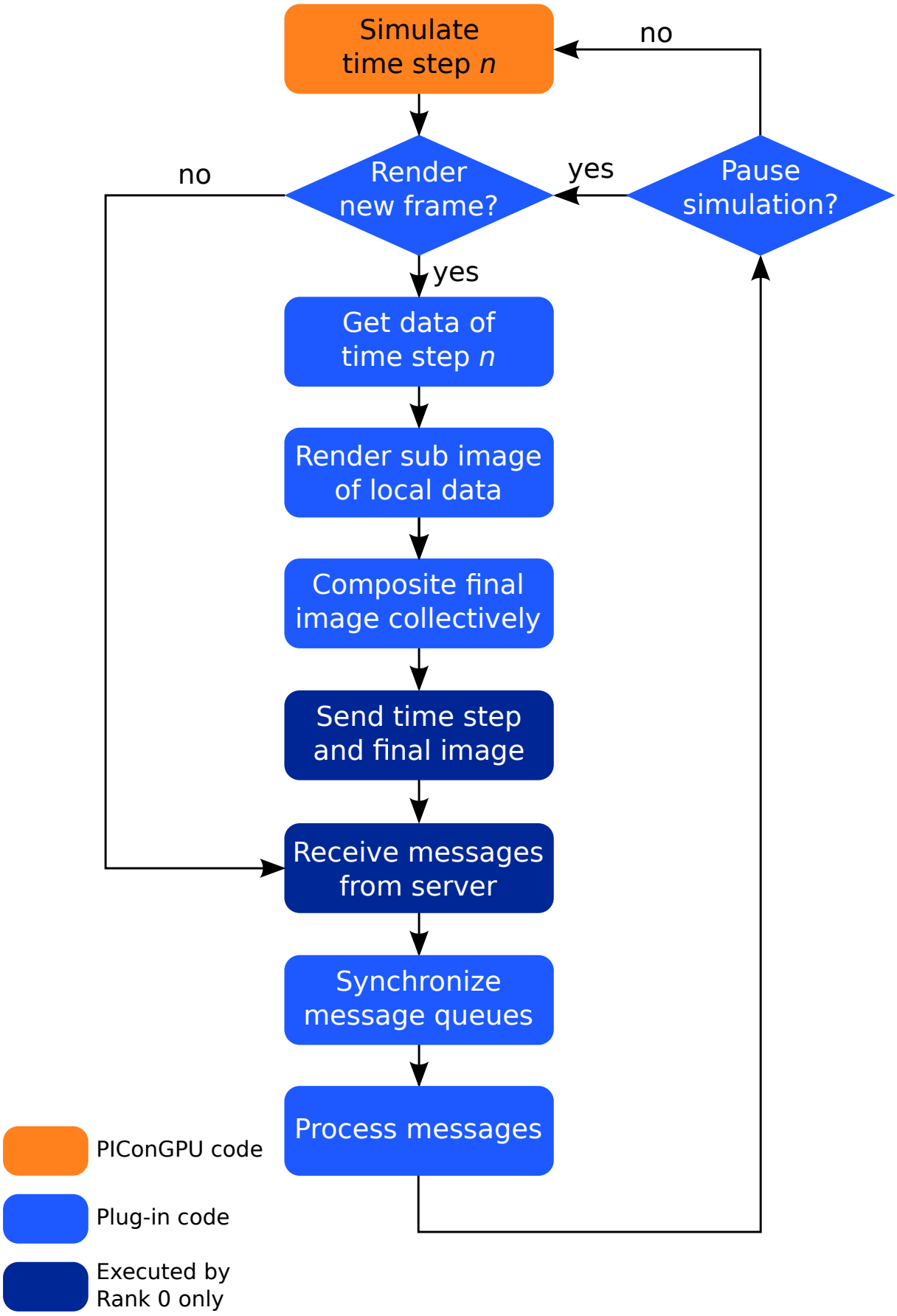



Figure 4.1: Interactive visualization loop.

This object is stored in global memory and persists throughout the whole application runtime and is not lost after the lifetime of the creating thread:

```
/// executed on the host
__host__ virtual void initialize()
{
    dim3 tb(1);
    ::cudaMalloc(&m_devicePtr, sizeof(IDeviceDataSource**));
    createDeviceDataSource<DeviceFieldESource><<<tb, tb>>>(m_devicePtr);
}

/// executed on the device
template <class DataSource>
__global__ void createDeviceDataSource(IDeviceDataSource ** devicePtr)
{
    (*devicePtr) = new DataSource();
}
```

A pointer to the device object is stored in the host object (`m_devicePtr`) and can be fetched by calling the `devicePtr()` method on the host source. The offset and data pointer are updated in the `update` method:

```
/// on the host
__host__ virtual void update(DataSpace<DIM3> offset, DataSpace<DIM3> guarding)
{
    DataConnector& dc = DataConnector::getInstance();
    FieldE * pFieldE = &(dc.getData<FieldE>(FIELD_E, true));
    m_DataBox = pFieldE->getDeviceDataBox();

    dim3 tb(1);

    setDeviceDataBox<DeviceFieldESource><<<tb, tb>>>(m_devicePtr, m_DataBox,
                                                     offset, guarding);
}

/// on the device
template <class DataSource, class BoxType>
__global__ void setDeviceDataBox(IDeviceDataSource ** devicePtr, BoxType box,
                                 DataSpace<DIM3> offset, DataSpace<DIM3> guarding)
{
    DataSource * source = (DataSource*)(*devicePtr);
    source->setOffsetGuarding(offset, guarding);
    source->setDataBox(box);
}
```

In the same manner the data source is released:

```
__host__ virtual void release()
{
    dim3 tb(1);
    deleteDeviceDataSource<<<tb, tb>>>(m_devicePtr);
    ::cudaFree(m_devicePtr);
}

__global__ void deleteDeviceDataSource(IDeviceDataSource ** devicePtr)
{
    if ( (*devicePtr) != NULL )
        delete (*devicePtr);
}
```

This pattern is used to implement a number of data sources and allows to extend the set of observable data sources.

### 4.3.2 CUDA Volume Ray-Casting

The actual rendering process is controlled by IceT. We simply call the function `icetDrawFrame` in each process and obtain an `icetImage` in our root process which encloses the final image. IceT invokes the draw callback as necessary to render sub images needed for compositing the full image.

Individual sub images are generated by volume ray-casting on the local data of each GPU. Ray-casting was chosen because it is an embarrassingly parallel algorithm. Moreover, it is a direct volume rendering technique that does not employ any proxy geometry for rendering. No graphics primitives such as polygons need to be rasterized and thus the lack of OpenGL support does not pose a problem.

To utilize the massive parallelism exposed by the CUDA architecture our ray-casting algorithm is parallelized in image space. For each pixel one thread is started. This is achieved by calling the rendering kernel with an execution configuration that assigns a thread-block to a coherent portion of the screen. A possible configuration would render a square of $256$ pixels by a block of $16 \times 16$ threads.

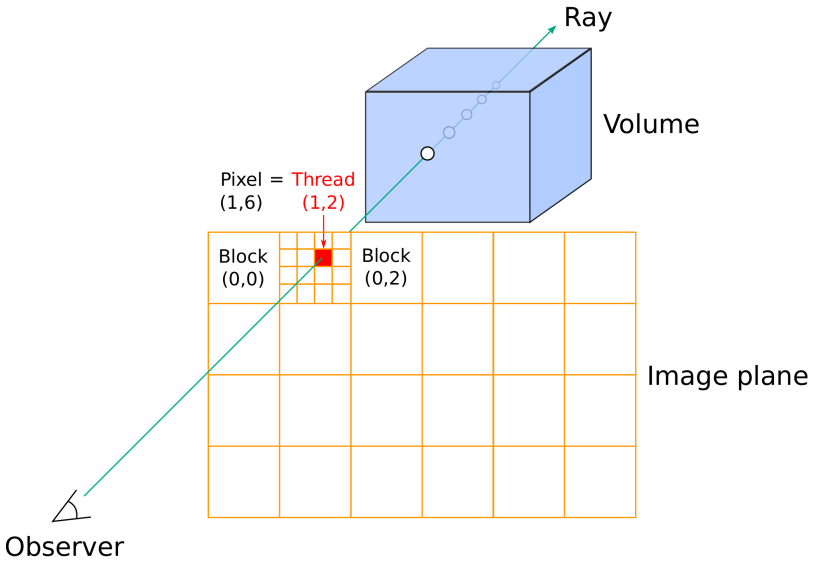



Figure 4.2: Parallelization of volume ray-casting in image space. A grid of $6 \times 4$ blocks with $4 \times 4$ threads is configured. The final image resolution is $24 \times 16$ pixels. A thread can calculate the coordinates of the pixel he is responsible from the block dimensions and his thread index. Thus, thread $(1, 2)$ in block $(0, 1)$ renders pixel $(1, 6)$.

This means one thread casts one ray for one pixel. At first our kernel needs to construct that ray. As we utilize perspective projection the origin of all rays is given by the observer position. This position can be extracted from the last column of the inverse view matrix.

```
ray.origin = make_float3from4(c_invViewMatrix * make_float4(0.0f, 0.0f, 0.0f, 1.0f));
```

The direction of the ray is determined by the point through which he passes the image plane. This point is given by the coordinate of the pixel this ray is cast for. A thread calculates the pixel he is responsible for from the block and thread index and the dimensions of a block.

```
const int x = blockDim.x * blockIdx.x + threadIdx.x;
const int y = blockDim.y * blockIdx.y + threadIdx.y;
```

These pixel coordinates are transformed to normalized screen coordinates in the range $[-1; 1[$ by scaling and translating them.

```
float u = ((float)x / (float)imageWidth) * 2.0f - 1.0f;
float v = ((float)y / (float)imageHeight) * 2.0f - 1.0f;
```

From these normalized screen coordinates we can build a direction vector.

```
ray.dir = normalize(make_float3(u, v, -1.0f));
```

This direction vector now has to be transformed from camera space to world space, the space our volume is located in. This is achieved by appending a zero as homogeneous w-component and multiplying the normalized direction vector with the inverse view transform.

```
ray.dir = c_invViewMatrix * make_float4(ray.dir.x, ray.dir.y, ray.dir.z, 0.0f);
```

The next step is to test our ray for intersection with the local volume bounds. Before that, the bounding box is cropped to fit the clipping volume defined by the six clipping planes.

```
float3 clipped_box_min = fmaxf(volMin, clip_min);
float3 clipped_box_max = fminf(volMax, clip_max);
```

Now, this cropped box is tested for intersection with the ray. A fast method developed by Kay and Kayjia [Owe13] based on "slabs" where a slab is the space between two parallel planes is utilized.

```
hit = intersectBox(ray, clipped_box_min, clipped_box_max, &t_in, &t_out);
```

If the ray misses the bounding box we simply write transparent black to the color buffer and the maximum depth of 1.0 to the depth buffer and terminate the thread. In the case of hitting the box we set the ray parameter `t` to the entry parameter `t_in`. We have to assure that the entry ray parameter is a multiple of the sampling interval (defined by the constant `TSTEP`). Otherwise artifacts may occur when a ray crosses more than one sub volume because superfluous sampling points are taken and composited. Figure 4.3 illustrates the problem.

```
t = TSTEP * ceil(t_in / TSTEP);
```

Now we enter the ray-casting loop which solves the volume rendering integral. First we need to sample the volume data set(s). By parameterizing our ray with the parameter `t` we obtain the sampling position in world space. Now we can query our data source(s) for a scalar value at that point. Additionally a second sample one step further along the ray is fetched.

```
float3 sampling_position = ray.origin + ray.dir * t;
float sample_A_front = (*dataSourceA)->valueAt(sampling_position);
float sample_A_back = (*dataSourceA)->valueAt(sampling_position + ray.dir * TSTEP);
```

The data source maps the sampling position from world space to data space. A nearest neighbor or trilinear interpolation is performed inside the `valueAt` method, depending on the underlying data set. Only some data sets support higher-order interpolation schemes. Others lack the necessary data at the border of their sub volume. After the scalar values are fetched from the data sources we can map them to color and opacity by utilizing the transfer function. Our transfer function is stored in a 2D CUDA texture object. Texture lookups are performed with the `tex2Dlayered` function.

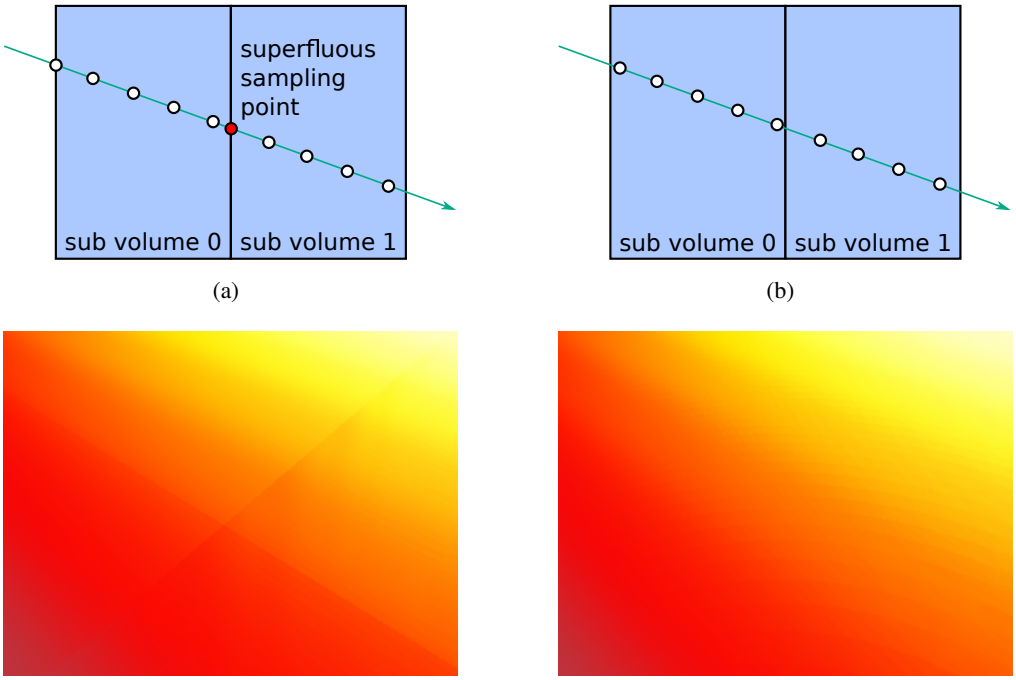



Figure 4.3: Oversampling rays at the border of sub volumes can cause artifacts (4.3(a) and 4.3(c)). This can be avoided by maintaining a constant sampling interval over sub volume borders (4.3(b) and 4.3(d)).

```
float4 color_A = tex2DLayered<float4>(transferFuncTex, sample_A_front, sample_A_back, 0);
float4 color_B = tex2DLayered<float4>(transferFuncTex, sample_B_front, sample_B_back, 1);
```

Two samples are needed because a pre-integrated transfer function is utilized (see Section 4.3.3). A layered texture stores both transfer functions - one for each data source - in a single texture object.

Now that the simulation data values are transformed to optical properties the next step is to apply a compositing scheme. By default this is the front-to-back alpha compositing scheme. In that, the color just obtained is blended with the color resident in the image buffer. But first we have to blend our two data sources according to the weighting parameter chosen by the user. A linear interpolation is applied in the following way:

```
float4 weighted_color = color_A * weight_of_A + color_B * (1.0f - weight_of_A);
```

Now we can apply front-to-back blending (see Equation 2.7), where `final_color.w` denotes the accumulated opacity:

```
final_color = final_color + weighted_color * (1.0f - final_color.w);
```

The last step in our ray-casting loop is to increment the sampling parameter `t` and advance the sampling position:

```
t += TSTEP;
sampling_position += ray.dir * TSTEP;
```

If the sampling parameter exceeds the volume exit parameter `t_out` the loop is terminated. The final color is written to the color buffer.

```
pixelBuffer[y][x].r = final_color.x;
pixelBuffer[y][x].g = final_color.y;
pixelBuffer[y][x].b = final_color.z;
pixelBuffer[y][x].a = final_color.w;
```

In addition to alpha blending the compositing modes *Maximum Intensity Projection* and *Iso Surface* have been implemented. For those the compositing equations 2.9 respectively 2.10 are used to determine the final fragment color.

The three compositing schemes implemented are shown in Figure 4.4. The image obtained by applying alpha blending as compositing scheme shows how different levels of field strength (i.e. intensity) enclose each other, resulting in more a dense volume towards the middle. By switching to maximum intensity projection we can observe the change in intensity towards the center without occlusion by lower intensity values. This gives full information about the range of intensities present in the volume. Anyhow, the image produced loses depth information. By rotating the viewpoint around the volume we can get some cues about the spatial structure by exploiting the movement parallax. The iso surface composting shows a region of equal intensity and provides a very good sense of depth. The Blinn-Phong model was applied as local illumination method to make it easier to perceive the contours and orientation of the surface. For gradient estimation the central differences technique was used (see Equation 4.1).

$$\nabla f(x,y,z) = \frac{1}{2h}\begin{pmatrix} f(x+h,y,z) - f(x-h,y,z) \\ f(x,y+h,z) - f(x,y-h,z) \\ f(x,y,z+h) - f(x,y,z-h) \end{pmatrix} \tag{4.1}$$

It requires only six additional samples around the sampling position and keeps the computational cost minimal. However, in our case it is not sufficient to produce a smooth surface shading. Fast changing gradients and areas of homogeneous intensity values lead to artifacts and numerical instabilities (i.e. if the differences obtained by sampling the neighborhood of a point are close to zero, the calculated surface normal will be zero, producing black spots on the surface). Higher order schemes with filter kernels considering more voxels may yield better results. But they would require more neighboring cells. Those cells are not always present in the simulation data structures. Thus, changing the simulation code or exchanging them before rendering would be necessary.

(a) Alpha Blending (b) Iso Surface (c) Maximum Intensity Projection

Figure 4.4: The three different compositing modes by example of the same data and time step.

### 4.3.3 Pre-Integrated Transfer Functions

Earlier we mentioned that a pre-integrated transfer function is employed. As the discrete sampling of the scalar field retrieved from the data is just an approximation, we can improve the sampling and therewith the image quality by taking the continuous nature of the scalar field into account. The basic idea is to render the intervals between the samples yielding a much more precise reconstruction of the underlying scalar field. These intervals are defined by successive sampled scalar values $s_f$ and $s_b$ on a ray (see Figure 4.5).

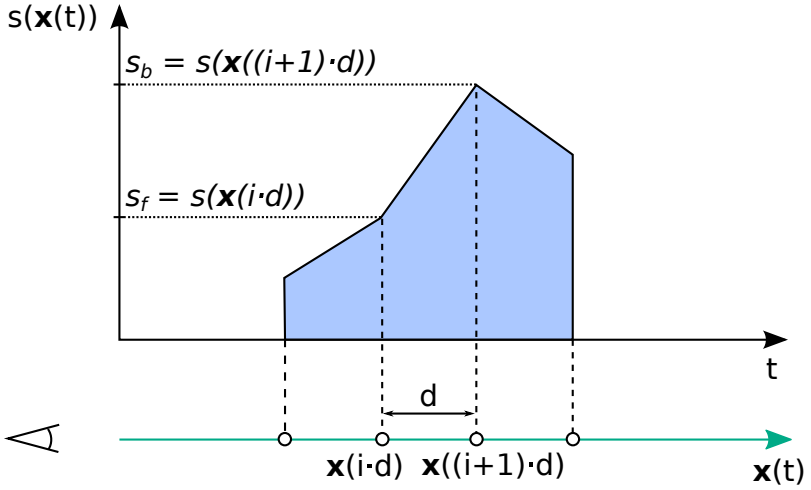



Figure 4.5: Definition of the i-th ray segment. $d$ denotes the length of the sampling interval, $\mathbf{x}(t)$ a sampling position on the ray determined by the ray parameter $t$ and $s(\mathbf{x})$ the scalar value at position $\mathbf{x}$.

To speed up the computation of color and opacity of such an interval a lookup table storing all possible combinations of sampling values and interval sizes is used. This would require a 3D lookup table. As the sampling interval is usually constant the dimensionality of the table decreases to two. A lookup is then performed with the two scalar values $s_f$ and $s_b$ defining the interval. See the paper of Engel et al. for a more thorough discussion [EKE01].

Whenever the transfer function is altered the lookup table has to be recalculated. In CPU and even hardware-accelerated implementations this can take from several seconds to minutes. As we want the user to interactively tweak the transfer function for his needs, the recalculation is performed in a highly parallel manner. The user specifies a 1D lookup table of color and opacity values for each of the two transfer functions in the client. These 1D tables are sent to the renderer. There, we utilize CUDA to pre-integrate these 1D tables and store them in a layered 2D texture. A kernel performing the pre-integration is called with an execution configuration that employs one thread per entry in the resulting pre-integration table. That means, if we receive two lookup tables with $64$ scalar to color/opacity entries each from the user, a grid with $4 \times 4$ blocks and each block containing $16 \times 16 \times 2$ threads is configured and executed.

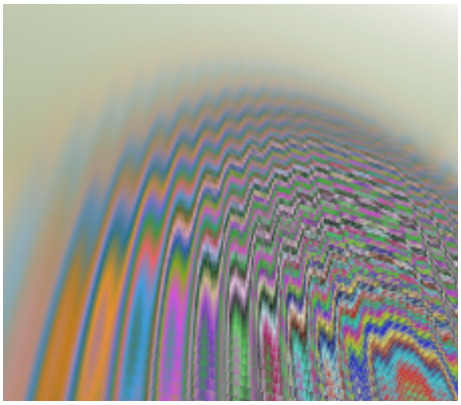
```
dim3 blocks(4, 4, 1);
dim3 threads(16, 16, 2);
kernelPreintegrateTransferFunc<<<blocks, threads>>>(preintTFDeviceBuffer);
```

Now each thread identifies the entry he shall compute by its block and thread index. At the end, he writes

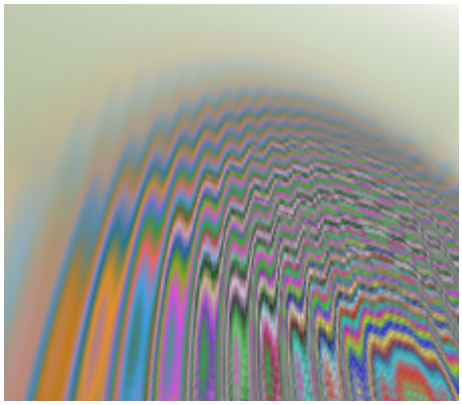
the computed value to the corresponding entry of the buffer storing the pre-integration table.

```
__global__ void kernelPreintegrateTransferFunc(TF_DataBox targetBuffer, float tstep)
{
        const int sf = blockDim.x * blockIdx.x + threadIdx.x;
        const int sb = blockDim.y * blockIdx.y + threadIdx.y;
        int layer = threadIdx.z;

        ColorRGBA pre_integrated_color;

        /// perform pre-integration ...

        targetBuffer[layer][sb][sf] = saturate(pre_integrated_color);
}
```

Figure 4.6 shows the improvement in image quality as well as the original and the pre-integrated transfer function.



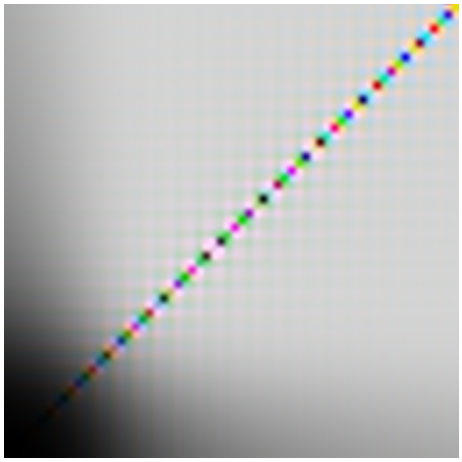
(a) Rendered without Pre-Integration



(b) Rendered with Pre-Integration



(c) Original Transfer Function



(d) Pre-Intergated Transfer Function

Figure 4.6: Pre-integration transforms the original transfer function into a two-dimensional lookup table. High frequencies in the transfer function are reproduced giving a better visual result.

### 4.3.4 Image Compositing

The final stage in image generation is collectively compositing the sub images rendered by each process individually into one final image. This step is performed by IceT on the CPU. The same compositing scheme used in the ray-casting kernel to combine the fragments on a ray has now to be applied to all fragments from the sub images covering the same pixel in the final image.

When using the iso surface compositing scheme only the depth of the fragments needs to be compared and the fragment with the smallest depth value is written to the final image. This is a well-known technique known as depth compositing or Z-/depth-buffering and directly supported by IceT:

```
::icetCompositeMode(ICET_COMPOSITE_MODE_Z_BUFFER);
::icetSetColorFormat(ICET_IMAGE_COLOR_RGBA_FLOAT);
::icetSetDepthFormat(ICET_IMAGE_DEPTH_FLOAT);
```

If alpha blending is used for compositing we need to account for the mathematical properties of the blending operator.

$$C_{out} = C_{dst} + (1 - \alpha_{dst}) \cdot C_{src}$$

The operator is associative which allows to split rays into segments and parallelize ray-casting in general. However, it is not commutative. That means we have to keep the correct order when compositing the ray segments as we have to keep the correct order when marching through the volume, sampling and compositing the obtained fragments. To capacitate IceT to perform the image composition in the correct order we need to tell it which process' sub volume is in front or behind of that of another process. This is accomplished by two methods executed before every draw call. The `gatherVolumeCenters` method computes the process' local volume bounds. Therefore it reads the local data grid size and offset from the data structures representing the observed simulation area (i.e. `MovingWindow`) and the local simulation grid (i.e. `SubGrid`). The center of the local volume in world space is computed and all processes exchange their centers. The centers of all processes are now stored in the member `m_volumeCenters` sorted by rank.

```
VirtualWindow window(MovingWindow::getInstance().getVirtualWindow(currentStep));
DataSpace<3> localSize = window.localSize;
DataSpace<3> globalOffset = SubGrid<3>::getInstance().getSimulationBox().getGlobalOffset();

this->gatherVolumeCenters(globalOffset + window.localOffset, window.localSize);

void gatherVolumeCenters(DataSpace<DIM3> globalOffset, DataSpace<DIM3> localSize)
{
        m_localVolumeMin.x = globalOffset.x() * CELL_WIDTH;
        m_localVolumeMin.y = globalOffset.y() * CELL_HEIGHT;
        m_localVolumeMin.z = globalOffset.z() * CELL_DEPTH;

        m_localVolumeMax.x = (globalOffset.x() + localSize.x()) * CELL_WIDTH;
        m_localVolumeMax.y = (globalOffset.y() + localSize.y()) * CELL_HEIGHT;
        m_localVolumeMax.z = (globalOffset.z() + localSize.z()) * CELL_DEPTH;

        /// exchange local volume centers with other MPI processes
        float volumeCenter[3];

        volumeCenter[0] = m_localVolumeMin.x+0.5f*(m_localVolumeMax.x - m_localVolumeMin.x);
```

```
        volumeCenter[1] = m_localVolumeMin.y+0.5f*(m_localVolumeMax.y - m_localVolumeMin.y);
        volumeCenter[2] = m_localVolumeMin.z+0.5f*(m_localVolumeMax.z - m_localVolumeMin.z);

        /// the volume centers are stored to the receive buffer ordered by rank
        ::MPI_Allgather(volumeCenter, 3, MPI_FLOAT,
                        m_volumeCenters, 3, MPI_FLOAT, MPI_COMM_WORLD);
}
```

Now, the method `computeCompositeOrder` can compute a visibility ordering.

```
1  int * computeCompositeOrder()
2  {
3          std::multimap<float, int> distancesToRanks;
4
5          for (int r = 0; r < m_mpiSize; r++)
6          {
7                  float3 cam_pos = m_camera->getPosition();
8
9                  float distance = /// compute distance to camera
10
11                 distancesToRanks.insert( std::pair<float, int>(distance, r) );
12         }
13
14         int * compositeOrder = new int[m_mpiSize];
15         std::multimap<float, int>::iterator it = distancesToRanks.begin();
16
17         for (int i = 0; i < m_mpiSize; i++)
18         {
19                 compositeOrder[i] = it->second;
20                 it++;
21         }
22
23         return compositeOrder;
24 }
```

To sort the ranks according to their distance to the camera a `multimap` container is utilized where the distance serves as key and the process rank as value. Upon insertion of a new key-value pair the container is sorted by the `less` predicate on the key and thus ranks with a smaller distance to the camera are always stored in front of those with a bigger distance. Now, the values are simply copied to an array of `int`s which is returned. This array is set as the depth order for IceT:

```
::icetCompositeOrder(computeCompositeOrder());
```

Figure 4.7 illustrates a possible depth ordering.

Looking at the sub images and the final image shows the sub volumes and the complete simulation area (see Figure 4.8).

The third compositing scheme supported by our renderer is maximum intensity projection. This compositing operation is not directly supported by IceT. Nonetheless we can emulate it with depth compositing. Therefore we used the inverse intensity as our fragment depth.

```
final_depth = 1.0f - intensity;
```

This assures the fragment with the highest intensity value to be on top of all others and therefore be picked by depth compositing as the final pixel color.

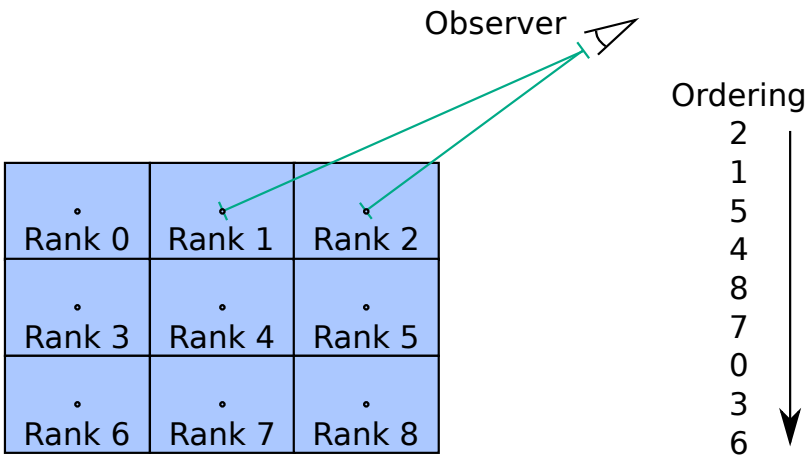



Figure 4.7: Sub volumes with their ranks and the resulting depth ordering.

After the image has been composited it is send to the server together with the current time step (done only by the master process).

```
if (m_mpiRank == 0)
{
        m_socket_stream->send(TimeStep, &currentStep, sizeof(uint32_t));
        IceTFloat * pixels = ::icetImageGetColorf(image);
        m_socket_stream->send(Image, pixels, image_width * image_height * sizeof(IceTFloat));
}
```

Note that the whole process of rendering a new frame is only performed if necessary, i.e., if new simulation data is available or the user performed an action that causes a change in the image (see Figure 4.1, “Render new frame?”). This avoids needless computations and network traffic.

### 4.3.5 Steering

To allow the interactive steering of the simulation and tweaking the rendering parameters we need a means of communicating with the renderer. Thus, a set of classes was implemented to encapsulate message transfer via TCP. We have already seen the `TCPConnector` and `TCPStream` classes in use when initializing our plug-in (see section 4.2). The connector class simply tries to establish a TCP connection to the machine with the given IP address on the given port. If successful, an object of type `TCPStream` is returned. The stream class provides the methods `send` and `receive` to asynchronously exchange messages. Internally, the stream class puts messages which are to be sent in a message queue and runs a worker thread which performs the socket communication.

To send a message its ID, size in bytes and a pointer to the message data are supplied to the streams `send` method. To receive a message we provide the `receive` method with pointers to integer variables where ID and size shall be stored as well as a `void` pointer to store the message data. Additionally, we can specify if we want to wait for a message to arrive (i.e. block program execution) or return an empty message if no messages are available (i.e. non-blocking).

The first step in handling steering messages is to receive them from the server. This task is performed by the master process only. When a message is received the master process stores it in its message queue. In

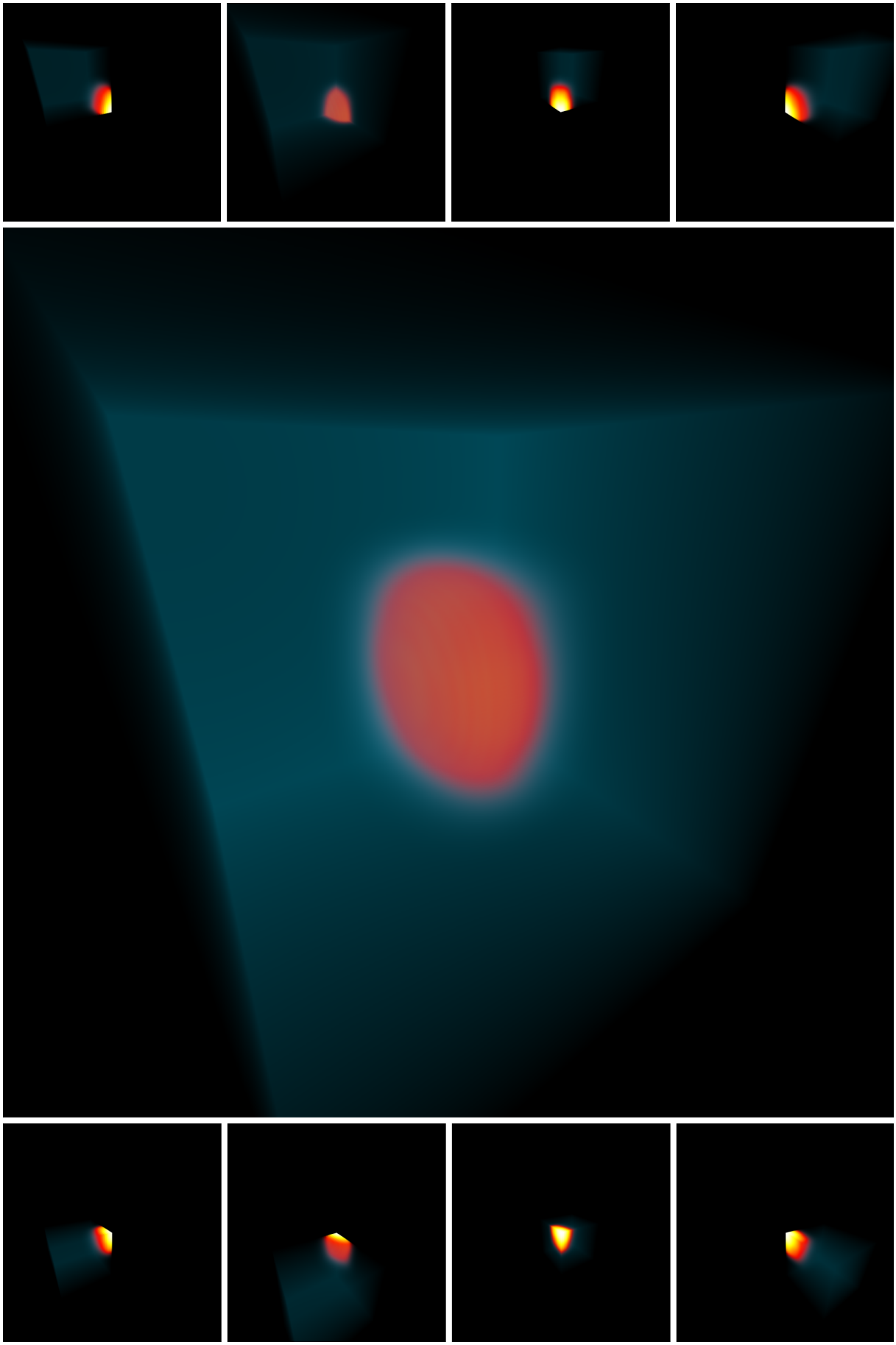


Figure 4.8: Eight sub images and the resulting final image.

a second step, the masters message queue is synchronized with that of all other processes. The method `synchronizeMessageQueue` implements this functionality by utilizing our `TCPStream` class and the `MPI_Bcast` procedure.

```
1  void synchronizeMessageQueue()
2  {
3          m_messageQueue.clear();
4
5          if (m_mpiRank == 0) {
6                  uint32_t id, length;
7                  void * buffer;
8                  while (1) {
9                          m_socket_stream->receive(&id, buffer, &length, false);
10
11                         if (id == NoMessage) break;
12                         else {
13                                 Message msg;
14                                 msg.id = id;
15                                 msg.length = length;
16                                 msg.data = buffer;
17
18                                 m_messageQueue.push_back(msg);
19                         }
20                 }
21         }
22
23         int mqLength = m_messageQueue.size();
24         ::MPI_Bcast(&mqLength, 1, MPI_INT, 0, MPI_COMM_WORLD);
25
26         for (int i = 0; i < mqLength; i++) {
27                 Message msg;
28
29                 if (m_mpiRank == 0)
30                         msg = m_messageQueue[i];
31
32                 ::MPI_Bcast(&msg.id, 1, MPI_INT, 0, MPI_COMM_WORLD);
33                 ::MPI_Bcast(&msg.length, 1, MPI_INT, 0, MPI_COMM_WORLD);
34
35                 if (m_mpiRank != 0)
36                         msg.data = new char[msg.length];
37
38                 ::MPI_Bcast(msg.data, msg.length, MPI_BYTE, 0, MPI_COMM_WORLD);
39
40                 if (m_mpiRank != 0)
41                         m_messageQueue.push_back(msg);
42         }
43 }
```

After the message queue is cleared and the master process received all messages available the length of the queue is broadcast to all processes. Now we can broadcast the ID and size of each message in the masters queue to the other processes. After that, the message data is broadcast as a byte-array. This results in all processes having the exact same messages ordered in the same way in their queue. If this was not the case, some processes may render the data with a different transfer function or camera position resulting in image artifacts.

Eventually, each process can handle the messages in his queue. To investigate the current state of the simulation more precisely the user can pause the computation of new time steps. The corresponding

message would carry the `SimPause` ID and no data. To allow the simulation to resume running the `SimPlay` command is sent (see Figure 4.1, “Pause simulation?”). These messages are handled by the `handleMessages` method of our renderer (Appendix A contains a complete list of all message IDs).

```
void handleMessages()
{
        for (std::deque<Message>::iterator it = m_messageQueue.begin();
                it != m_messageQueue.end(); it++)
        {
                switch (it->id)
                {
                        case SimPlay: {
                                m_isRunning = true;
                        } break;

                        case SimPause: {
                                m_isRunning = false;
                        } break;

                        /// handle other message IDs here ...
                }
        }
}
```

## 4.4 Clean-Up

To release allocated resources and safely shut down our plug-in we utilize the `moduleUnload` method. It is automatically called at the end of the simulation run.

```
if (m_mpiRank == 0) {
        m_socket_stream->send(CloseConnection, NULL, 0);
        m_socket_stream->wait_async_completed();
        delete m_socket_stream;
        m_socket_stream = NULL;

        delete m_socket_connector;
        m_socket_connector = NULL;
}
```

We tell the server that the TCP stream connection should be terminated, wait for the message to be sent and delete the `TCPStream` object. This closes the streams internal socket descriptor used for communication.

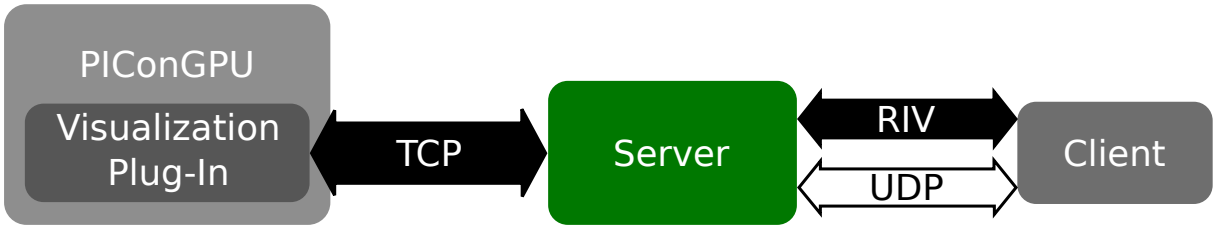
IceT is shutdown by simply destroying the context created in the initialization phase.

```
::icetDestroyContext(m_icetContext);
```

Lastly, we free the allocated color and depth buffer memory.

```
delete m_pPixelBuffer;
delete m_pDepthBuffer;
```

# 5 The Visualization Server




The visualization server is the component making the link between simulations using our plug-in and clients intending to observe (and steer) an in situ visualization. The server has to run on a machine that is reachable from both the simulation and the client(s). In our case, the cluster head node was usually utilized to run the server, because it has a high-speed interconnect to the compute nodes and is reachable from every machine within the corporate network. Via VPN[1] it is possible to connect from any machine on the internet.

Before a simulation is started and a client can connect to it, the server has to be set up. When started, the server opens a port for incoming connections from simulations in a separate thread. This port can be specified by the command-line argument `--port`. After the server has been initialized, a simulation can be started and try to connect to the server (see `serverport` and `serverip` parameters in simulation configuration file). To encapsulate the tasks of setting up a socket descriptor and listening for connections the class `TCPAcceptor` was implemented. By calling its `accept` method one can wait (blocking) for a connection attempt and get an instance of `TCPStream` back if the connection was established successfully.

```
TCPStream * incoming_stream = m_vis_acceptor->accept();
```

For the major part of client side communication the RIVLib is used. It is initialized by creating an object of type `rivlib::core` and `rivlib::ip_communicator` and then adding the communicator to the core.

```
this->m_rivcore = rivlib::core::create();
m_ip_comm = rivlib::ip_communicator::create(rivlib::ip_communicator::DEFAULT_PORT);
this->m_rivcore->add_communicator(m_ip_comm);
```

When a simulation connects to the server, `incoming_stream` points to a stream which handles all data transfer between a simulation and the server. An object of type `Visualization` is created and initialized with the stream object. At first, the name of the visualization and the resolution of images generated by it are received.

Secondly, a `rivlib::provider` is instantiated with the information received.

[1] *Virtual Private Network*

Now the `Visualization` object runs all further operations in a separate thread. It is responsible for receiving and forwarding (image) data to clients as well as sending commands from the client to the simulation. All communication is asynchronous.

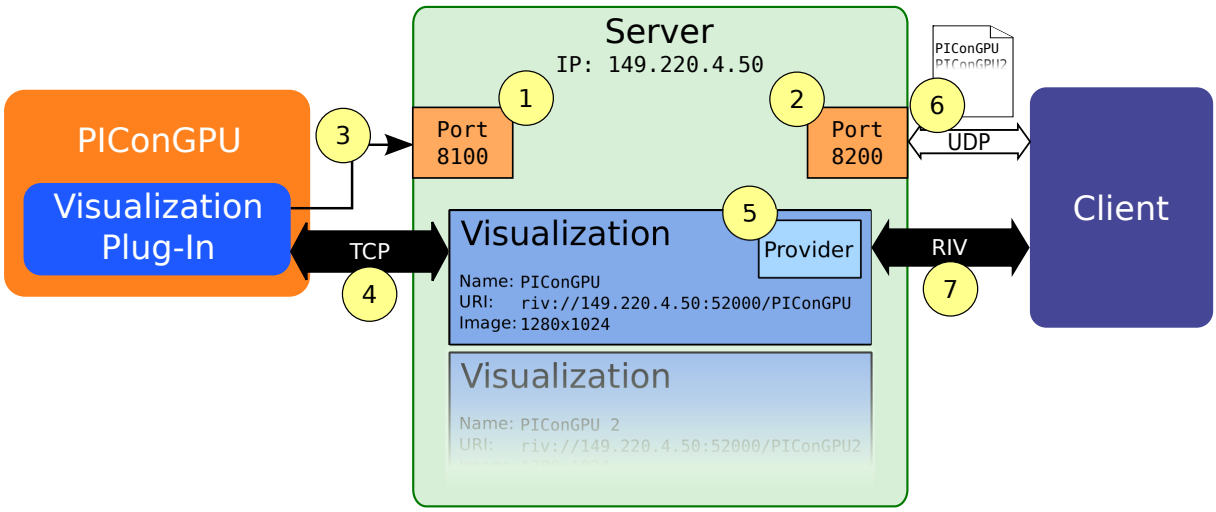
Clients can connect to visualizations running on the server by connecting to the visualizations RIVLib provider. To identify a provider it exposes a URI. Such a URI my look like this:

```
riv://149.220.4.50:52000/PIConGPUInSituVis
```

The first part specifies IP and port of the machine where the provider is located. The second part is the simulations name.

A list of simulation names (see `name` parameter in simulation configuration file) with their corresponding URI can be retrieved from the server. Therefore, the server runs another thread waiting for UDP messages asking for that list. If such a message arrives, the server stores the name and URI of a simulation into a UDP message and sends it to the client. As the UDP protocol does not guarantee message delivery in the order of sending, packing name and URI into one message was necessary to define the communication pattern and identify them on the client side. The port on which clients can query the server for available visualizations is defined at startup by the command-line argument `--infoport`.

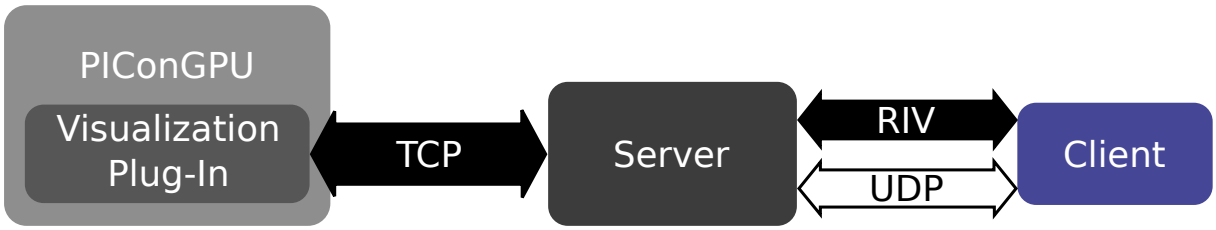
Figure 5.1 summarizes the steps performed by the plug-in, server and client to initiate an interactive visualization session.




1. The server opens a socket on port 8100 by using the TCPAcceptor class in a separate thread and waits for incoming connections.
2. The server opens a UDP socket waiting for clients to query the list of available visualizations.
3. The plug-in tries to connect to the server by using the TCPConnector class.
4. Once the connection attempt is accepted a TCP stream is established through the TCPStream class.
5. After name and image size have been received from the plug-in a new Visualization object containing a RIV-provider is instantiated and runs in a separate thread.
6. A client asks the server to send a list of all running simulations via UDP.
7. Eventually, the client connects via RIVLib to a running simulation. An image and control connection is established.

Figure 5.1: Steps executed to establish a two-way connection between a simulation and a client via the server.

# 6 The Client




The third component of our system is the client. While a basic in situ visualization client may only display the final image generated by the volume renderer, our client provides the user with a graphical interface through which he can interact with the simulation. On the one hand, the client should offer enough controls to adjust the visualization parameters like transfer function, viewpoint and clipping planes to match the users needs. On the other hand, it should not distract the scientist from the visualization and the task at hand.

A first client implementation offers full control over the features of the renderer. For example, the transfer function can be specified as a piecewise linear function in its RGB and opacity components (see Figure 6.1), allowing for precise tweaking. Anyhow, it turned out that this interface is too complicated for intuitive and efficient use. It is hard to make a cognitive connection between the sliders modified and the effect seen in the image. Jumping back and forth between the controls and the visualization is exhausting. The user may overlook subtle changes in the image and miss interesting features because he is distracted by the UI.

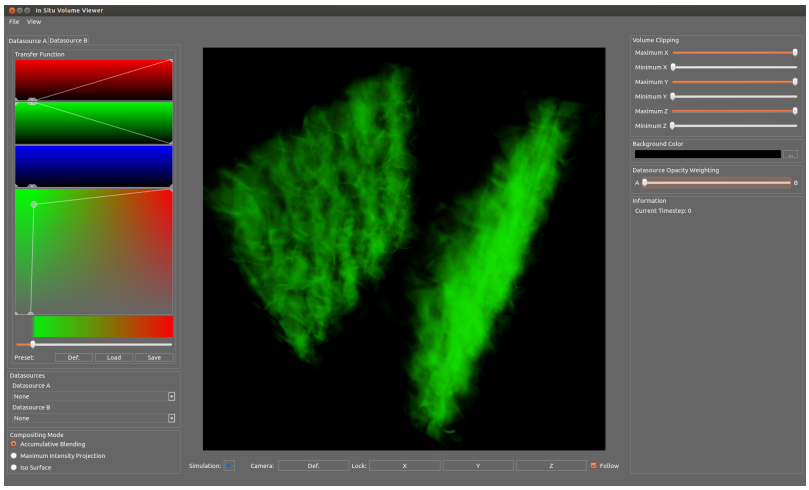


Figure 6.1: First user interface designed for testing the rendering capabilities of the CUDA Ray-Casting Plug-in.

Thus, a second client was designed to limit the possible interactions just enough to allow for simple yet

effective handling. It can be learned by exploration and experimentation. To focus the users attention on the visualization, all widgets can be partly or fully hidden (i.e. by pressing the `F1` to `F5` keys for hiding/showing each of the control panels and `F11` to hide the whole UI). Partly hiding the UI has the advantage of showing the configured parameters like the transfer functions color scale and the data source opacity weighting, while at the same time retaining as much screen space as possible for the visualization (compare Figures 6.2 and 6.3).

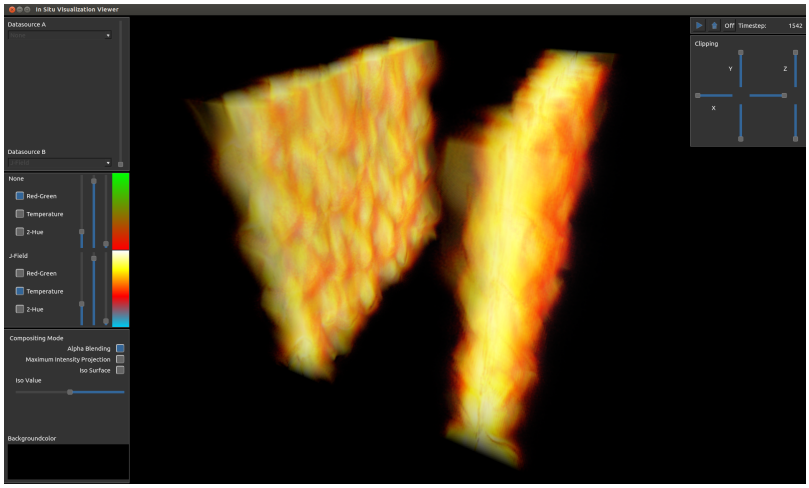



Figure 6.2: Simplified UI with five control panels.

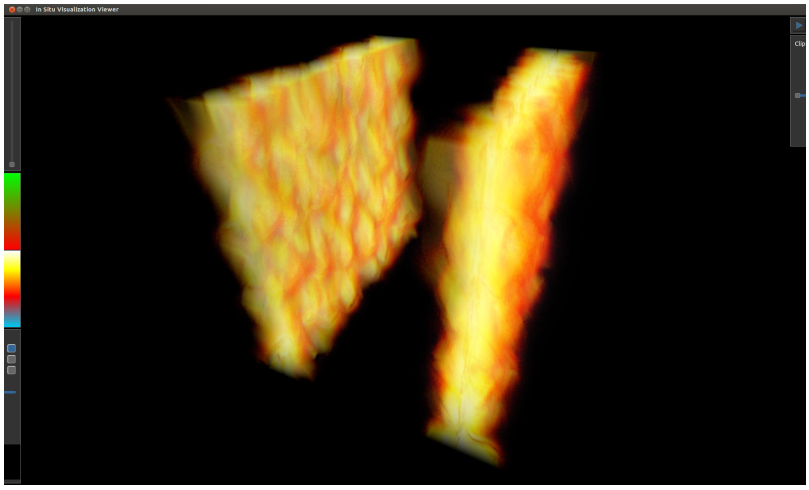



Figure 6.3: The partly hidden panels still display the most important settings.

On the top left the user can choose up to two data sources that should be visualized. The vertical slider next to the drop-down boxes determines their opacity weighting, which is given by the linear interpolation

$$C_{final} = w_A \cdot C_A + (1 - w_A) \cdot C_B,$$

where $w_A$ denotes the weight of the first data source, and $C_A$ and $C_B$ represent the fragment colors obtained from data sources A and B (or, more precisely transfer functions A and B). By weighting two data sources a relationship between them can be investigated. Figure 6.4 shows the electric field intensity together with the current flow weighted by a factor of $0.5$.

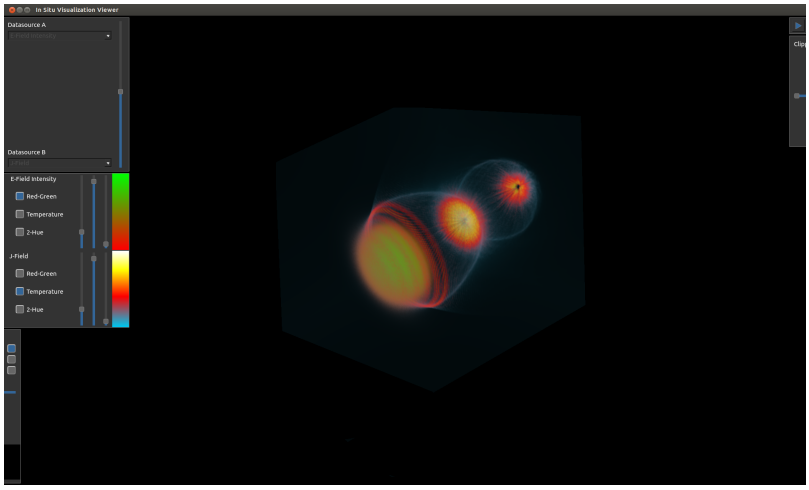



Figure 6.4: Weighting of the data sources *E-Field Intensity* with Red-Green and *J-Field* with Temperature color scales. At the front of the bubble the green tint shows the electric field of the laser pulse.

On the middle right, the user can choose one of three predefined color scales for each data source, i.e. a Red-Green, a Temperature and a Two-Hue scale. Together with the three sliders they configure a transfer function. A parameterized error-function underlies the transfer functions opacity mapping. Its slope as well as x- and y-offset can be adjusted. The leftmost slider determines the x-offset, the middle one the slope and the slider on the right the y-offset. Equation 6.1 shows the underlying formula.

$$\alpha(x) = \mathrm{erf}(s \cdot (x - o_x)) \cdot o_y + o_y \tag{6.1}$$

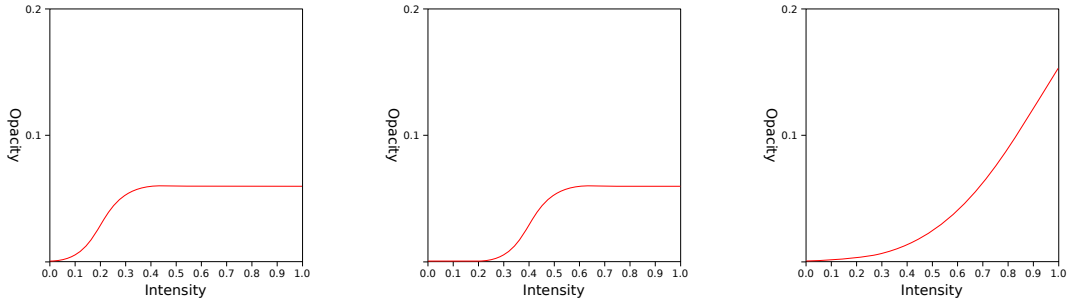
$s$ denotes the slope, $o_x$ the x-offset and $o_y$ the y-offset and scaling. Therewith, an opacity curve emphasizing high intensity values can be specified in an easy manner. Figure 6.5 depicts three different parameter configurations and the resulting images.

On the bottom right, the user interface offers controls for setting the compositing mode, a slider to set the iso value for the surface extraction mode and a button that shows and modifies the background color. To obtain optimal contrast in the image an arbitrary background color can be chosen (see Figure 6.6).

The top right panel provides the user with the ability to pause the simulation. This enables him to inspect the data of the current time step more precisely. The time step is displayed on the right. In between, buttons for reseting the camera to the default position and for writing the current image to disk are located. If image writing is enabled, every frame rendered will be stored in the simulation output folder on the cluster file system. This feature can be used to save images for later investigation or to communicate and present observations made during the simulation run.

Below, a slider setup for manipulating the six clipping planes on the negative and positive half-axes is placed. By manipulating these sliders, a region of interest can be extracted from the simulation area. In Figure 6.7 the red highlight shows that the simulation volume is clipped by a factor of $0.4$ on the positive z-axis.

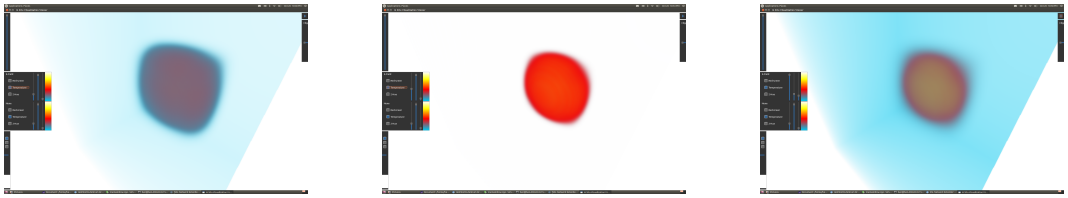
The viewpoint is changed by pressing the left, middle or right mouse button and moving the mouse (or, alternatively, by using the arrow keys). The client maintains a local camera model which is altered by




(a) $s = 9.5$, $o_x = 0.2$, $o_y = 0.03$

(b) $s = 9.5$, $o_x = 0.41$, $o_y = 0.03$

(c) $s = 2.1$, $o_x = 0.96$, $o_y = 0.14$



(d) High intensity values are mapped to a high opacity.

(e) The x-offset shifts the curve towards higher intensity values leaving small intensities with an $\alpha$ close to zero.

(f) If the slope is decreased, the transition between low and high opacities is more smooth, resulting in a volume of higher perceived density.

Figure 6.5: The opacity mapping of the transfer function is parameterized with different values for *slope*, *x-offset* and *y-offset*.

these input events. By pressing the left mouse button and moving the mouse the camera orbits around its focal point which is by default the center of the simulation volume. This input event is recognized by Qt and invokes a handler method. Therein, the local camera is modified and the new camera position is send to the simulation.

```
m_camera->orbitYawPitchRoll(mouse_delta_x, mouse_delta_y, 0.0f);
send_message(CameraPosition, 3 * sizeof(float), m_camera->getPosition());
```

A complete listing of steering commands exchanged between client, server and plug-in is attached in appendix A.

To connect to a visualization a gallery presents the available visualizations on the server with name and URI. The user can simply browse through that gallery by using the left and right arrow keys and connect to a visualization by pressing `Enter` (see Figure 6.8). By pressing the `F12` key the gallery can be shown and hidden.

As our client is intended to run on a variety of platforms and devices it is implemented in C++ and Qt. Thus, the client is portable to Linux, Windows and Mac OS.

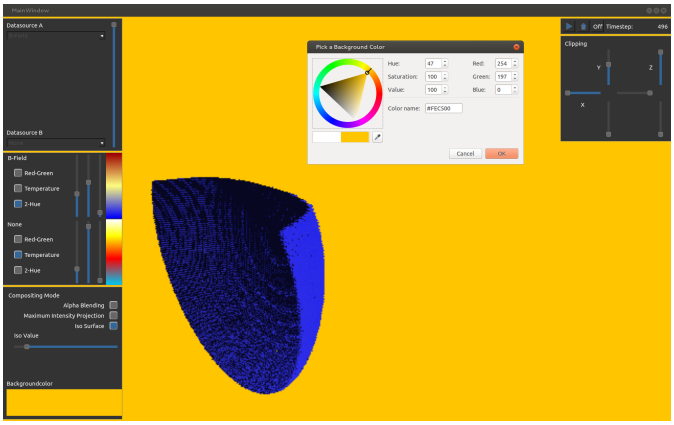



Figure 6.6: Enhancing the contrast between the used color scale and the background.

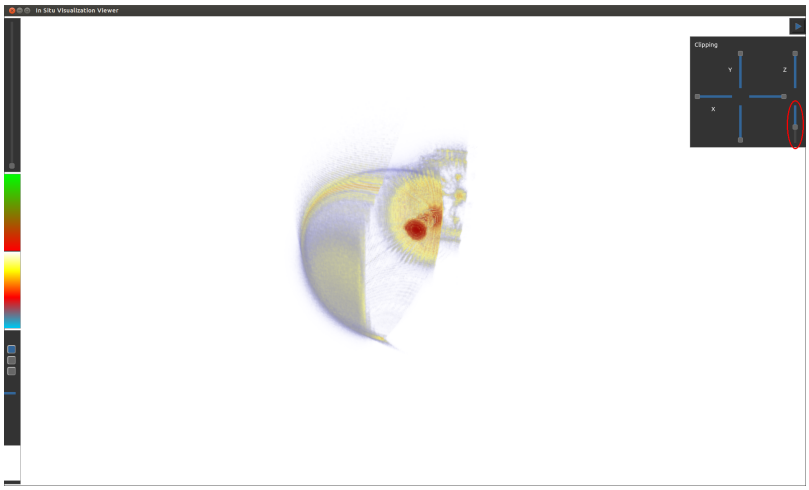



Figure 6.7: The simulation volume is clipped by a factor of $0.4$ in positive z-direction. This allows us to see the so-called injection (red tint in the visualization) in the Laser-Wakefield Acceleration simulation without any occlusion.

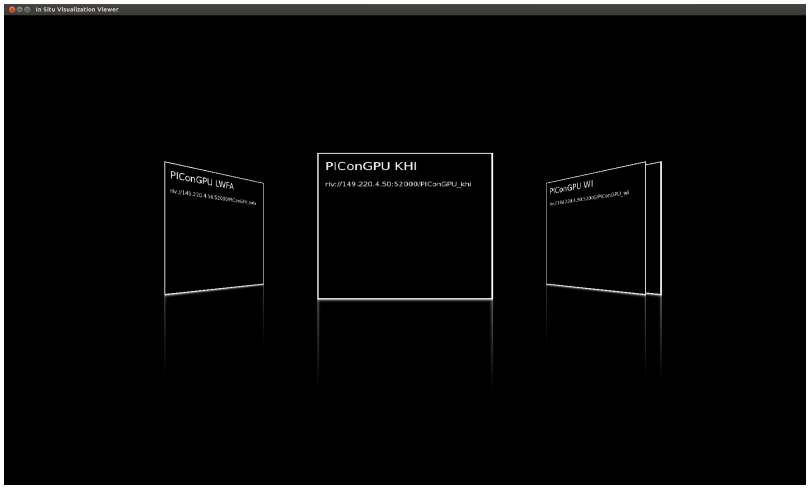



Figure 6.8: The simulation gallery presents the available visualizations as 3D slides. This is a first step towards a more immersive user interface.

# 7 Evaluation

The results presented in this chapter were conducted on the *Hypnos* cluster at HZDR. The simulation and visualization has been run on up to 16 compute nodes (i.e. 64 Kepler GPUs). In section 3.2 the cluster hardware is described in detail. We evaluated the performance of the volume rendering plug-in for different image and simulation grid sizes, measuring render and compositing time. Scalability was tested as well as the latency between the user sending a steering command from the client and receiving a new image.

We chose the *Laser-Wakefield Acceleration* experiment as simulation, observed from a fixed viewpoint. The camera position and focal point were set such that the volume fills the screen and an oblique birds eye view is obtained. This is the worst case in terms of rendering performance, because almost all rays hit the sub volume maximizing rendering costs. The transfer function was fixed to the Red-Green scale with the default opacity curve (see Figure 6.5(a)). As compositing scheme alpha blending was used.

The data being visualized was taken from the *J-Field* and *E-Field* data sources. The J-Field source uses nearest neighbor, the E-Field source trilinear interpolation.

The time needed to compute a time step is not included in the measured values.

## 7.1 Rendering Performance

At first we measured the rendering time for different image and simulation grid sizes as well as different GPU numbers. Therefore, the time needed for executing the ray-casting kernel was measured on each of the 64 GPUs over 100 time steps. Then, the rendering time of the slowest GPU was selected for each time step. Those 100 worst-case values were averaged.

### 7.1.1 Varying Image Size

By keeping the simulation grid size constant and changing the image resolution we tested the scalability for varying image sizes. The grid size chosen was $64 \times 128 \times 128$ cells per GPU. Table 7.1 shows the measured rendering times. Figure 7.1 summarizes the results. As expected the time needed for rendering increases with the image size, due to the higher number of rays cast through the volume and therewith more threads sampling the volume. As the E-Field data source performs trilinear interpolation much more data has to be read from memory to sample a single point inside the volume. This explains the high rendering time even for small image sizes. Resolutions above $1600 \times 1200$ pixels were not measured because we already left the region of interactive frame rates.

| Image Size | Time [ms] | |
|---|---|---|
| | J-Field | E-Field |
| $512 \times 512$ | 28.2 | 137.9 |
| $1024 \times 1024$ | 84.2 | 321.6 |
| $1280 \times 1024$ | 85.0 | 323.8 |
| $1600 \times 1200$ | 111.8 | 402.8 |
| $1920 \times 1200$ | 111.9 | - |
| $2560 \times 1600$ | 187.6 | - |
| $3840 \times 2160$ | 326.2 | - |

Table 7.1: Rendering time for different image resolutions.

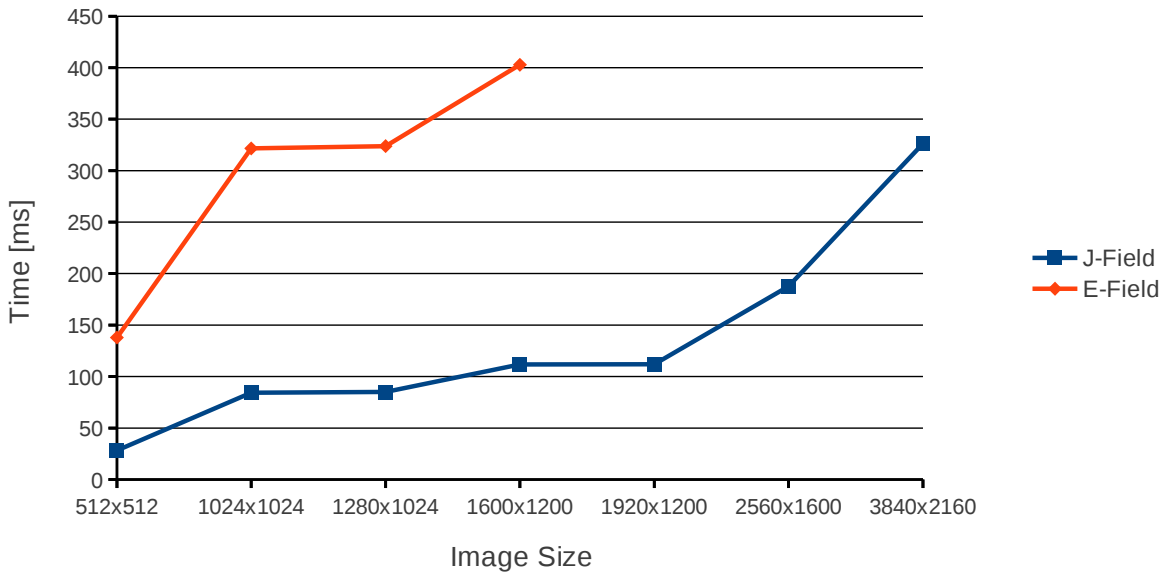



Figure 7.1: Increasing image resolution and corresponding rendering times of our volume renderer.

### 7.1.2 Varying Simulation Grid Size

With a fixed image size of $512 \times 512$ pixels we tested how the rendering time scales with increasing simulation grid size. Table 7.2 and Figure 7.2 show the timing values obtained by running the simulation on 64 GPUs. The time needed to generate an image for simulation volumes doubling in size increases linearly. This can be explained by the prolonged segment of the rays which intersects the (sub) volume. Thus, more samples have to be fetched from memory, mapped by the transfer function and composited to the final pixel color.

| Grid Size | $256 \times 256 \times 256$ | $256 \times 512 \times 256$ | $256 \times 1024 \times 256$ | $512 \times 1024 \times 256$ |
|---|---|---|---|---|
| Time [ms] | 12.2 | 20.0 | 29.2 | 36.7 |

Table 7.2: Rendering time for different grid sizes.

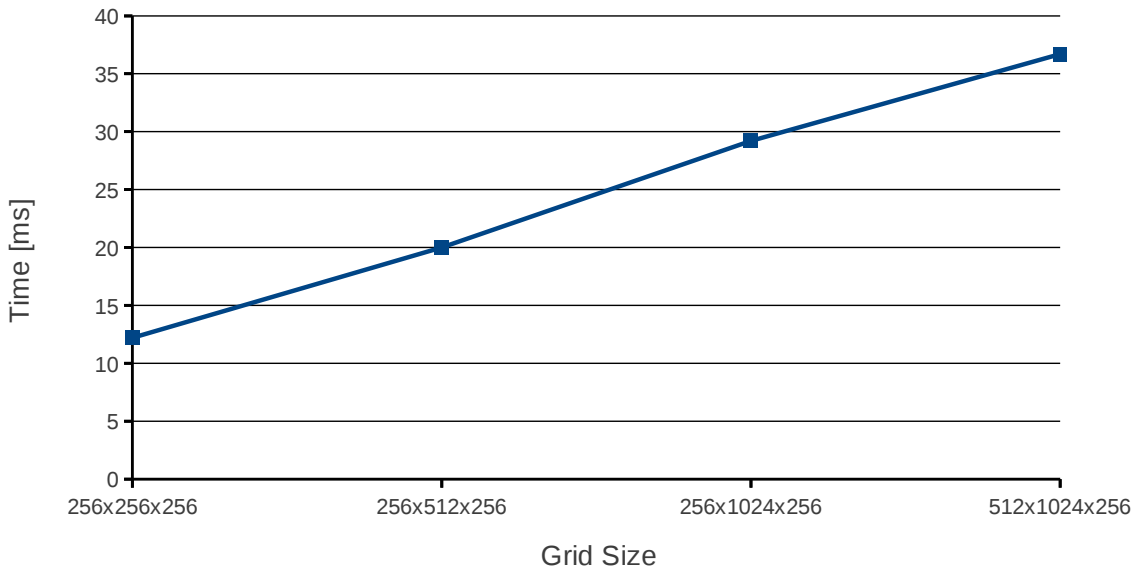



Figure 7.2: Scaling up grid size of the PIConGPU simulation and resulting rendering time.

### 7.1.3 Varying Number of GPUs

Keeping the image resolution and simulation grid size fixed we can observe how increasing the number of GPUs influences the rendering time. The image size for this test was $1024 \times 1024$. The total simulation grid had dimensions of $256 \times 512 \times 128$ cells. Solving a problem of fixed size with an increasing amount of processors is known as *strong scaling*. As can be seen from Table 7.3 and Figure 7.3 our system exposes an almost linear scaling. This is explained by the smaller sub volume size per GPU, so that the segment of the rays intersecting the volume become shorter and thus less samples have to be taken.

| # GPUs | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| Time [ms] | 207.1 | 145.7 | 92.3 | 75.7 | 50.4 |

Table 7.3: Rendering time for different numbers of GPUs.

## 7.2 Compositing Performance

After the sub images are generated by each GPU individually, they have to be copied to main memory and combined into a final image. These stages are known as *readback* and compositing. To assess how our system handles the compositing of high image resolutions on an increasing number of processes, we measured the compositing time for configurations of four to 64 GPUs with image sizes of $1024 \times 1024$ and $1600 \times 1200$ pixels. It turned out that the compositing time is only dependent on the image size, needing more time at higher resolutions. For varying numbers of processes it stays almost the same (see Table 7.4 and Figure 7.4). The higher compositing time for larger image sizes is explained by the increasing readback time to copy a sub image from the GPU to system memory, the time needed for exchanging more image data over the network and lastly by the additional computations needed to

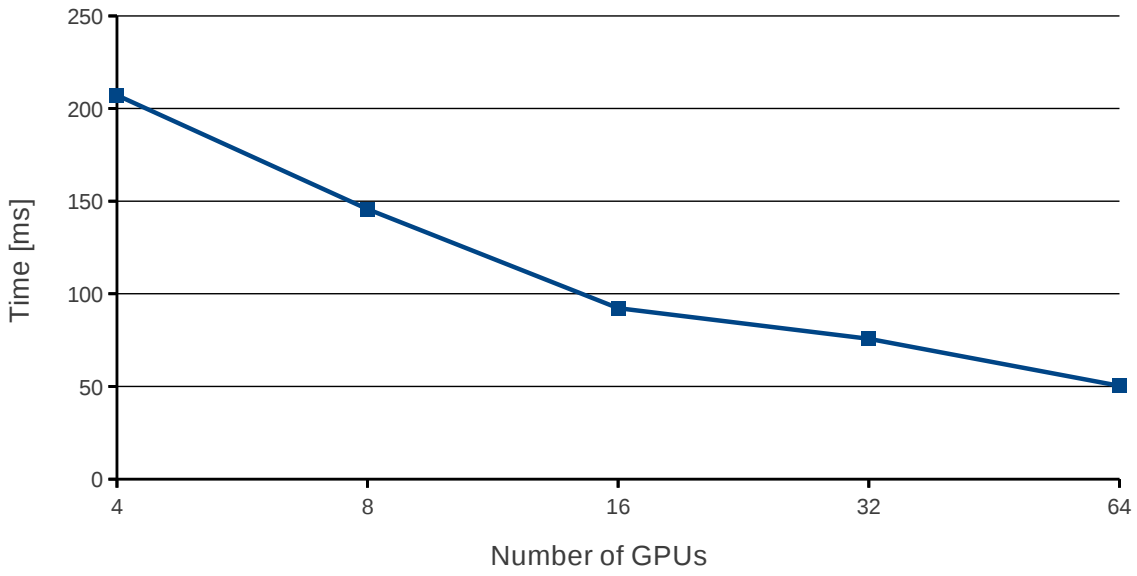



Figure 7.3: Strong scaling measured by keeping image and total grid size constant.

compose more pixels (i.e. blending or depth comparison). The stable compositing time for varying process numbers is achieved by the optimized algorithms implemented in IceT. As shown in section 2.5 those algorithms try to distribute the compositing work over all processes and minimize communication. Moreland et al. [MKPH11] have proved this behavior on much higher processor counts.

| Image Size | # GPUs | Time [ms] |
|---|---|---|
| $1024 \times 1024$ | 4 | 30.7 |
| | 8 | 26.1 |
| | 16 | 28.6 |
| | 32 | 28.1 |
| | 64 | 30.5 |
| $1600 \times 1200$ | 4 | 52.1 |
| | 8 | 46.8 |
| | 16 | 51.2 |
| | 32 | 48.0 |
| | 64 | 51.4 |

Table 7.4: Compositing time for different image resolutions and GPU numbers.

## 7.3 Interactivity

Lastly, we want to assess if our system is usable for interactive viewing and steering of simulations. Therefore, we provide some guidance values for the time between the client sending a command to the renderer and receiving an updated image back. The measured latency is, anyhow, only a rough guideline for what the user has to expect. It depends on a variety of factors, including the kind of network between client and server (Ethernet, Wifi) and its utilization as well as the cluster load.

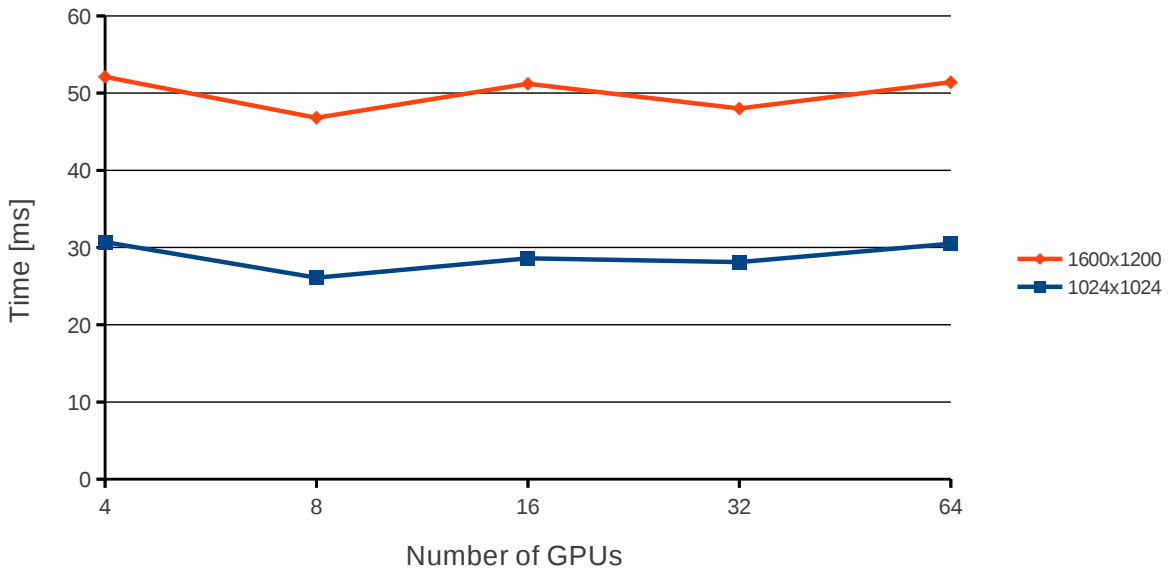



Figure 7.4: Time to read back and composite the sub images of all GPUs for different process counts and image resolutions.

Our test scenario runs a simulation on 64 GPUs which simulate a grid of $256 \times 1024 \times 256$ cells and renders images of $512 \times 512$ respectively $1024 \times 1024$ pixels. Running the client on a machine inside the corporate network of the HZDR, latencies around $100\,\mathrm{ms}$ to $400\,\mathrm{ms}$ were measured. The user experiences fast feedback for his actions, especially when several commands in a row are sent, e.g. when moving the camera. Then, transmission can overlap with image rendering and delivery. Smooth motions and a responsive UI are experienced. The four times higher resolution of $1024 \times 1024$ incurred latencies of $300\,\mathrm{ms}$ to $600\,\mathrm{ms}$.

From the university wireless network latencies of $700\,\mathrm{ms}$ to $1200\,\mathrm{ms}$ for $512 \times 512$ pixels and $2600\,\mathrm{ms}$ to $3600\,\mathrm{ms}$ for $1024 \times 1024$ allow users to observe the progress of a simulation. Interactive viewing is still possible even though the delay is well noticeable.

## 7.4 Conclusion

The performance evaluation shows a near ideal strong scaling for our volume renderer, similar to the simulations strong scaling behavior [HSW+10].

The major bottleneck in our CUDA ray-casting algorithm is reading simulation data from global memory to a threads local memory when sampling the volume. The cache used for global memory reads is not optimized for spatial but linear memory address locality. Thus, cache misses are very frequent. Using texture memory could improve the memory bandwidth through better cache utilization considerably. Anyhow, this was not possible due to alignment of data structures (e.g. no `float3` textures are supported) and flexibility (i.e. instantiating template classes holding the simulation data with various data types). Invasive changes to the simulation code would be necessary.

Moreover, the computational complexity of the ray-casting algorithm is relatively low when compared to the necessary memory operations. This means, the GPU cannot hide the high latency involved when reading data from global memory – as it normally would by scheduling warps such that the multiprocessors are kept busy.

Especially for high image resolutions the network connecting server and client is the limiting factor. In section 8.2 we discuss some possibilities to improve frame rates and shorten image delivery time and latency.

# 8 Summary & Outlook

The goal of this thesis was to make the reader aware of the rising importance to find novel ways for data analysis, i.e. through in situ processing and visualization. Pressing issues like increasing power consumption and data transfer bottlenecks already limit the effectively usable computational power of HPC systems. Moving towards exa-scale computing these issues will and do already determine the pace of scientific progress.

By example of PIConGPU we showed how an existing simulation code is extended to provide interactive in situ visualization. A tightly-coupled approach to avoid data replication and movement was followed. Thereby, the bottlenecks of disk storage and network bandwidth are eliminated or significantly reduced.

## 8.1 Results

Due to the aforementioned bottlenecks previous visualizations only allowed for viewing a 2D slice of the 3D simulation. With our system it is possible to observe the full simulation volume as a whole, giving an impression of large scale phenomena as well as the ability to investigate particular regions of interest by extracting them through volume clipping and transfer function adjustment.

By visualizing two data sources in the same image it is possible to detect relationships among different physical quantities such as the influence of particle motion on electromagnetic fields an vice versa. Observing them separately makes it much harder to see immediate correlations.

Furthermore, enabling scientists to inspect a simulation while it is running closes the loop between configuring a parameter set, executing the simulation and analyzing the resulting data. Now, there is no need to wait until a simulation run ends just to notice that the desired phenomena were not observable or an error in the configuration corrupted the results. Ill-behaving jobs can be killed immediately and restarted with a correct parameter set. This increases turnover rates and optimizes utilization of the personnel and computational resources available.

The exchange of ideas and the discussion of observations plays a central role in research. Our system offers the possibility to view the same visualization from multiple clients, thus, allowing for collaborative work even from distant locations.

Even though we have seen that very high image resolutions are not feasible for interactive visualization they can be employed to produce high quality images and videos for publications and presentations.

Lastly, the system is not only of use for scientists but also provides developers with another method of debugging the simulation code. Errors caused by numerical instabilities or communication among GPUs might be missed by a two-dimensional view.

Figures 8.1, 8.2, 8.3, 8.4 and 8.5 show images of the Laser-Wakefield Acceleration, the Weibel and the Kelvin-Helmholtz Instability generated by our volume renderer.

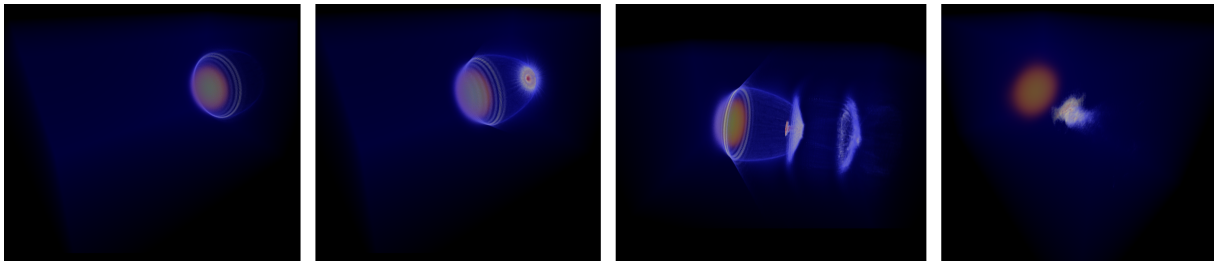


Figure 8.1: Showing the bubble and laser pulse moving though the plasma. Time step 1000, 1500, 7100 and 11100 show the temporal development. At the last time step depicted the laser has already left the gas.

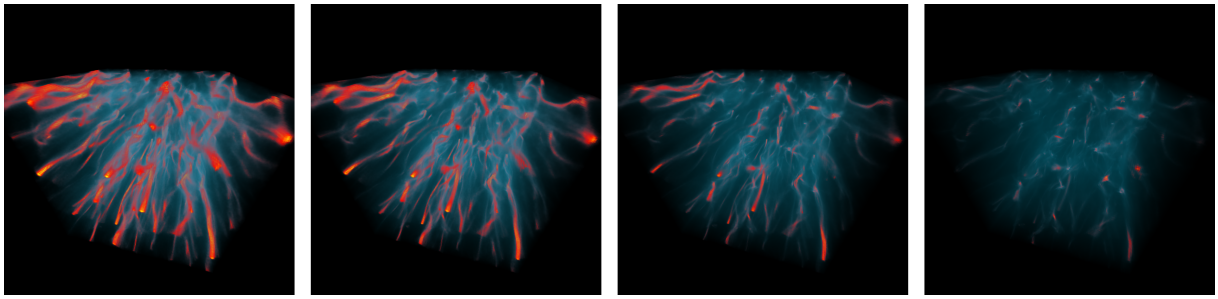


Figure 8.2: The Weibel instability at time steps 8926, 8948, 8970 and 8992.

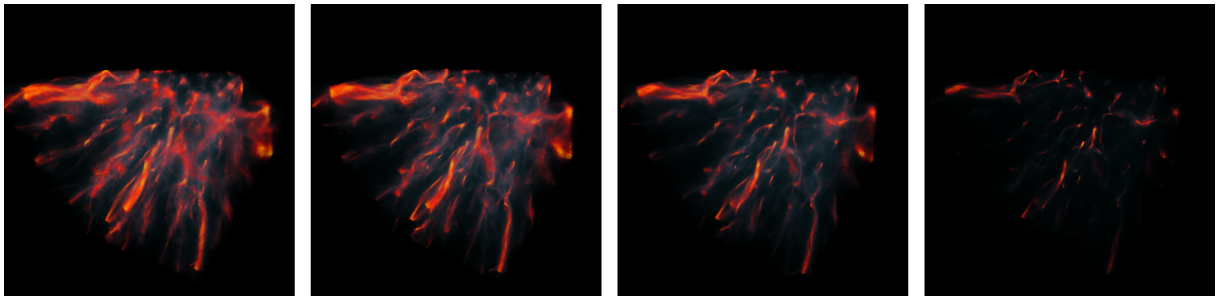


Figure 8.3: The Weibel instability at time steps 9210, 9276, 9230 and 9364 showing the typical filament-like beams.

## 8.2 Future Work

The software developed in this thesis can be extended in various ways, with respect to functional and non-functional aspects.

Extending the renderer to support three (or more) data sources at once might allow for additional investigative possibilities. However, the problem of overloading the image with information has to be tackled by finding a method to weight and distinguish the different data sets visible. This can be achieved by employing two-level volume rendering [HMIBG01], introducing local and global compositing schemes

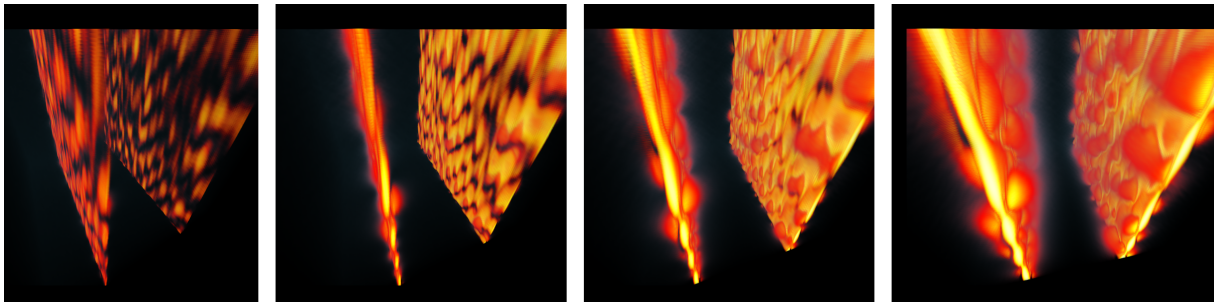


Figure 8.4: The Kelvin-Helmholtz instability at time steps 950, 1050, 1150 and 1230. Turbulent flow at the plasma interfaces grow stronger over time.

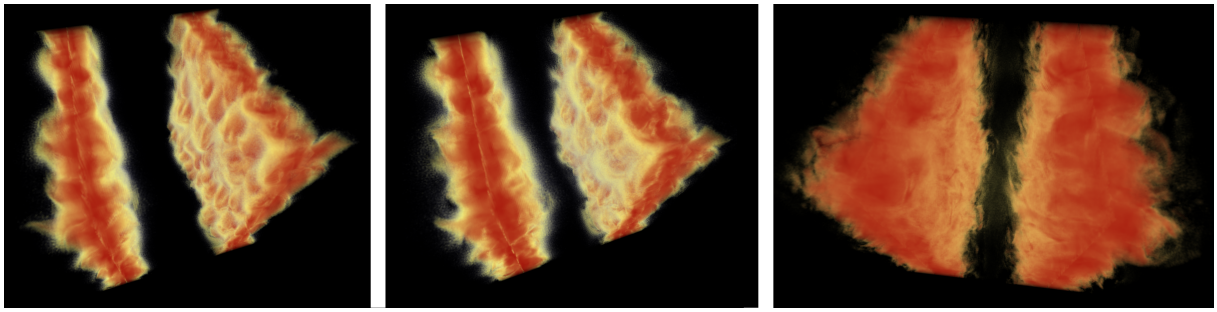


Figure 8.5: The KHI using the two-hue color scale at time steps 1550, 1700 and 2270.

as well as adding more distinctive predefined transfer functions. Automatic generation of transfer functions is still an active topic of research [RBB+11, ZT09]. Once a good transfer function is found it can be stored together with the parameter set used to run the simulation. The image quality can be improved by using higher order interpolation schemes (e.g. B-Spline and Catmull-Rom) to reconstruct sampling values and estimate gradients. Anyhow, to preserve interactive frame rates this has to come with an optimized ray-casting implementation.

To improve the speed of rendering and compositing we have to exploit the properties and features the CUDA platform exposes. An intelligent cache pre-fetching data from global to shared memory using coalesced reads can decrease rendering time significantly, especially if a data value is read several times (which is the case for larger filter kernels). Such a cache can incorporate knowledge about ray direction and coherence among neighboring rays.

For the final image compositing we utilized the IceT library, which performs the blending respectively depth combination for each pixel on the CPU. This requires to copy the pixel and depth buffer from the GPUs global memory to main memory. By using a feature called **GPUDirect**[1] first introduced in 2010 the GPU can directly communicate with the InfiniBand driver making it possible to send data from one GPU to another, avoiding one copy of the data in main memory. As each pixel of the final image can be composited independently, the GPU could perform this task in a highly parallel manner. Combined with compression algorithms as suggested by Lietsch [LM07] and O'Neil [OB11] the compositing time might be reduced significantly, even on low-bandwidth networks.

Beyond in situ visualization, a broader possibility is in situ processing, including data processing and

[1] https://developer.nvidia.com/gpudirect

analysis. In situ processing will allow scientists to study the full extent of the data generated by their simulations [YWG+10]. Extracting and tracking features spotted by themselves or by intelligent (learning) algorithms can greatly enhance the insights won.

# Bibliography

[Bli94] Jim Blinn. Image Compositing-Theory: The mathematics and a new derivation of the use of alpha in image blending, and a justification for using associated colors. *IEEE Computer Graphics and Applications*, 14(5), September 1994.

[Bus13] Michael Bussmann. PIConGPU Website. http://picongpu.hzdr.de, October 2013.

[BWH+10] Heiko Burau, René Widera, Wolfgang Hoenig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Tomas E. Cowan, Roland Sauerbrey, and Michael Bussmann. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. In *IEEE Transactions on Plasma Science*, volume 38, pages 2831 – 2839. IEEE, October 2010.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, VVS '94, pages 91–98, New York, NY, USA, 1994. ACM.

[CGM+06] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James Ahrens, and Jean Favre. Remote large data visualization in the paraview framework. In *Proceedings of the 6th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'06, pages 163–170, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[CN94] Timothy J. Cullip and Ulrich Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical report, Chapel Hill, NC, USA, 1994.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, June 1988.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, pages 9–16, New York, NY, USA, 2001. ACM.

[EMP09] Stefan Eilemann, Maxim Makhinya, and Renato Pajarola. Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, May 2009.

[EP07] Stefan Eilemann and Renato Pajarola. Direct send compositing for parallel sort-last rendering. In *Proceedings of the 7th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'07, pages 29–36, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[ERC06] Aurelien Esnard, Nicolas Richart, and Olivier Coulaud. A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations. In *Proceedings of the 10th IEEE*

*international symposium on Distributed Simulation and Real-Time Applications*, DS-RT '06, pages 7–14, Washington, DC, USA, 2006. IEEE Computer Society.

[FCS+10] Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large data visualization on distributed memory multi-GPU clusters. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 57–66, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[For09] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, September 2009.

[Fri59] Burton D. Fried. Mechanism for Instability of Transverse Plasma Waves. *Physics of Fluids (1958-1988)*, 2(3):337–337, 1959.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Har55] Francis H. Harlow. A Machine Calculation Method for Hydrodynamic Problems, November 1955.

[HHN+02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 693–702, New York, NY, USA, 2002. ACM.

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

[HMIBG01] H. Hauser, L. Mroz, G. Italo Bischi, and E. Groller. Two-level volume rendering. *Visualization and Computer Graphics, IEEE Transactions on*, 7(3):242–252, 2001.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[Hsu93] William M. Hsu. Segmented ray casting for data parallel volume rendering. In *Proceedings of the 1993 symposium on Parallel rendering*, PRS '93, pages 7–14, New York, NY, USA, 1993. ACM.

[HSW+10] Wolfgang Hönig, Felix Schmitt, René Widera, Heiko Burau, Guido Juckeland, M.S. Müller, and Michael Bussmann. A Generic Approach for Developing Highly Scalable Particle–mesh Codes for GPUs. *SAAHPC, USA, Jul*, 2010.

[JH11] Oliver Jato and André Hinkenjann. Volt: Interaktives Volumenrendering mit CUDA. In Sina Mostafawy Christian-A. Bohn, editor, *8. Workshop Virtuelle und Erweiterte Realität der GI-Fachgruppe VR/AR*, pages 73–84. Shaker Verlag, 2011.

[KPB12] Thomas Kroes, Frits H. Post, and Charl P. Botha. Exposure Render: An Interactive Photo-Realistic Volume Rendering Framework. *PLoS ONE*, 7(7):e38586, 07 2012.

[KPH+10] Wesley Kendall, Tom Peterka, Jian Huang, Han-Wei Shen, and Robert Ross. Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings of the 10th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'10, pages 101–110, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[KW03] J. Kruger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[KY13] Akira Kageyama and Tomoki Yamada. An Approach to Exascale Visualization: Interactive Viewing of In-Situ Visualization. *CoRR*, abs/1301.4546, 2013.

[Lev88] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 451–458, New York, NY, USA, 1994. ACM.

[LM07] Stefan Lietsch and Oliver Marquardt. A CUDA-supported approach to remote rendering. In *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part I*, ISVC'07, pages 724–733, Berlin, Heidelberg, 2007. Springer-Verlag.

[LRN96] Tong-Yee Lee, C. S. Raghavendra, and John B. Nicholas. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, September 1996.

[LSWP09] M. Lazar, R. Schlickeiser, R. Wielebinski, and S. Poedts. Cosmological Effects of Weibel-Type Instabilities. *The Astrophysical Journal*, 693(2):1133, 2009.

[Ma09] Kwan-Liu Ma. In Situ Visualization at Extreme Scale: Challenges and Opportunities. *IEEE Comput. Graph. Appl.*, 29(6):14–19, November 2009.

[McC88] B. H. McCormick. Visualization in scientific computing. *SIGBIO Newsl.*, 10(1):15–21, March 1988.

[MCEF94] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, July 1994.

[MI97] Kwan-Liu Ma and Victoria Interrante. Extracting feature lines from 3D unstructured grids. In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 285–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[MKPH11] Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 25:1–25:10, New York, NY, USA, 2011. ACM.

[MMD08] Stéphane Marchesin, Catherine Mongenet, and Jean-Michel Dischler. Multi-GPU sort-last volume visualization. In *Proceedings of the 8th Eurographics conference on Parallel*

*Graphics and Visualization*, EG PGV'08, pages 1–8, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[Mor11] Kenneth Moreland. IceT Users' Guide and Reference. Sandia report, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, August 2011.

[MPHK94] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, July 1994.

[MT03] Kenneth Moreland and David Thompson. From Cluster to Wall with VTK. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 5–, Washington, DC, USA, 2003. IEEE Computer Society.

[Neu94] Ulrich Neumann. Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Comput. Graph. Appl.*, 14(4):49–58, July 1994.

[NVi13] NVidia. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, July 2013.

[OB11] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 7:1–7:7, New York, NY, USA, 2011. ACM.

[Owe13] G. Scott Owen. Ray - Box Intersection. http://www.siggraph.org/education/materials /HyperGraph/raytrace/rtinter3.htm, November 2013.

[PGR+09] Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 4:1–4:10, New York, NY, USA, 2009. ACM.

[RBB+11] M. Ruiz, A. Bardera, I. Boada, I. Viola, M. Feixas, and M. Sbert. Automatic Transfer Functions Based on Informational Divergence. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):1932–1941, 2011.

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, pages 109–118, New York, NY, USA, 2000. ACM.

[SFLS00] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, pages 97–108, New York, NY, USA, 2000. ACM.

[Sou12] Jérôme Soumagne. *An In-situ Visualization Approach for Parallel Coupling and Steering of Simulations through Distributed Shared Memory Files*. PhD thesis, L'UNIVERSITE BORDEAUX I, December 2012.

[ST90] Peter Shirley and Allan Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. In *Computer Graphics*, pages 63–70, 1990.

[Taj08] Toshiki Tajima. Laser wakefields: Bringing accelerators down to size. *Nature Photonics*, 2(9):526–527, 2008.

[TD79] T. Tajima and J. M. Dawson. Laser Electron Accelerator. *Phys. Rev. Lett.*, 43:267–270, Jul 1979.

[Tho71] William Thomson. XLVI. Hydrokinetic solutions and observations. *Philosophical Magazine Series 4*, 42(281):362–377, 1871.

[VBS+11] H.T. Vo, J. Bronson, B. Summa, J.L.D. Comba, J. Freire, B. Howe, V. Pascucci, and C.T. Silva. Parallel visualization on large clusters using MapReduce. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 81 –88, oct. 2011.

[Wei59] E.S. Weibel. Spontaneously Growing Transverse Waves in a Plasma Due to an Anisotropic Velocity Distribution. *Physical Review Letters*, 2:83–84, February 1959.

[Wes91] Lee Alan Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. PhD thesis, Chapel Hill, NC, USA, 1991. UMI Order No. GAX92-08005.

[WFM11] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[Wik13] Wikipedia. Message passing. http://en.wikipedia.org/wiki/Message_passing, November 2013.

[YWG+10] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In Situ Visualization for Large-Scale Combustion Simulations. *IEEE Comput. Graph. Appl.*, 30(3):45–57, May 2010.

[YWM08] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 48:1–48:11, Piscataway, NJ, USA, 2008. IEEE Press.

[ZT09] Jianlong Zhou and M. Takatsuka. Automatic transfer function generation using contour tree controlled residue flow model and color harmonics. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1481–1488, 2009.

[Züh11] Lukas Zühl. Visualisierung von Laser-Plasma-Simulationen, 2011.

# List of Figures

1.1 Processor-Memory Gap (after [HP12]) . . . 3
1.2 Comparison between post-, co- and in situ processing. Both, post- and co-processing involve the transfer of raw data. In situ processing provides the most scalable solution to deal with large scale simulation data (after [YWG+10]). . . . 7

2.1 NVidia Tesla K20 . . . 9
2.2 Grid of Thread-blocks (from [NVi13]) . . . 10
2.3 CUDA Memory Hierarchy (from [NVi13]) . . . 11
2.4 MPI procedures for collective communication . . . 12
2.5 The PIConGPU simulation cycle. . . . 14
2.6 Hybrid parallelism of CUDA threads and MPI processes . . . 15
2.7 GPU Subdomains with Core (orange) and Border (blue) area . . . 15
2.8 The simulation grid is distributed on four GPUs. The observed area framed in blue has the size of two GPUs but spans three. When the window hits the end of GPU three, number one will be reinitialized and attached to the end of the simulation area. . . . 18
2.9 Development of a bubble regime by separation of electrons and ions after the Laser-Wakefield Acceleration principle [Züh11]. . . . 18
2.10 An example of the Kelvin-Helmholtz Instability in Saturn's atmosphere. At the boundary of two latitudinal bands turbulences curl repeatedly. Photo taken with Cassini spacecraft narrow image camera. . . . 19
2.11 A cell formed by eight voxels. . . . 20
2.12 Light interactions from left to right: emission, absorption, in-scattering and out-scattering (after [HKRs+06]) . . . 20
2.13 Compositing schemes Maximum Intensity Projection, Iso-Surface and Alpha Blending. . . 23
2.14 Ray-casting (after [HKRs+06]) . . . 24
2.15 Texture Slicing for perspective projection . . . 25
2.16 Shear-warp volume rendering. . . . 25
2.17 Binary-Swap image compositing with four processes. . . . 27

3.1 HPC cluster and network architecture . . . 31
3.2 The three components and their communication protocols. . . . 32

4.1 Interactive visualization loop. . . . 38

4.2 Parallelization of volume ray-casting in image space. A grid of $6 \times 4$ blocks with $4 \times 4$ threads is configured. The final image resolution is $24 \times 16$ pixels. A thread can calculate the coordinates of the pixel he is responsible from the block dimensions and his thread index. Thus, thread $(1, 2)$ in block $(0, 1)$ renders pixel $(1, 6)$. . . . . 40
4.3 Oversampling rays at the border of sub volumes can cause artifacts (4.3(a) and 4.3(c)). This can be avoided by maintaining a constant sampling interval over sub volume borders (4.3(b) and 4.3(d)). . . . . 42
4.4 The three different compositing modes by example of the same data and time step. . . . 43
4.5 Definition of the i-th ray segment. *d* denotes the length of the sampling interval, $\mathbf{x}(t)$ a sampling position on the ray determined by the ray parameter $t$ and $s(\mathbf{x})$ the scalar value at position $\mathbf{x}$. . . . . 44
4.6 Pre-integration transforms the original transfer function into a two-dimensional lookup table. High frequencies in the transfer function are reproduced giving a better visual result. 45
4.7 Sub volumes with their ranks and the resulting depth ordering. . . . . 48
4.8 Eight sub images and the resulting final image. . . . . 49

5.1 Steps executed to establish a two-way connection between a simulation and a client via the server. . . . . 55

6.1 First user interface designed for testing the rendering capabilities of the CUDA Ray-Casting Plug-in. . . . . 57
6.2 Simplified UI with five control panels. . . . . 58
6.3 The partly hidden panels still display the most important settings. . . . . 58
6.4 Weighting of the data sources *E-Field Intensity* with Red-Green and *J-Field* with Temperature color scales. At the front of the bubble the green tint shows the electric field of the laser pulse. . . . . 59
6.5 The opacity mapping of the transfer function is parameterized with different values for *slope*, *x-offset* and *y-offset*. . . . . 60
6.6 Enhancing the contrast between the used color scale and the background. . . . . 61
6.7 The simulation volume is clipped by a factor of $0.4$ in positive z-direction. This allows us to see the so-called injection (red tint in the visualization) in the Laser-Wakefield Acceleration simulation without any occlusion. . . . . 61
6.8 The simulation gallery presents the available visualizations as 3D slides. This is a first step towards a more immersive user interface. . . . . 61

7.1 Increasing image resolution and corresponding rendering times of our volume renderer. . 64
7.2 Scaling up grid size of the PIConGPU simulation and resulting rendering time. . . . . 65
7.3 Strong scaling measured by keeping image and total grid size constant. . . . . 66
7.4 Time to read back and composite the sub images of all GPUs for different process counts and image resolutions. . . . . 67

8.1 Showing the bubble and laser pulse moving though the plasma. Time step 1000, 1500, 7100 and 11100 show the temporal development. At the last time step depicted the laser has already left the gas. . . . . 70

8.2 The Weibel instability at time steps 8926, 8948, 8970 and 8992. . . . . 70
8.3 The Weibel instability at time steps 9210, 9276, 9230 and 9364 showing the typical filament-like beams. . . . . 70
8.4 The Kelvin-Helmholtz instability at time steps 950, 1050, 1150 and 1230. Turbulent flow at the plasma interfaces grow stronger over time. . . . . 71
8.5 The KHI using the two-hue color scale at time steps 1550, 1700 and 2270. . . . . 71

# List of Tables

2.1 Maxwell Equations . . . . . 14

7.1 Rendering time for different image resolutions. . . . . 64
7.2 Rendering time for different grid sizes. . . . . 64
7.3 Rendering time for different numbers of GPUs. . . . . 65
7.4 Compositing time for different image resolutions and GPU numbers. . . . . 66

A.1 Message Identifiers. . . . . 86

# A Message Protocol

A message consists of an identifier (ID), a length $l$ and an array of $l$ bytes containing the message data. This datatype is declared as a C++ structure:

```
struct Message
{
        uint32_t id;
        uint32_t length;
        void * data;
};
```

All message IDs that are used for communication are enumerated by `enum MessageID` in the file `message_ids.hpp`.

| | |
|---|---|
| `NoMessage` | Token indicating that no messages were available. Used for non-blocking communication. |
| `Image` | Indicates that the message contains image data. |
| `ImageSize` | Message of two integers specifying the final image width and height. |
| `TimeStep` | Message of one integer containing the current time step. |
| `VisName` | Message of a string storing the name of the simulation. |
| `CameraPosition` | The message contains three floats determining the camera position in $x, y, z$. |
| `CameraFocalPoint` | Message of three floats determining the look at point of the camera. |
| `Weighting` | Message of one float with the weighting of the two data sources. |
| `SimPlay` | Token indicating to continue the simulation. |
| `SimPause` | Token indicating to pause the simulation. |
| `CloseConnection` | Token indicating that a connection is about to be closed. |
| `TransferFunctionA` | Message containing an array of float4s describing a color lookup table for the first transfer function. |
| `TransferFunctionB` | Message containing an array of float4s describing a color lookup table for the second transfer function. |
| `Clipping` | Message of six floats in range $[0; 1]$ defining a clipping box with minimum and maximum $x, y, z$ coordinates. |
| `BackgroundColor` | Message of three floats with RGB values in range $[0; 1]$ that determine the background color of the final image. |
| `AvailableDataSource` | Message containing the name of an available data source as a string. |
| `DataSourceA` | Message with a name string setting the first data source to be visualized. |
| `DataSourceB` | Message with a name string setting the second data source to be visualized. |
| `RequestDataSources` | Token to ask the simulation which data sources it provides for visualization. |
| `ListVisualizations` | Token used to query the visualization server for a list of available visualizations. |
| `VisListLength` | Message containing an integer with the count of visualizations available on the server. |
| `VisRivURI` | Message with the URI of a visualization as string. |
| `CompositingModeAlphaBlend` | Token indicating to set alpha blending as compositing mode. |
| `CompositingModeIsoSurface` | Token indicating to set iso surface as compositing mode. |
| `CompositingModeMIP` | Token indicating to set maximum intensity projection as compositing mode. |
| `VisibleSimulationArea` | Message of six floats describing the minimum and maximum coordinates $x, y, z$ of the visible simulation area. |
| `IsoSurfaceValue` | Message of a single float in range $[0; 1]$ that is the normalized iso surface value. |
| `PngWriterOn` | Token indicating to start writing images to disk. |
| `PngWriterOff` | Token to stop writing images to disk. |

Table A.1: Message Identifiers.

# B Running an Interactive In Situ Visualization

The first step is to run the server – possibly configured with certain port numbers. If neither the path to the RIVLib library file is part of the `LD_LIBRARY_PATH` environment variable nor the library files are installed in the standard directories we have to specify their location.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/rivlib

./server --port 8100 --infoport 8200
```

Next, a simulation has to be started. Therefore, a configuration file has to be provided setting the number of GPUs used in each dimension, the overall grid size, the number of time steps computed and which plug-ins should be loaded with which parameters. Secondly, a template file that is adapted to the HPC system we want our job to be executed on has to be provided. Finally, a tool called **tbg** takes the configuration and template file as well as the name of a folder for simulation output and schedules a batch job.

tbg -c *config file* -t *template file* ../../runs/*output folder*

A configuration file to run PIConGPU with the in situ volume rendering plug-in looks like this:

```
#!/bin/bash
# Copyright 2013 Axel Huebl
#
# This file is part of PIConGPU.
#
# PIConGPU is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PIConGPU is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PIConGPU.
# If not, see <http://www.gnu.org/licenses/>.
#

# variables which are created by tbg
# TBG_jobName
# TBG_jobNameShort
# TBG_cfgPath
# TBG_cfgFile
# TBG_projectPath
# TBG_dstPath


```

```
30 TBG_wallTime="1:00:00"
31
32 TBG_gpu_x=4
33 TBG_gpu_y=4
34 TBG_gpu_z=4
35
36 TBG_gridSize="-g 256 1536 256"
37 TBG_steps="-s 10000"
38 TBG_movingWindow="-m"
39 TBG_devices="-d !TBG_gpu_x !TBG_gpu_y !TBG_gpu_z"
40
41 TBG_programParams="!TBG_devices     \
42                    !TBG_gridSize    \
43                    !TBG_steps       \
44                    !TBG_movingWindow \
45                    !TBG_analyser  | tee output"
46
47 TBG_analyser="!TBG_insituvv"
48
49 TBG_insituvv="--insituvolvis.period 1 \
50               --insituvolvis.imagewidth 1280 \
51               --insituvolvis.imageheight 1024 \
52               --insituvolvis.serverip 149.220.4.50 \
53               --insituvolvis.serverport 8100 \
54               --insituvolvis.name PIConGPU"
55
56 # TOTAL number of GPUs
57 TBG_tasks="$(( TBG_gpu_x * TBG_gpu_y * TBG_gpu_z ))"
58
59 "$TBG_cfgPath"/submitAction.sh
```

The corresponding template file for the *Hypnos* cluster at HZDR has the following form:

```
1  #!/bin/bash
2  # Copyright 2013 Axel Huebl, Anton Helm, René Widera
3  #
4  # This file is part of PIConGPU.
5  #
6  # PIConGPU is free software: you can redistribute it and/or modify
7  # it under the terms of the GNU General Public License as published by
8  # the Free Software Foundation, either version 3 of the License, or
9  # (at your option) any later version.
10 #
11 # PIConGPU is distributed in the hope that it will be useful,
12 # but WITHOUT ANY WARRANTY; without even the implied warranty of
13 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14 # GNU General Public License for more details.
15 #
16 # You should have received a copy of the GNU General Public License
17 # along with PIConGPU.
18 # If not, see <http://www.gnu.org/licenses/>.
19 #
20
21 ## calculation are done by tbg ##
22 TBG_queue="k20"
23 TBG_mailSettings="bea"
24 TBG_mailAdress="someone@example.com"
25
26 # 4 gpus per node if we need more than 4 gpus else same count as TBG_tasks
27 TBG_gpusPerNode=`if [ $TBG_tasks -gt 4 ] ; then echo 4; else echo $TBG_tasks; fi`
```

```
28
29 #number of cores per parallel node / default is 2 cores per gpu on k20 queue
30 TBG_coresPerNode="$(( TBG_gpusPerNode * 2 ))"
31
32 # use ceil to caculate nodes
33 TBG_nodes="$(( ( TBG_tasks + TBG_gpusPerNode -1 ) / TBG_gpusPerNode))"
34 ## end calculations ##
35
36 # PIConGPU batch script for hypnos PBS batch system
37
38 #PBS -q !TBG_queue
39 #PBS -l walltime=!TBG_wallTime
40 # Sets batch job's name
41 #PBS -N !TBG_jobName
42 #PBS -l nodes=!TBG_nodes:ppn=!TBG_coresPerNode
43 # send me a mail on (b)egin, (e)nd, (a)bortion
44 ##PBS -m !TBG_mailSettings -M !TBG_mailAdress
45 #PBS -d !TBG_dstPath
46
47 #PBS -o stdout
48 #PBS -e stderr
49
50 echo 'Running program...'
51
52 cd !TBG_dstPath
53
54 export MODULES_NO_OUTPUT=1
55 source ~/picongpu.profile
56 unset MODULES_NO_OUTPUT
57
58 #set user rights to u=rwx;g=r-x;o=---
59 umask 0027
60
61 mkdir simOutput 2> /dev/null
62 cd simOutput
63
64 #wait that all nodes see ouput folder
65 sleep 1
66
67 mpiexec --prefix $MPIHOME -tag-output --display-map -x LIBRARY_PATH -x LD_LIBRARY_PATH \
68         -am !TBG_dstPath/tbg/openib.conf  -npernode !TBG_gpusPerNode -n !TBG_tasks \
69         !TBG_dstPath/picongpu/bin/cuda_memtest.sh
70
71 if [ $? -eq 0 ] ; then
72   mpiexec --prefix $MPIHOME -x LIBRARY_PATH -x LD_LIBRARY_PATH -tag-output \
73   --display-map -am !TBG_dstPath/tbg/openib.conf -npernode !TBG_gpusPerNode \
74   -n !TBG_tasks !TBG_dstPath/picongpu/bin/picongpu !TBG_programParams
75 fi
76
77 mpiexec --prefix $MPIHOME -x LIBRARY_PATH -x LD_LIBRARY_PATH \
78         -npernode !TBG_gpusPerNode -n !TBG_tasks killall -9 picongpu
```

After the our batch job starts running the simulation will connect to the server. Now we can start the client select our simulation. This may also require a script setting the library path and configuring IP and port on which the server can be reached.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/rivlib

./client --serverip 149.220.4.50 --serverinfoport 8200
```

# Acknowledgments

I would like to express my appreciation to Prof. Dr. Stefan Gumhold and Dr. Sebastian Grottel who have been my supervisors during this thesis. Both provided me with valuable advice which greatly helped to improve the quality of this thesis.

A special thanks goes to Dr. Michael Bussmann, my supervisor at Helmholtz-Zentrum Dresden-Rossendorf, for giving me the opportunity to work on this interesting task and many inspiring talks.

In my daily work I have been surrounded by a group of friendly and helpful students. In particular Axel Hübl and Richard Pausch helped me to understand the PIConGPU code. For always having an open ear to questions and solving problems while using the cluster system I would like to thank René Widera and Dr. Henrik Schulz.

Finally, I would like to express my greatest gratitude to my parents for supporting me throughout all my studies at University of Technology Dresden and beyond.

# Copyright Information