FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Interactive Visualization and Simulation of Fluids

Achim Däubler

Bachelor Thesis

Interactive Visualization and Simulation of Fluids

Achim Däubler

Bachelor Thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Inf. Simon Bogner, Sebastian Kuckuk, M. Sc.
Bearbeitungszeitraum:	9.6.2014 – 21.11.2014

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelor Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 21. November 2014 .....

Abstract

Simulations play an important role in many fields. This is especially true for fluid flow simulations. This theses will describe fluid flow simulations that are based on the lattice Boltzmann model. A framework that implements this model is waLBerla, a massively parallel multi-physics software framework. The huge amount of data created by simulations has to be made human comprehensible, in order to analyze it. This is done with the help of visualization software. One of these is VisIt, an open source, interactive, scalable, visualization, animation and analysis tool.
There are different approaches to visualize the data. Usually simulations store their data, so it can be visualized later. Another approach is to visualize the data, while the simulation is running. Tools like VisIt are able to run in in situ mode, which means they are capable of visualizing data, as it is created.
VisIt offers a library, which programmers can use to "build the bridge" between the simulation code and the visualization tool. It will be explained how this library works and how it integrates into an existing simulation code. Furthermore the waLBerla framework was extended, using the VisIt library. This makes it possible to connect to a running waLBerla simulation with VisIt and analyze the data, as it is created. Furthermore it was explored, which possibilities VisIt offers to steer the execution of the running simulation.

Contents

List of Figures

1 Introduction

Simulations play an important role in many fields. They are used for technical, as well as scientific purposes, to simulate scenarios that would be difficult or expensive to test in reality. In order to simulate with high resolution of the data, often supercomputers are used, that exploit massive parallelism techniques. One important field of simulation are fluid flow simulations. For these the so called Lattice Boltzmann Models are often used. They are also implemented in waLBerla, which is a massively parallel multiphysics software framework, designed to simulate fluid flows at a high parallelity rate.
Simulations create huge amounts of data, which are far too huge to be analyzed by looking at values. To analyze the data, it has to be visualized, which means the data is transformed into human comprehensible images. There are different approaches to do this. Often simulations store their data, so it can be visualized later. Another approach is to visualize the data, while the simulation is running. There are different visualization tools, one of which is VisIt, that are capable of connecting to a running simulation and visualizing data as it is created.
VisIt offers a library, which programmers can use to "build the bridge" between the simulation code and the visualization tool. The main goal of this thesis was to build this bridge, in order to be able to visualize the data as it is computed, using VisIt [19]. Furthermore it was explored, which possibilities VisIt offers to steer the execution of the running simulation.

2 Visualizing Simulations

This chapter it will be described how fluid flow simulations, based on the lattice Boltzmann method, work. Furthermore it is explained how simulation data is visualized and which concepts are used for that. Especially techniques that are used for parallel visualization and simulation are shown.

2.1 Simulations

2.1.1 The Lattice Boltzmann Method

Lattice **B**oltzmann **M**odels (LBM) are methods for the simulation of fluid flows. They are based on **L**attice-**G**as **C**ellular **A**utomata (LGCA) proposed by Frisch et al. in 1986 [8]. To simulate fluid flow inside a domain it is discretized into discrete locations. LGCA uses fictive particles, which are either present in such a location or not and can only move in certain discrete directions. That is, in the direction of neighboring discrete locations. Time is also separated into timesteps. Each timestep consists of a collision and a propagation step. In the collision step the particles exchange momentum while following the basic physical principle of conserving the overall momentum and mass. In the propagation step the particles move along their associated direction to their next neighbor node. LBMs improve this concept by using continuous **P**article **D**istribution **F**unctions (PDF) instead of single particles. LGCA and LBM use a "bottom-up" approach meaning that they use a discrete microscopic model, namely the particles respectively the PDFs. Macroscopic values like density can be defined, using the PDFs. More information on LCGAs and LBMs can be found in [24].
LBM simulations also use discrete velocities. For simulations in two dimensions the d2q9 (figure 1) model is often used. For three-dimensional simulations the d3q19 (figure 2) is very popular. These models discretize the velocity space, so for example the d3q19 model has 19 discrete velocities in three dimensions. There also exist other models with different numbers of discrete velocities, like the d3q27 model. In general such schemes have the form dDqQ, with D being the number of dimensions and Q being the number of discrete velocities.

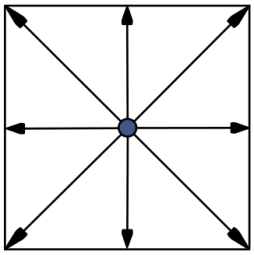

Figure 1: d2q9 [18]

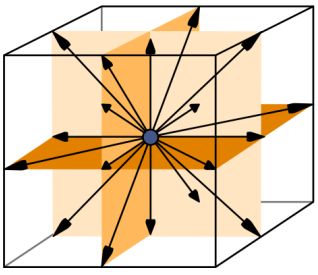

Figure 2: d3q19 [18]

For Q velocities the discretized Lattice Boltzmann equation can be written as:

$$f_\alpha(\vec{x_i} + \vec{e_\alpha}\delta t, t + \delta t) - f_\alpha(\vec{x_i}, t) = \Omega_\alpha(f), \alpha = 0, ..., Q - 1$$

where f_α is the α-th particle distribution function. The whole domain is split into cells and x_i is the i-th cell in the discretized simulation domain. Ω_α denotes the LBM collision operator and e_α is the α-th discrete velocity.
Simulations now have to solve this equation in each timestep in order to update the PDFs per cell. Usually the cells are cubic and are part of a uniform grid, that forms the simulation domain. The equation can be solved in a two step procedure, so each timestep is separated into two steps. The first step is the collision step (1) and the second one is the streaming step (2). For this the equation is rewritten as:

$$f_\alpha(\vec{x_i} + \vec{e_\alpha}\delta t, t + \delta t) = f_\alpha(\vec{x_i}, t) + \Omega_\alpha(f)$$

This can be separated in into the two steps:

$$f^*_\alpha(\vec{x_i}, t) = f_\alpha(\vec{x_i}, t) + \Omega_\alpha(f) \quad (1)$$

$$f_\alpha(\vec{x_i} + \vec{e_\alpha}\delta t, t + \delta t) = f^*_\alpha(\vec{x_i}, t) \quad (2)$$

f^*_α is the post-collision value of the distribution function f_α. Solving the collision equation (1) does not depend on any information from neighboring cells, which makes it well suited for parallel simulations. In order to solve the equation one has to find a good model for the collision function Ω_α. Different approximations exist, examples being the **S**ingle **R**elaxation **T**ime (SRT) model proposed by Bhatnagar et al. [2] and **T**wo **R**elaxation **T**ime (TRT) model proposed by Ginzburg et al. [9].
The result f^*_α of the collision step is the distribution function of the neighboring cell along its associated discrete velocity vector e_α. So in the streaming step (2) the distribution functions of the current node are copied into their associated neighbour cell. Depending on the implementation the values can also be pulled from the neighbor cells instead of pushing the values.

2.1.2 Domain Decomposition

Because of the large amount of data simulations often have to exploit parallelism. Simulations use parallel computers or clusters with enough memory to hold the data that is needed by the simulation. Parallel computers often consist of multiple nodes where each node contains multiple cores. These cores share the memory of their node but have no knowledge of the memory of other nodes. In order to synchronize the computation a mechanism is needed to share data between nodes. For this purpose *distributed-memory* parallelism techniques are required [1]. The most commonly used approach for this is message passing. This means the processes exchange data using messages, which are sent over a network connecting the nodes. For this normally the Message Passing Interface (MPI) [16] is used.
Shared-memory parallelism techniques can be used to increase performance within a node. This is possible as all processors on one node have access to a common memory. The advantage is that all processes can access all data directly and can communicate through their common address space. The problem is, however, that locking mechanisms are required to synchronize data access of different processes. The shared memory techniques do not have to be implemented, as the processors can also use message passing to communicate instead of using the common memory.

In order to distribute the whole simulation domain among the processes, it has to be split into subdomains. This splitting is called domain decomposition. After the splitting every process gets one (or more) subdomains, which it is responsible for. The simulation domain is usually discretized into a uniform Cartesian grid, consisting of equidistant cells. So the subdomains consist of a subset of these cells.
Simulations mostly use purely distributed-memory techniques, so the the individual processes do not have access to the whole domain but only to their subdomain, which is stored in the local memory of the process. A big advantage of LBMs is that the collision step is purely local, as mentioned in 2.1.1. This makes them well suited for parallel simulations, as the simulation domain can be split into data independent subdomains. The streaming step, however, needs information from neighboring cells.
To access the data a common approach is to add an additional ghost layer of cells to each subdomain. This can be seen in figure 3, where two neighboring subdomains are shown. Each of them stores the outermost layer of computational cells of the neighboring subdomain in their ghostlayer. This way the data of all cells can be calculated in each timestep, even on the outermost cells, as the information needed from the neighbouring cells is available in the ghost layers. Then after each timestep there is an extra communication step, which exchanges data between the processes by sending messages over the network. These messages contain the information needed to update the ghost data of each subdomain.

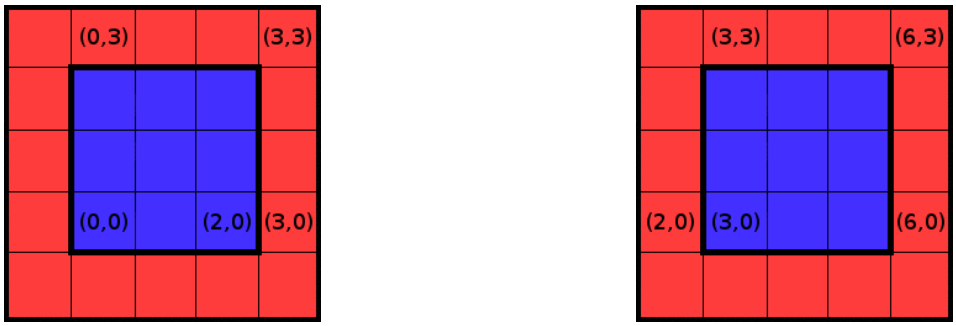

Figure 3: ghost layer

There are different algorithms for the domain decomposition. The goal should be to split the domain in a way that its computational load is uniformly distributed on the nodes and that the communication time is minimal.

2.1.3 Accessing macroscopic values

One is interested in macroscopic values like density or fluid velocity. As mentioned in 2.1.1 these macroscopic values can be related to the microdynamic model. For LBM simulations one has to calculate them from the PDFs at each cell.
For Q discrete velocities the density ρ inside a cell x_i is given as:

$$\rho(\vec{x_i}) = \sum_{\alpha=0}^{Q-1} f_\alpha(\vec{x_i})$$

For the calculation of the momentum $\rho\vec{u}$ ($\vec{u}$ is the fluid velocity of the cell) we additionally need the discrete velocities c_α:

$$\rho(\vec{x_i})\vec{u}(\vec{x_i}) = \sum_{\alpha=0}^{Q-1} f_\alpha(\vec{x_i})\vec{c_\alpha}$$

2.1.4 Boundary Conditions

Simulations are limited to a finite domain. The problem that arises is how to handle the fluid flow at the boundaries. Should the fluid behave like there is a solid wall or should it behave like there is no wall at all? There exist different models to simulate such and other boundary conditions.
First there is the so called no-slip boundary condition, which is used to simulate fluids that hit static walls and applies a certain amount of friction to the fluid. In lattice-Boltzmann simulations the distributions which would be streamed into an obstacle cell are simply reversed, as can be seen in figure 4.

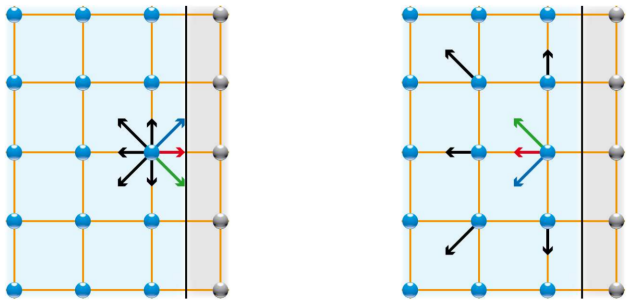

Figure 4: no-slip boundary condition [14]

The second popular condition is the free-slip boundary condition, which models walls without friction. For this the PDFs, pointing to an boundary cell are reflected along their component normal to the wall (figure 5).

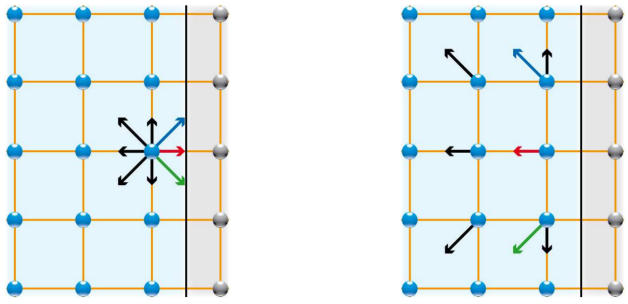

Figure 5: free-slip boundary condition [14]

Then there is the inflow and outflow conditions. They are used to simulate fluid entering or leaving the simulation domain by setting constant PDFs in the boundary cells. This way they hold constant velocity and density values.

As these conditions are used to model the boundaries of the simulation domain, it is reasonable to store them in the ghost layers at the domain border. Another possibility is to define the domain boundaries as periodic, which means that data streamed out of the domain enters at the opposite side. They also can be used to model obstacles inside the domain. One could for example set up a solid obstacle by applying the no-slip boundary condition to a set of cells inside the simulation domain.

2.2 Visualization

2.2.1 Software Design

There are many ways to design a visualization framework that is capable of handling the often huge amounts of data, that are produced by simulations. One way to deal with large data sets is data subsetting. This means that only the salient pieces of the data set are processed and the parts that do not affect the final picture are ignored. Another method is multiresolution processing. This method processes the data at finer resolutions only when necessary. These two methods take advantage of the fact that not the whole data set has to be processed. There also is the streaming technique, which splits the data into multiple pieces and processes these one at a time. Another method is in situ processing, which means that the simulation data is visualized, using the resources allocated for the simulation code. The last technique is pure parallelism, which means that many processors are used to read the entire data set into primary memory. More information about these techniques can be found in the book by E. Wes Bethel et al [1].
Visualization frameworks commonly use so called Data Flow Networks. These are frameworks which provide an execution model, a data model and algorithms to transform data. One of the most popular examples is the Visualization ToolKit (VTK) [15]. Data Flow Networks send the data through a pipeline to create a visualization. The pipeline consists of different modules, which are either sources, sinks or filters. They perform algorithmic operations on the data as it flows through the network. Data flow networks are very flexible, as modules can be replaced to handle different data types or to perform different operations. For example a source can be a file reader, which opens a file containing simulation data and creates a mesh containing data values in its cells. Then different filters are applied to the data, like eg. an isosurface filter, which creates isosurfaces from the data values. In the end a rendering sink can be used to transform the data into an image.
As for simulations, the most common approach to handle huge data sets is parallelization. The above mentioned methods also normally occur in combination with parallelism. Parallelization is supported by all modern visualization frameworks, like for example VisIt, the framework that will be discussed later on in this theses. These visualization frameworks use parallel computers or clusters with enough memory to hold the data, needed for the visualization. The distributed-memory and shared-memory parallelism techniques, mentioned in the Simulations chapter (2.1.2), are also used by visualization programs.
Pure parallelism is the most common way to process data. Similar to simulations, visualization frameworks distribute the whole domain, that should be visualized, over the available processes. Each process operates simultaneously on its portion of the data set without communication with the other processes. The processes usually use a data flow network to read and process the data of its subdomain. In the end the data is rendered. Rendering is a

quite complicated step, as it requires parallel coordination in order to combine the data, produced by the processes. More detailed information about pure parallelism and examples on visualization at extreme scale concurrency can be found in [1].

2.2.2 Distributed Visualization

Many visualization frameworks support a distributed mode of visualization. This means that parts of the visualization pipeline are run on a remote machine (figure 6), for example a super computer. The resulting data is then sent to the client, which executes the remaining steps of the pipeline. There are three possible ways of partitioning the pipeline. Send Data, Send Images, and Send Geometry [1].

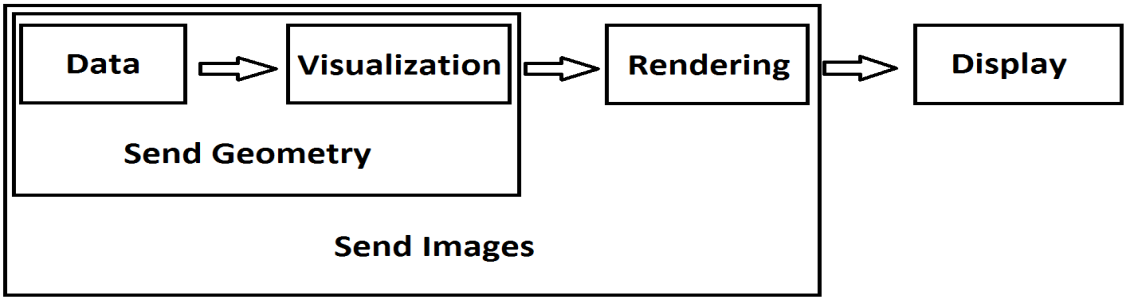

Figure 6: The Visualization Pipeline

Send Data partitioning means that simulation data is sent to the client, that has not been transformed to geometry. This approach has become increasingly impractical as the data amount, especially in scientific applications, grow and the data may exceed the client computers memory. Moving full resolution source data over the network also is a huge bottleneck for big simulations.
A more common approach is the *Send Geometry* partitioning. Here the simulation data is transformed to geometry data by the server. The client then receives renderable data. Ideally only the geometric primitives, that lie in the view-frustum are sent,in order to optimize network usage. A disadvantage is that, depending on the visualization operations, the size of the renderable geometry data may be even bigger than the original data set, although that is usually not the case.
The most common approach, especially for huge data sets, is the *Send Images* partitioning. This means all visualizations steps, up to a viewable image, are done on the server. Scalable rendering operations produce images on the high performance server, which are then sent to the client. This has the advantage that there is a fixed maximum size of the images sent over the network, thus the network load is fixed. A problem is however that, especially for interactive simulations, every time a new frame is requested, it is rendered on the server and sent through the network. So if the client wants to explore the data and needs to update the image often, the network can again be a bottleneck.
Hybrid approaches exist, which can switch between the above mentioned partitionings, in order to save network load and memory usage. This is also supported by VisIt.

2.2.3 In Situ Processing

Visualization of large data is usually done as a post processing step, using reduced data sets, which were produced by a simulation. In situ processing is used to visualize data as it is generated, without the need of expensive I/O operations. I/O is one of the primary bottlenecks of simulations, as it needs time to write the data to the disk. Furthermore the data written is usually compressed, which introduces additional errors to the original simulation data. There are two types of in situ processing.
Co-processing [4] means the visualization routines are part of the simulation code and can directly access the simulation's memory. The visualization routines could be implemented in the simulation code. This is, however, time consuming and not very flexible. There exist richly featured visualization software, which is more flexible an can be integrated into almost any simulation code, although it may cost a bit of performance as it is not optimized for a particular simulation. The biggest advantage of co-processing is, that the simulation data can be accessed in a very efficient way, as the co-processing routines can read it directly. The biggest disadvantage is, that the memory and network bandwidth available to the simulation is reduced by the visualization routines.
Second there is *concurrent-processing*, which means the visualization program runs separately on distinct resources and the simulation data is transferred to the visualization resource through the network. This is quite similar to post-processing visualization, with the difference that the data is not stored to the disk, but sent over the network instead. This way simulation and visualization are separated as it is the case in post-processing, but expensive I/O operations are bypassed. However, the size of the data that has to be transferred over the network, is increased.
There are hybrid forms, which combine these approaches. Here the simulation data is processed by the co-processing routines to a certain degree. It is then sent over a network to a visualization resource, which executes the remaining visualization steps. VisIt also is able to use this hybrid form of in situ processing [22], as will be seen later in this thesis.

2.2.4 Adaptor Layer

As mentioned before, an in situ simulation, which uses co-processing, integrates the visualization routines into its code. So the routines operate in the same address space. In order to provide the simulation data to the routines, a so called adaptor layer is needed. The adaptor layer is responsible for exposing the simulation's data structures in a way that is compatible with the visualization routines. In the best case the data layout of the simulation and the one of the visualization system are very similar. If this is the case developers may be able to simply share pointers to the simulation's data with the visualization code, which does not require any copying. However often the data structures defer, which makes it necessary to implement an adaptor layer, which reorganizes the simulation data. Depending on how big the difference is, this can have a huge impact on the simulations performance. It can be necessary to copy the whole simulation data structure into an object, which is compatible with the visualization pipeline. This means additional computation time is needed as well as additional memory to store the copied data, decreasing the memory available to the simulation.

3 The Visualization Tool VisIt

This theses focuses on the high performance visualization tool VisIt [19]. VisIt is an open source, interactive, scalable, visualization, animation and analysis tool. It is available on Windows, Unix and Mac platforms. VisIt is designed to handle extremely large data sets created by supercomputers, but is also suitable for desktop sized projects. It is build on top of the Data Flow Network VTK. This chapter will especially describe how to use VisIt in in situ mode, as this will be used in the practical part.

3.1 Design

VisIt consists of multiple programs and provides a client/server architecture in order to separate visualization and data analysis into different component programs [3, 11, 22]. The following three components are usually run locally on the client computer, so that they can use its fast graphics hardware. The first component is the *GUI*, which provides a graphical user interface and menus. It is build from the Qt-widget set. There is also the *Command Line Interface* (CLI), which is a command line user interface where the VisIt Python Interface is built-in. The last client-side component is the *Viewer*, which displays all of the visualizations in its visualization windows. It is also responsible for keeping track of VisIt's state and for talking to the rest of the components.
The server components are intended to run on a remote machine, for example a supercomputer. There is the *Database Server*, which is the program that browses the remote file system and passes information about the files there to the GUI. It also opens the files and reads their metadata (eg. the list of variables). Then there is the *Compute Engine*. When the user requests a plot, it is instructed to actually read the data files, assemble the requested data flow networks, generate plots from the data using the data flow networks, and send the plots to VisIt's viewer where the plot can be displayed. The last server-side component is the ***V****isIt* ***C****omponent* ***L****auncher* (VCL), which is the program that is responsible for launching other VisIt components on remote computers. For example it starts the Compute Engine or the Database Server there. The connection between the components can be seen in figure 7.
VisIt offers multiple processing modes. These are multiresolution processing, in situ processing, and out of core processing. The most often used mode, however, is pure parallelism. VisIt uses the Message Parsing Interface (MPI) for its pure parallelism mode. When a request is made by the client, every MPI task of the Compute Engine executes an identical data flow network, but on different pieces of the simulation‘s data set. VisIt‘s data flow networks are based on VTK and extend these by contracts. Contracts [6] are a mechanism to apply different optimizations to the data flow networks depending on the requested operations.
As mentioned in the Distributed Visualization chapter (2.2.2), VisIt also splits the tasks of the visualization pipeline. Based on the size of the resulting surface data, VisIt decides if the data is rendered on the server or if it is to be sent to the client [5]. So surfaces with a small number of primitives are sent to the client and rendered by its graphics hardware, if available. When VisIt‘s heuristic detects that this is not reasonable anymore, it switches to sending images. Each process then creates an image, that is equal in size and contains depth values. After that the images are sent to the client, which composes the final result, using the images depth values. More on these rendering techniques can be found in [13] and [12]. The heuristic, which decides which rendering method to use, can be set by the user and the

user can decide to use a fixed rendering mode.

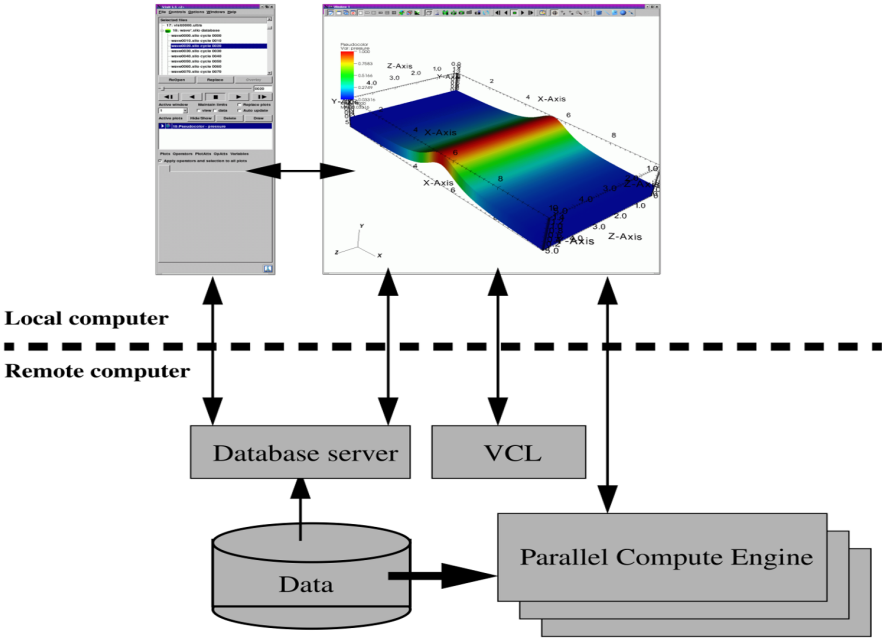

Figure 7: Connection between VisIt Components [11]

3.2 Instrumenting a Simulation Code

As mentioned earlier, VisIt is able to run in in situ processing mode. So you can augment a simulation code in a way that it does not have to write files to the disk, so there are no expensive I/O operations. This chapter will describe how the in situ coupling of VisIt and a simulation code is done. More detailed information about the in situ coupling can be found in [3, 21, 22].
VisIt's in situ mode uses co-processing, which means the simulation and the visualization routines share the same memory. The routines are included in VisIt's library called *libsim*. Libsim is a library that is available as a C version for simulations written in C or C++ and as a Fortran version for Fortran simulations. This thesis however will only target the C version, as it is the one used in the implementation part. Using the library with a simulation code allows to use VisIt as a runtime graphics package.
There are two interfaces in libsim: The fist one is the *Control Interface*, which contains the functions to communicate with VisIt clients. So it can listen for incoming connections, connect back to the client, handle requests and tell the client when the simulation has new data. The second one is the so called *Data Interface*, which contains the functions to get data into VisIt's processing pipeline. More information on that follows in the Data Access chapter (3.4).
The component that processes data is the Compute Engine. When integrating the functions defined by the interfaces into a simulation, it behaves like a Compute Engine. That makes it possible for users to create plots from a running simulation, almost the same way as creating them from data files.
When augmenting a simulation one first has to include the header files. Then a struct should be created, which contains the global state of the simulation. For example it could look like this:

```
typedef struct {
  int    cycle;
  double  time;
  int    runMode;
  int    done;
  int    par_rank;
  int    par_size;
  int    nTotalDomains;
  int    nDomains;
  Domain* domains;
} simulation_data;
```

The first two variables contain information about the current cycle and the current time. This can be displayed by the Viewer in the visualization window. "runMode" is used to tell if the simulation is running or if it is stopped. "done" is set to true when the simulation should end. "par_rank" is the MPI rank of this process and "par_size" is the total number of simulations processes. "nTotalDomains" is the number of all subdomains of the whole simulation domain, "nDomains" is the number of locally allocated domains and "domains" is a array that holds information about these subdomains. More on the subdomains follows in the Domain Decomposition chapter (3.3). Of course this is only an example and one could store other information describing the simulation‘s global state.
Then the simulations mainloop, which executes the discrete timesteps, has to be restructured. A restructured mainloop consists of the following: At the beginning of the mainloop, process 0 checks for inbound VisIt connections, via the function VisItDetectInput(blocking, cmdinput), while the other processes wait in MPI_Bcast until process 0 also calls MPI_Bcast.

```
int blocking, visitstate;
do
{
  blocking = (sim->runMode == VISIT_SIMMODE_RUNNING) ? 0 : 1;
  /* Get input from VisIt or timeout so the simulation can run. */
  if(sim->par_rank == 0)
    visitstate = VisItDetectInput(blocking, fileno(stdin));

  MPI_Bcast(visitstate, 1, MPI_INT, 0, MPI_COMM_WORLD);

  /* Do different things depending on the output from VisItDetectInput. */
  switch(visitstate){
  case 0:
    /* There was no input from VisIt, return control to sim. */
  case 1:
    /* VisIt is trying to connect to sim -
    try to complete the connection and
    register callback functions */
  case 2:
    /* VisIt wants to tell the engine something. */
  case 3:
    /* VisItDetectInput detected console input -
    do something with it. */
  }
} while(!sim->done);
```

In a parallel simulation only the first process communicates with the Viewer. The return value of VisItDetectInput is broadcast to the other processes via the call to MPI_Bcast which then ends the blocking and all processes can process the request simultaneously. VisItDetectInput can be instructed to block indefinitely or to time out after a given period of time allowing the simulation to run while periodically listening for connection requests. Depending on the return value different actions are initiated:

- The return value was 1: There was a connection request. Now libsim's dynamic runtime library is loaded, which is now needed for Compute Engine operations. Also the callback functions for the data access are registered, as they are used by libsim. After that, the simulation connects back to the viewer. It is now fully connected and appears in the GUI's "Compute Engines" and "Simulations" windows. Once connected subsequent calls to VisItDetectInput return different values depending on what VisIt wants to tell the simulation.
- The return value was 0: There was no input from VisIt so the simulation can simply executes one timestep.
- The return value was 2: There was input from VisIt's viewer which has to be processed. It could be a command to generate plots or to do other interactions with the simulation. The request is handled in a way, which ensures all processes call the VisItProcessEngineCommand function. On all processes VisItProcessEngineCommand reads the commands coming from the viewer and processes them.
- The return value was 3: The user has entered a command in the console. The simulation can do something with it. Commands from stdin can be registered, as fileno(stdin) was provided to VisItDetectInput in the example. If no commands should be detected -1 is given to this function and this case can not happen.

3.3 Domain Decomposition

The simulation data is usually split into subdomains, which was discussed in chapter 2.1.2. VisIt's libsim has to know which subdomain belongs to which process, in order to identify the subdomain in the data access functions. The subdomains are identified via an integer value. One should create a struct for subdomains, in order to relate this value to a subdomain and it's data. An example for this could look like:

```
typedef struct{
  int globalIndex;  //global domain number

  int     nNodesX;   //number of nodes in x-direction
  int     nNodesY;   //number of nodes in y-direction
  int     nNodesZ;   //number of nodes in z-direction

  double *x;  //1D coordinates of the of nodes in x-direction
  double *y;  //1D coordinates of the of nodes in y-direction
  double *z;  //1D coordinates of the of nodes in z-direction

  double *velocity  //data array for velocity
  double *densitiy  //data array for density
} Domain;
```

Where "globalIndex" is the unique value, that identifies the subdomain. Additionally the struct contains the information about the data of this domain. That is the number of nodes in each direction, the coordinates for each direction, and variable data. The coordinates are used to create the mesh, that represents this subdomain and the variable arrays hold the data, that "lives" on the mesh.

3.4 Data Access

VisIt's Data Interface uses data access callback functions to read data from the simulation's memory. These callback functions are provided by the augmented simulation. Simulations decide, which callback functions they want to implement. They are registered when the runtime library is loaded, which will call them on demand. The data access callback functions call library functions to allocate handles to data objects like metadata, meshes, and variable objects, and write the corresponding data into the objects.
As soon as a connection to the viewer is established, metadata is requested from the simulation using the metadata callback function. The metadata contains the list of meshes and fields, that are visualizable and also information about the subdomains, if the simulations domain was split. Parallel simulations have to provide a callback function for the domain list. This list contains the information which subdomains of the whole simulation space belong to which process.
When the user requests a plot, VisIt starts executing the corresponding data flow networks. This invokes only the data access functions, that are needed for the requested visualization operations. The request by the client contains constracts, which flow through the data flow network upstream to the data source. Each filter of the network then modifies the contract [5]. At the end the contract is used optimize the size, dimension, extents and ranges of the data, the sources (eg. a file format reader or the SimV2 database reader plug-in) have to read. More precisely it eliminates data, that does not affect the final picture, assigns data to different processors in an optimal way and eliminates unnecessary ghost data.
VisIt uses VTK data sets, which flow through it's data flow networks. Ideally the simulations data array layout matches that of VTK and the simulation runtime library can create a VTK object without copying data. Otherwise the simulation has to provide an adaptor, as stated in chapter 2.2.4 to expose the data to the visualization routines.

A simple data access callback function that returns data stored on the cells looks like this:

```
visit_handle SimGetVariable(int subdomain, const char *name, void *cbdata)
{
  simulation_data *sim = (simulation_data *)cbdata;

  /* Find the right domain. */
  Domain *dom = NULL;
  for(int i = 0; i < sim->nDomains; ++i){
    if(sim->domains[i].globalIndex == domain){
      dom = &sim->domains[i];
      break;
    }
  }

  visit_handle h = VISIT_INVALID_HANDLE;
  int nComponents;
```

```
16    //nTuples = number of cells in this domain
      int nTuples = (dom->nNodesX-1)*(dom->nNodesY-1)*(dom->nNodesZ-1);
18
      if(dom != NULL && VisIt_VariableData_alloc(&h) == VISIT_OKAY){
20      if(strcmp(name, "density") == 0){
          nComponents = 1;
22          VisIt_VariableData_setDataD(h, VISIT_OWNER_SIM, nComponents,
                nTuples, dom->density);
24      }else if(strcmp(name, "velocity") == 0){
          nComponents = 3;
26          VisIt_VariableData_setDataD(h, VISIT_OWNER_SIM, nComponents,
                nTuples, dom->velocity);
28      }
      }
30      return h;
  }
```

In this example, when the user requests a plot containing the density variable, this callback function is called with the char string "density" for every subdomain of the processes. Which subdomains belong to which process, is known from the domainlist. First the data, that belongs to the subdomain has to be found. It is stored in the Domain struct "dom". On every process, we search for the Domain struct with the requested global index number. Then a handle for the data object is allocated via VisIt_VariableData_alloc(handle) and the corresponding variable array is connected to the handle. VISIT_OWNER_SIM means that the simulation is responsible to destroy the array at some point. When an adaptor has to be used and data has to be copied one can change this, so VisIt takes care of destroying the copied data array when it is not needed anymore. For the density only one component is needed, while for velocity three are needed (in this example a 3D velocity). The number of tuples means the number of variables in the mesh corresponding to the subdomain, that was given as parameter.

3.5 Running an instrumented simulation

To connect to an augmented simulation the "visit" command has to be set in the path variable, as libsim needs this to find the libsim runtime library. This is especially important if VisIt component should be run on a remote machine. Instrumented simulations create a .sim2 file every time they are run. This file contains all the information needed to connect to the simulation, including the hostname of the computer running the simulation and the port, used to connect to the simulation. In VisIt‘s GUI the file can now be opened as any other file and VisIt connects to the sim, using ssh. VisIt knows that the data comes from a simulation as the file will be opened with VisIt‘s SimV2 database reader plug-in [21]. After a successful connection the metadata is read, using the metadata access callback. The name of the simulation appears in the Compute engines window and the Simulations window, and plots can be requested.

3.6 Control Functions

There are different methods that can be implemented to control an already running simulation with libsim [21]. Either using the command line, or via reserved buttons in the GUI. One can also specify a user provided .ui file, or control the simulation via the visualization windows.

3.6.1 via the command line

If you want to monitor the console for typed commands, pass fileno(stdin) to VisItDetectInput, as shown in chapter 3.2. If a user typed command is registered, the entered string is read. If the simulation specified a function for the entered string this function is run.

3.6.2 via the reserved buttons in the GUI

VisIt offers the possibility to specify user defined commands in the metadata object. If commands were specified they appear in the GUI‘s simulation window. It is possible to define up to six commands this way. This can be seen in the Implementation chapter in figure 8. Also a control command callback function has to be registered. When clicking one of the buttons, it causes a chain of events that ends up calling the command callback function, which executes function, corresponding to the pressed button. These custom commands give the opportunity to perform limited steering of the simulation from within VisIt.

3.6.3 via a user provided .ui file

Via the above method button clicks can be registered, but what if the user wants to send values to the simulations or if the desired steering possibilities require more than six buttons? VisIt offers the possibility to specify a .ui file, which makes it possible to add user-defined interface elements to the GUI. Files in the. ui format describe the user interface configuration of a program. They are stored in an XML format and contain definitions of Qt-widgets. The files can be easily created with, for example, the tool Qt-Designer [17]. If such a file was specified, one button in the simulations window with the name "Custom..." appears. By clicking the button a new window opens, which is constructed from the descriptions of the .ui file. In the simulation there has to be a callback function registered for each interactive widget. Depending on what widget is used, the callback function either takes a value as input or not. For example a button click ends up simply calling its callback function, which executes the associated function. A change to a SpinBox ends up calling its callback function, which takes as parameter the value, the SpinBox shows in the window. There are also functions for sending values from the simulation to the widgets of the user defined window. An example of a window, containing the widgets, can be seen in the Implementation chapter in figure 9.

3.6.4 via the visualization windows

VisIt offers some operators to restrict the area, that is being plotted and that can be interactively changed in the viewer‘s visualization windows. An example is the box operator, which removes areas of a plot, that are either partially or completely outside of the volume, defined by an axis-aligned box. Now changes to this operator have to be registered by the simulation. VisIt‘s CLI is a command line interface, which uses the VisIt python interface. It is connected to the viewer and updates its state based on what the viewer sends [23]. VisIt’s CLI provides a callback function mechanism, that lets one install custom Python callback functions on changes to this state. This way changes, to eg. the the box operator, can be registered in the callback function. The callback function then sends the list of box attributes to viewer, which sends it to the simulation. In the simulation the registered control command callback is called (the same as, when a button is clicked), with the box command and its arguments. So now the arguments can be used in the simulation for example to specify the selection area.

4 Fluid Simulations with WaLBerla

WaLBerla [20] is a massively parallel multiphysics software framework. It is centered around simulating fluid flows with the lattice Boltzmann method (LBM).

4.1 Design

WaLBerla is written in C++ and is designed to give excellent runtime performance on massively parallel architectures [7]. This is, among other things, achieved by using heavily templated classes and functions in low level codes. It has a modular design, which makes the integration of new simulation scenarios and numerical methods possible.
The waLBerla framework splits the whole simulation domain into so called blocks, which are equal in size [10]. WaLBerla offers block structures, which are able to represent the simulation data in a octree. This means the blocks can be further subdivided into eight subblocks in order to provide finer resolutions on parts of the domain, where it is desirable. The blocks consist of cells arranged in a uniform grid. The actual simulation data is stored in so called fields, which are assigned to the blocks. Each block can hold multiple fields.
For parallel programs waLBerla uses MPI. When running in parallel, all blocks are distributed among the available processes by waLBerlas load balancer. Depending on the load balancing strategy one process can get either one block, multiple blocks, or no blocks. This is especially useful if the computation load per block varies significantly. The structure holding the blocks is also fully distributed, which means each process can only access the blocks, that were assigned to it and has no knowledge about other blocks. This way the processes don‘t allocate data they never touch and dont easily exceed their memory in huge simulations. Of course communication has to be done, if cell values depend on their neighbors.
As mentioned before, simulations are discretized into timesteps. Often these timesteps are executed in a simple while loop. WaLBerla executes the timesteps in so called Sweeps. Sweeps are functions, that operate on a single block and modify its data. The user decides which operations to apply to the block‘s data. The Sweeps are executed iteratively in the so called Timeloop, which is a class that manages the execution of the Sweeps. The Timeloop class allows to add functions that should be executed before the Sweeps and functions that should be executed after them. A typical function to add before the Sweeps is one that does the communication, if it is needed.
Especially in LBM simulations, it is reasonable to use a parameter file due to the large amount of parameters, that influence the simulation. This is because one can change the parameters (eg. boundary conditions or initial velocities) for the simulation without having to recompile the application. For smaller simulations waLBerla also offers its own GUI, which is able to view slices of the field data. Alternatively one can output a VTK file.

So the basic steps when running a simple simulation with waLBerla are:

1. Create an Environment object (this for example initializes MPI info and reads in a parameter file, if specified)

2. Create the blocks by setting up a block structure. This assigns the blocks to each process

3. Add fields to the blocks, which hold the cell data (eg. a field of scalars for density and a field of vectors for velocity). The fields are identified by a unique ID (BlockDataID)

4. Initialize the field values (waLBerla offers some functions to do this automatically eg. by using grayscale images)

5. Create a Timeloop and add Sweeps to it

6. Run the Timeloop

More detailed descriptions on how to use the waLBerla framework can be found in the documentation [20].

4.2 LBM Simulations with waLBerla

One of waLBerla's core features is fluid simulation, using lattice Boltzmann methods. LBM simulations are based on a lattice model. This is mainly defined by the following two features. As described in chapter 2.1.1 LBMs use discrete velocities, defined by models like d2q9 to describe the neighbor cells, that should be taken into account. In WaLBerla, these models are implemented in so called Stencils. The lattice models, currently provided by waLBerla for LBM simulations, are based on the d2q9 model for 2D simulations and the d3q19 and d3q27 models for 3D simulations. The other important feature of the lattice model is the collision model (also described in 2.1.1), which defines which method to use in the collide step. WaLBerla offers some predefined collisions models, like SRT and TRT, but users can also implement their own models.
As described in the Domain Decomposition chapter (2.1.2), waLBerla also uses a ghost layer in LBM simulation, as this is important to calculate the values on the block boundaries. For LBMs there is a special type of field, called PdfField. It stores the particle distribution functions for each cell and is a ghostlayer field with the additional layer. It also provides member functions to calculate macroscopic values, like density or velocity.
The information about boundary conditions and geometry is stored in a FlagField. It stores information for each cell about its type (eg. NoSlip boundary cell). The boundary conditions, that should be used for the simulation are grouped together in the BoundaryHandling class, which provides some common Lattice Boltzmann boundary conditions, like eg. NoSlip and FreeSlip, which were shortly described in chapter 2.1.4. Users can add implementations of their own boundary conditions. WaLBerlas geometry module can be used to set up the domain (usually consisting of a FlagField and a PdfField) by using the given parameters. It can initialize the boundary conditions at the domain borders, place obstacles by using eg. images, etc.
Communication is necessary as the ghost layers need to be synchronized. For that the PackInfo class is used, which creates messages containing the ghost layers. It also writes the message data into the communication partners corresponding ghost layer. In order to know where to send the messages waLBerla uses a so called scheme. The scheme defines which processes need to communicate and sends the packages over MPI. It communicates with all direct neighbors, defined by the stencil, used for the simulation.
For LBM simulations multiple Sweeps are added to the Timeloop. First we have to add the Sweep for the communication. Secondly we add the Sweep for the boundary handling, which sets valid PDF values at the boundaries. Thirdly the LBM Sweep is added, that contains the code for the stream and collide step.

4.3 Accessing Macroscopic Values

The macroscopic values have to be calculated from the PDFs. WaLBerla offers adaptors, which can be used to calculate the values. These adaptors behave like fields. The difference is that they do not store values, but calculate them based on the PdfField, using the lattice model. For other types of simulations, other adaptors can be implemented.
They are added to the blockstorage via a special function (addFieldAdaptor), which returns the BlockDataID for the adaptor. So the adaptors can be used like any other field. They are identified by their ID in a block and their values are read, via their data access functions, which are the same as for a normal field.

5 Implementation

The practical part of this theses was to "build the bridge" between the visualization framework VisIt and the simulation framework waLBerla. Also it was to be found what possibilities there are to control the running simulation. The following chapters describe how the connection works, how VisIt's libsim library integrates into the simulation code, and what mechanisms where implemented to steer the simulation.

5.1 General Design

The goal was to implement a class that can be used instead of the GUI class provided by waLBerla and integrates similarly into the code, written by waLBerla users. So instead of using waLBerla's GUI class, users can use the VisitGUI class. This class adds routines to do the communication with a VisIt client. Using VisIt's GUI on the client machine, the user can request plots to analyze the data, while the simulation is running. He can also steer the execution of the waLBerla code, eg. instruct the simulation to run or halt. Furthermore he can place obstacles inside the simulation domain and set density and velocity values interactively, for LBM simulations.
When using the VisitGUI class, the user has to call its registerAdaptor(adaptor) method in order to register an adaptor object, which a instantiation of a subclass of the abstract VisitAdaptor class. These adaptors are used to expose the simulation data to VisIt's visualization routines. When the user wants a field, he added to waLBerla's blockstorage, to be visualizeable, he has to create an adaptor object for it. There are three different types of adaptors, which are the FlagFieldVisitAdaptor, the ScalarFieldVisitAdaptor, and the VectorFieldVisitAdaptor. Depending on the types of fields the user wants to visualize, he has to include the corresponding header file.
When compiling an application, that should use VisIt, one has to set the CMake switch WALBERLA_ENABLE_GUI to ON, as described in waLBerlas documentation [20]. If it is set to OFF the application will compile a version of the VisitGUI, that simply calls the Timeloop's run() member function. In other words, if the switch is not enabled the simulation runs as if no GUI was used. Applications that include the VisitGUI.h file, have to add the visit module to its dependency list, so it is linked to the application. As waLBerla offers the possibility to be compiled without using MPI, when the WALBERLA_BUILD_WITH_MPI switch is set to OFF, the visit module also uses the definitions of the switch and can be compiled as a serial version.
If the user wants to interactively set values in a field, he has to call the VisitGUI's registerInteraction(interaction) member function, in order to register an object of a subclass of the FieldInteractions class. Currently there is only the PdfFieldInteractions class, which provides functionality to set density and velocity values or to set up obstacles. As the name suggests, this class only works with PdfFields, because it uses functions from the PdfField class to set the values.
The PdfFieldInteractions class is not part of the visit module, although the VisitGUI class can use it to set values on fields. This is because it includes headers from the lbm module. If it was part of the visit module, every time an application depends on the visit module the lbm module would also have to be compiled and linked to the applications executable. As the lbm module itself depends on many modules, there would be many unnecessary modules linked

to the simulation. As a consequence this class is part of the lbm module, so only applications that actually use waLBerla's LBM functions need to compile and link to the module.

A (shortened) example for a simple user application may look like this:

```
//walberla includes
....

//includes important for the visit module:
#include "visit/VisitGui.h"
#include "visit/ScalarFieldVisitAdaptor.h"

using namespace walberla;

int main( int argc, char ** argv )
{
  //set up the environment and create a block storage
  ...

  //add a scalar field to the blockstorage
  typedef GhostLayerField<real_t,1> ScalarField;
  BlockDataID fieldID = field::addToStorage<ScalarField>
  (
      blockstorage,  // the block storage
           "nameOfField",  // name of the field
    );

  // Create a communication scheme
  // and add a PackInfo that packs/unpacks our field
  ...

  // Initialize the field
  ...

  // Create Timeloop
  ...

  // Registering the Sweep
  timeloop.add()<< BeforeFunction(myCommScheme, "Communication")
              << Sweep( MySweep(fieldID), "MySweep" );

  VisitGUI gui ( timeloop, blockstorage, argc, argv );
  visit::ScalarFieldVisitAdaptor<ScalarField> fieldAdaptor (
    //the BlockDataID of the field
    fieldID,
    //the name that was defined when adding the field to
    //the storage, could be any other name though
    blockstorage->getBlockDataIdentifier(fieldID)
  );
  gui.registerAdaptor(fieldAdaptor);
  gui.run();

  return 0;
}
```

In this application a GhostLayerField, which holds scalar values (real_t in this case), is

created. The field is added to the blockstorage, which returns a BlockDataID, which uniquely identifies the field. After that, a communication scheme is set up, which is needed in this example, as a GhostLayerField is used and the ghost data has to be synchronized. Then the field is initialized, for example with the use of an image file. Afterwards a Timeloop object is instantiated and its add() method is called, to register the communication function and the Sweep, that the user created. Finally the VisitGUI object is created, which uses the Timeloop and the block storage objects. As the field used in this example holds scalars, the ScalarFieldVisitAdaptor is used. Its constructor is given the BlockDataID of the field and a unique name. The ScalarFieldVisitAdaptor.h file has to be included, which only includes files necessary to view scalar values. The adaptor is then registered and the run() method is called, which starts the communication with VisIt.

5.2 Communication with VisIt

Most simulations contain a main loop, that executes the timesteps of the simulation. As described in 3.2 an augmented simulation's main loop waits for incoming connections from a VisIt client. In waLBerla, the class, that manages the execution of the Sweeps, is the Timeloop class. In simulations, that run without waLBerla's built-in GUI, an object of the Timeloop class is instantiated, its run() method is called and the execution of the Sweeps starts. When the GUI is used the, timeloop object is passed to the GUI. Then the GUI controls the execution of the Sweeps, via the Timeloop's functions.
To implement the connection between VisIt and waLBerla the VisitGUI class was created, which replaces the built-in GUI. It also takes as input the Timeloop object and steers its execution.
The run() method is structured like this:

```
void VisitGUI::run() {
  while(1) {
    visitCommStep();     //communication with VisIt
    if(sim.done == 1){
      break;          //simulation is done
    }else if(!executeTimestep){
      continue;       //VisIt sent a command
    }
    ...
    timeloop_.singleStep(); //execute one timestep
    ...
  }
}
```

Before each of waLBerla's timesteps, the communication method visitCommStep() of the VisitGUI object is called, to connect to VisIt or handle commands. It contains the VisItDetectInput() function and is very similar to the code example of the communication loop in chapter 3.2. Depending on the input from VisIt, this method is called several times. If no input was registered (VisItDetectInput() returned 0 and "executeTimestep" was set to true) the waLBerla code continues to execute. This means the Timeloop executes its registered functions and Sweeps. So if there was no input from VisIt the waLBerla simulation runs as usual.

5.3 Using VisIt's Domain Concept

As mentioned in chapter 4.1 waLBerla splits the simulation domain by using so called blocks, which hold the field data. In serial simulations waLBerla assigns all blocks to one process. In parallel programs, waLBerla's load balancer distributes all blocks among the available processes. Depending on the load balancing strategy one process can get either one block, multiple blocks, or no blocks.
Libsim offers a mechanism to deal with that situation. It can identify parts of the simulation domain by assigning a unique integer value to each subdomain. These subdomains represent a block in waLBerla. Blocks, however, are identified by a unique IBlockID. So the the IBlockIDs have to be linked to a subdomain number. They can't be used directly as libsim uses consecutive integer values in it's callback functions. Because of that a Domain struct (similar to the one in 3.3) is used, which links a unique integer value to each IBlockID:

```
typedef struct
{
  int globalIndex;  //global domain number

  //the BlockID of the block, that is represented by this domain
  shared_ptr<IBlockID> blockID;

  int nNodesX;  //number of nodes in x-direction
  int nNodesY;  //number of nodes in y-direction
  int nNodesZ;  //number of nodes in z-direction

  double *x;  //1D coordinates of the of nodes in x-direction
  double *y;  //1D coordinates of the of nodes in y-direction
  double *z;  //1D coordinates of the of nodes in z-direction
} Domain;
```

Where "globalIndex" is the global index number, that uniquely identifies the subdomain. "blockID" is the IBlockID, that uniquely identifies the block corresponding to this subdomain. The struct also contains the number of nodes in each direction and the coordinates for each direction. They are used to create the subdomain's mesh.
Everytime a plot is requested, the data access callback functions (see chapter 3.4), corresponding to the requested variables, are called. They are called for each subdomain, so that every part of the whole domain is processed. When libsim calls a data access callback, it provides the global index number of the subdomain as parameter. The callback now has to know which block belongs to this global index. This struct is now used to first identify the subdomain by the "globalIndex" value and to get the corresponding block by using the "blockID" value.

The global index values were initially assigned to the blocks like this:

```
 int index = 0;

std::vector< shared_ptr< IBlockID > > blockIDs;

//Returns the block ID of every local block in the simulation
sim->blockForest->getAllBlocks( blockIDs );
for (std::vector<shared_ptr<IBlockID>>::iterator blockID_it =
    blockIDs.begin(); blockID_it != blockIDs.end(); ++blockID_it){

  const BlockID blockID = *dynamic_cast<BlockID*>(blockID_it->get());
```

```
    AABB aabb;
12  sim->blockForest->getAABBFromBlockId(aabb, blockID);
    //every process gets zero, one or more globalIndex values
14  if(::walberla::MPIManager::instance()->numProcesses() == 1 ||
       sim->blockForest->blockExistsLocally(blockID))
16  {
      //the block exists locally -> add it to the process domain list
18    Domain_ctor(&sim->domains[sim->nDomains], index, *blockID_it, aabb);
      sim->nDomains++;
20  }
    index++;
22 }
```

First the IBlockID of every block in the simulation is stored in a vector. Note that this will only work if the application configured the blockstorage, so that it contains global block information after creating the blocks. In serial simulations this does not matter as there is only one process, which contains the BlockIDs of all blocks. In parallel simulations by default only the local BlockIDs are maintained.
Then it is checked for each block, if it is exists locally, meaning it is allocated on this process. If it is, a Domain struct for this block is created. In Domain_ctor() it is populated with data, including the global index and the IBlockID of the current block. Then the index value is incremented. This way every process gets a unique global index number, which starts with 0 and is numbered consecutively.

5.4 Data Adaptor

Libsim offers five functions, that can be used to expose arrays of different data types to the visualization pipeline. These functions exist for int, long int, double, float and char. The user can store any type of data in the simulation's fields. When accessing the field data in the callback function, it has to be determined which of these functions has to be used, depending on the data type. Another problem is, that the data arrays must have the same data layout as the VTK objects use in the pipeline. So the data has to be reorganized. Also the callback functions are called with the name (as character string) of the variable that is to be sent (again, see chapter 3.4). So we also have to know, which field corresponds to this string.
These problems are solved by using the abstract VisitAdaptor class. It contains many overloaded functions. Five of them are:

```
void VisIt_VariableData_setDataT(visit_handle obj, int owner, int nComps, int
    nTuples, char * data){
2    VisIt_VariableData_setDataC(obj, owner, nComps, nTuples, data);
}
4 void VisIt_VariableData_setDataT(visit_handle obj, int owner, int nComps, int
    nTuples, int * data){
  VisIt_VariableData_setDataI(obj, owner, nComps, nTuples, data);
6 }
  void VisIt_VariableData_setDataT(visit_handle obj, int owner, int nComps, int
    nTuples, float * data){
8 VisIt_VariableData_setDataF(obj, owner, nComps, nTuples, data);
}
10 void VisIt_VariableData_setDataT(visit_handle obj, int owner, int nComps, int
    nTuples, double * data){
  VisIt_VariableData_setDataD(obj, owner, nComps, nTuples, data);
```

```
}
void VisIt_VariableData_setDataT(visit_handle obj, int owner, int nComps, int
    nTuples, long * data){
  VisIt_VariableData_setDataL( obj,  owner,  nComps,  nTuples, data);
}
```

They are used by subclasses of the VisitApdaptor, in order to send field data of any type to VisIt. Depending on the data type, the right VisIt_VariableData_setDataT is chosen, which calls the correct libsim function. The VisitApdaptor class also contains further versions of VisIt_VariableData_setDataT, for other data types (eg. unsigned int), but these have to be cast or can not be used, as there is no corresponding libsim function to them.

As mentioned in chapter 5.1 the VisitAdaptor has three different subclasses:

1. ScalarFieldVisitAdaptor: used for fields holding a scalar value
2. VectorFieldVisitAdaptor: can currently only be used for fields that hold Vector3 values (waLBerla's representation of three-component vectors)
3. FlagFieldVisitAdaptor: used for FlagFields. It can be set up to send either the first char of the boundary string or the integer value representing the boundary.

Their constructors have the following two arguments. First the identifier of the field (BlockDataID) and second a string, which should be unique, as it is used to identify the adaptor in the data access callback functions. This string is also the name, which will appear in VisIt's GUI plot selection window. These classes have a template parameter, which expects the exact type of the field (eg. a field registered in the ScalarFieldVisitAdaptor can be a Field<valuetype,1> or a subclass of it, with "valuetype" being a scalar type).
Each adaptor implements the virtual method setDataInVisit(handle, currentBlock, nNodesX, int nNodesY, int nNodesZ). This method is used to expose the variable data arrays to the visualization pipeline. First the field data is read from the current block, using the BlockDataID of the field, that is represented by this adaptor.

```
const field_t * field = currentBlock.getData< field_t > ( fieldID_ );
```

This is why the template parameter "field_t" is important, as it is needed for the block's getData(fieldID) method. After that, the variable data is set via VisIt_VariableData_setDataT. But first the data has to be transformed, to match the VTK format.
This has to be done as waLBerla's fields internally use padding to store the data, in order to increase performance. The libsim functions, however, use a 1D array without padding. So the current solution is to simply copy the data into the new array. Of course this is not an efficient solution, as computation time and memory is needed. For the ScalarFieldVisitAdaptor this code segment looks like the following:

```
// Iterate over the field
for(int z=0; z<nNodesZ-1; z++){
  for(int y=0; y<nNodesY-1; y++){
    for(int x=0; x<nNodesX-1; x++){
      variable[x+y*(nNodesX-1)+z*(nNodesX-1)*(nNodesY-1)] =
        field->get(x,y,z);
```

```
    }
  }
}
VisIt_VariableData_setDataT(obj, VISIT_OWNER_VISIT, 1, nTuples, variable);
```

The other adaptor types basically work the same way and mostly defer in what they have to provide VisIt_VariableData_setDataT(handle, owner, nComps, nTuples, data) as the value for nComps (eg. 3 for VectorFieldVisitAdaptor, as the vectors have three components).

5.5 Manipulating Field Data

The FieldInteractions class is the base class for manipulating data in a field. The PdfFieldInteractions currently is the only subclass. It provides functionality to set density and velocity values or to set up boundary conditions in a given selection area. As the name suggests, this class only works with PdfFields, because it uses functions from the PdfField class to set the values. Its constructor takes the BlockDataID of the PdfField and the name of the field. It also takes the BlockDataID of the BoundaryHandling object, that was added to the blockstorage. This is important to be able to manipulate boundary flags. Also one has to provide template parameters, which are the type of the PdfField and the type of the BoundaryHandling object.
When the user instructs the simulation to set the variables, he specified before, it is iterated over all blocks. For all blocks, that are within the selection area, the density and velocity values are set via the PdfField's setDensityAndVelocity(cell, velocity, density) method:

```
field_t * pdfField = (field_t*)currentBlock.getData< field_t > ( fieldID_ );
for(CellInterval::const_iterator currCell_it = selection.begin();
    currCell_it != selection.end(); ++currCell_it)
{
  pdfField->setDensityAndVelocity(*currCell_it,velocity,density);
}
```

In this code snippet "field_t" is the typename of the PdfField. It is needed for the block's getData<fieldtype>(fieldID) method, which returns the PdfField, identified by the BlockDataID "fieldID_". Then the density and velocity values are set on all cells of the block, that are inside the selection. When setting the values one has to be very careful, as the numerical stability of the simulation can break, if values are set, which are too high or too low. If a stability check was registered, the simulation will abort in that case.

Boundary values can be set via the provided BoundaryHandling object. It uses a FlagField to store which boundary condition is applied in which cell. For all blocks, that are within the selection area, the boundaries are set like this:

```
houndary_handling_t * bh = (houndary_handling_t*)currentBlock.getData<
    houndary_handling_t>(boundaryHandlingId_);

//assume that there is a "NoSlip" boundary condition
BoundaryUID boundaryUID_NoSlip ("NoSlip");

typename houndary_handling_t::flag_t noSlip = bh->getBoundaryMask(
    boundaryUID_NoSlip);
```

```
if(noSlip == typename houndary_handling_t::flag_t(0)){
  WALBERLA_LOG_WARNING("boundary handling does not contain a boundary condition
     with the name \"NoSlip\"");
}else{
  bh->forceBoundary(noSlip, selection);
}
```

"boudary_handling_t" is the typename of the boundary handling class. As in the code above this snippet, the type is needed for the getData method. The flag for the boundary condition with the name "NoSlip" is set via the forceBoundary(boundaryFlag, selection) method. This is an experimental piece of code, as it is simply assumed, that the boundary handling object contains a flag for a boundary condition with the unique BoundaryUID("NoSlip"). If no flag is found, for this boundary ID, no boundaries are updated.

5.6 Steering the Simulation

As mentioned in chapter 3.6, VisIt offers four mechanisms to control a running simulation. They have been implemented in this project.

5.6.1 via the command line

The simulation monitors the console for typed commands, as fileno(stdin) was passed to VisItDetectInput. If the user types a command in the terminal, it is registered by VisItDetectInput, which returns the value 3. In this case ProcessConsoleCommand() is called, which is responsible for processing the command. In this function the string from the console is read via VisItReadConsole(). Then the function that corresponds to the string is executed. To show a list of available commands one can type "help" in the console.

5.6.2 via the reserved buttons in the GUI

VisIt offers the possibility to specify user defined commands in the metadata object. If commands were specified they appear in the GUI‘s simulation window. In this project the following five button commands were registered in the metadata access callback function:

```
/* Add some custom commands. */
const char *cmd_names[5] = {"halt", "step", "run", "update", "procid_plot"};

for(unsigned int i = 0; i < sizeof(cmd_names)/sizeof(const char *); i++)
{
    visit_handle cmd = VISIT_INVALID_HANDLE;
    if(VisIt_CommandMetaData_alloc(&cmd) == VISIT_OKAY)
    {
        VisIt_CommandMetaData_setName(cmd, cmd_names[i]);
        VisIt_SimulationMetaData_addGenericCommand(md, cmd);
    }
}
```

With these commands the simulation can be instructed to halt or run. It can also be instructed to execute a single timestep ("step"). "update" requests new metadata and updates the plots, selected in the GUI. Finally, "procid_plot" instructs the augmented simulation to plot which process is assigned to which part of the simulation. This is done by sending VisIt CLI Python

commands to the CLI. The commands are sent to VisIt in a non-blocking fashion and VisIt later translates the commands into requests to the simulation.
When clicking one of the buttons in the simulations window, it causes a chain of events that ends up calling the command callback. The callback then executes the function, corresponding to the pressed button. It is the same, as if it was called through the command line.

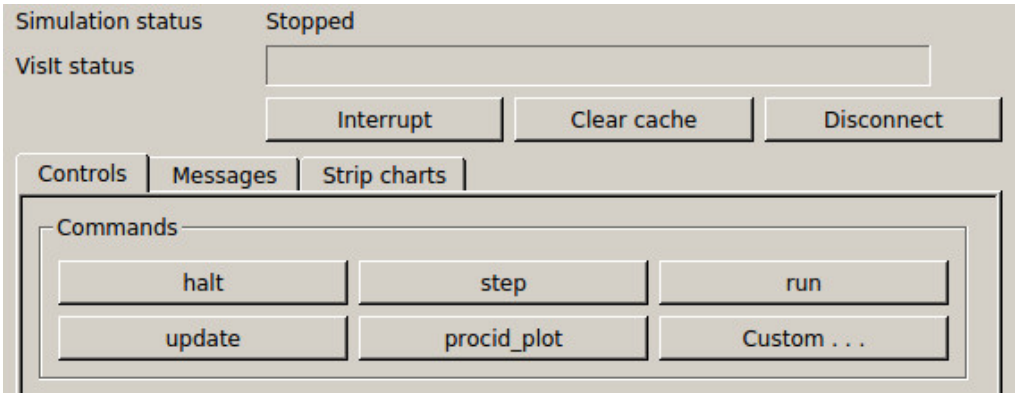

Figure 8: Simulation Window

5.6.3 via a user provided .ui file

The implementation invokes a .ui file, which is stored in waLBerla's src/visit/ directory. It has to be copied manually into VisIt's .visit/ui/ folder, as libsim will search for the file there. As such a file was specified in the implementation, when connecting to the simulation, the sixth button is now labeled "Custom...". When clicking on it a window opens up, which is constructed based on the definitions of the .ui file (figure 9).
It contains the same commands as in the simulation windows, but also adds new functionality. The "Step Size" command is used to increase the number of timesteps that are executed, when the "Step" button is pressed. The "Update Interval" field defines how many timesteps have to be executed until the simulation automatically updates, without having to press the update button (0 means: do not automatically update). The buttons in the "Steering" tab are used for setting velocity or density values in a PdfField. They only have an effect if a FieldInteractions object was registered via the VisitGUI's registerInteraction(interaction) method. The density and velocity values are set when the "Set Variables" button is pressed. The values are set inside the bounding box, that can be set via "Draw Selection". "Set Obstacle" sets the "NoSlip" flag inside the selection.
There is a callback function registered for each interactive widget. The callback functions for the Spinbox widgets get the value, that is displayed in the window, as parameter. They are called a soon as the user changes the value. So when the user types a new value into the "Density" field, its callback function is called by libsim and the value is used in the waLBerla method for setting density values. Currently, only integer values can be registered this way, as VisIt provides only callback functions for integers.

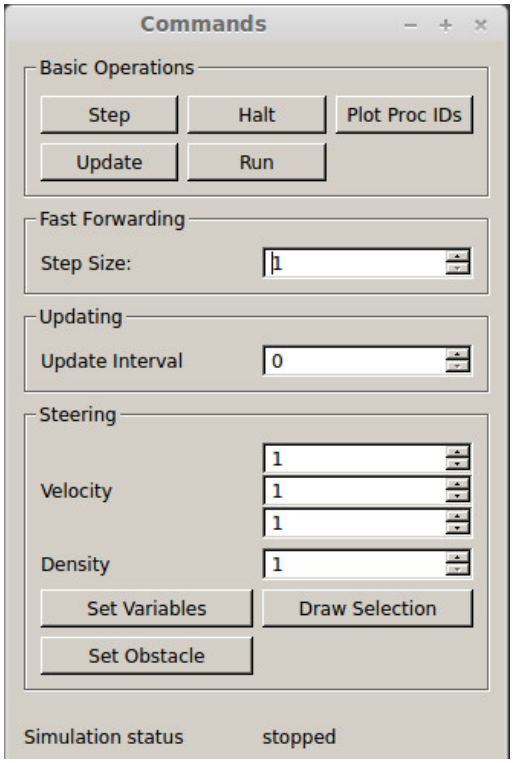

Figure 9: user defined interface, created from .ui file

5.6.4 via the visualization windows

The implementation can set density and velocity values inside a given selection area. This area is defined by using the box operator. The "Draw Selection" button (figure 9) executes a Python script via the CLI, which installs the callback function, in order to register changes to the box operator. When the user changes the selection area, the callback is invoked as described in 3.6.4. The Python callback sends the box operator information to the simulation, which updates the selection values, which are stored in its simulation_data struct. The selection will be used as the area, where the values are set, when the user presses the "set Variables" button.

6 LBM Example

The example simulation is a lid driven cavity problem. Here all cells at the domain border are set to no-slip, except for the cells at the top. They are marked with a velocity-bounce back-flag. The velocity-bounce-back boundary condition internally works with one constant velocity.

6.1 Parameters

In this example an LBM simulation was run. The simulation uses the d3q27 model and the SRT collision model. It is configurable via a parameter file, which uses the following parameters:

```
DomainSetup
{
   blocks         <  4,  3, 2 >;
   cellsPerBlock <  25, 20, 10 >;
   periodic       <  0,  0, 1 >;
   maxnrOfBlocksOn1Process  0;
}

Boundaries
{
  Velocity0 < 0.2, 0, 0 >;

  Border { direction W,E,S;     walldistance -1;  NoSlip     {} }
  Border { direction N;     walldistance -1;  Velocity0     {} }
}
```

The blockstructure is created by using the DomainSetup parameters. "blocks" is the number of blocks in x, y and z direction and "cellsPerBlock" is the number of cells of each block in each direction. After the initialization, there is a blockstorage, which contains 24 blocks. The last two parameters mean that the domain is periodic in z-direction and that each process can be assigned any number of blocks by the load balancer.
The boundary conditions are set via the "Boundaries" options. The "Velocity0" parameter is used to create a SimpleUBB (velocity bounce back) boundary condition on the north side of the domain. It works with one constant velocity, which is <0.2, 0, 0>.The west, east, and south sides use the NoSlip boundary condition. Bottom and top are periodic.

6.2 Registering Field Data

Here is an excerpt from the code, where the newly implemented classes are used to register fields (or field adaptors in this case), so that they can be displayed in VisIt. Also the PdfField is registered for interaction.

```
BlockDataID densityFieldID = field::addFieldAdaptor<DensityAdaptor_T> ( blocks,
    pdfFieldId, "density" );
BlockDataID veclocityFieldID = field::addFieldAdaptor<VelocityAdaptor_T>(
    blocks, pdfFieldId, "velocity" );


std::string density_name = blocks->getBlockDataIdentifier(densityFieldID);
std::string velocity_name = blocks->getBlockDataIdentifier(veclocityFieldID);
std::string flags_char_name = blocks->getBlockDataIdentifier(flagFieldID);
std::string flags_scalar_name = blocks->getBlockDataIdentifier(flagFieldID)+"
    _scalar";
std::string pdfField_name = blocks->getBlockDataIdentifier(pdfFieldId);

VisitGUI gui ( timeloop, blocks, argc, argv );
visit::ScalarFieldVisitAdaptor<DensityAdaptor_T> densityFieldAdaptor (
    densityFieldID, density_name);
gui.registerAdaptor(densityFieldAdaptor);
visit::VectorFieldVisitAdaptor<VelocityAdaptor_T> velocityFieldAdaptor (
    veclocityFieldID, velocity_name);
gui.registerAdaptor(velocityFieldAdaptor);
visit::FlagFieldVisitAdaptor<FlagField_T> flagFieldAdaptor (flagFieldID,
    flags_char_name, false);
gui.registerAdaptor(flagFieldAdaptor);
visit::FlagFieldVisitAdaptor<FlagField_T> flagFieldAdaptorScalar (flagFieldID,
    flags_scalar_name, true);
gui.registerAdaptor(flagFieldAdaptorScalar);
lbm::PdfFieldInteractions<PdfField_T, BoundaryHandling_T> interaction (
    pdfFieldId, pdfField_name, boundaryHandlingId);
gui.registerInteraction(interaction);
gui.run();
```

The simulation uses two field adaptors to calculate the macroscopic values density and velocity from the lbm::PdfField. The names, that are passed to the VisitAdaptors, are the ones that are displayed in VisIt to select the corresponding plot. Also a PdfFieldInteractions object is registered, to manipulate the simulation. In this example the same names are used for the objects, which were specified when adding the fields to the blockstorage. After registering all objects, the communication with VisIt is started via the run() method.

6.3 Running the Simulation

This example was run using five MPI processes. Which blocks where assigned to which process can be seen in figure 10. This plot was generated using the "procid_plot" command, described in chapter 5.6.2.

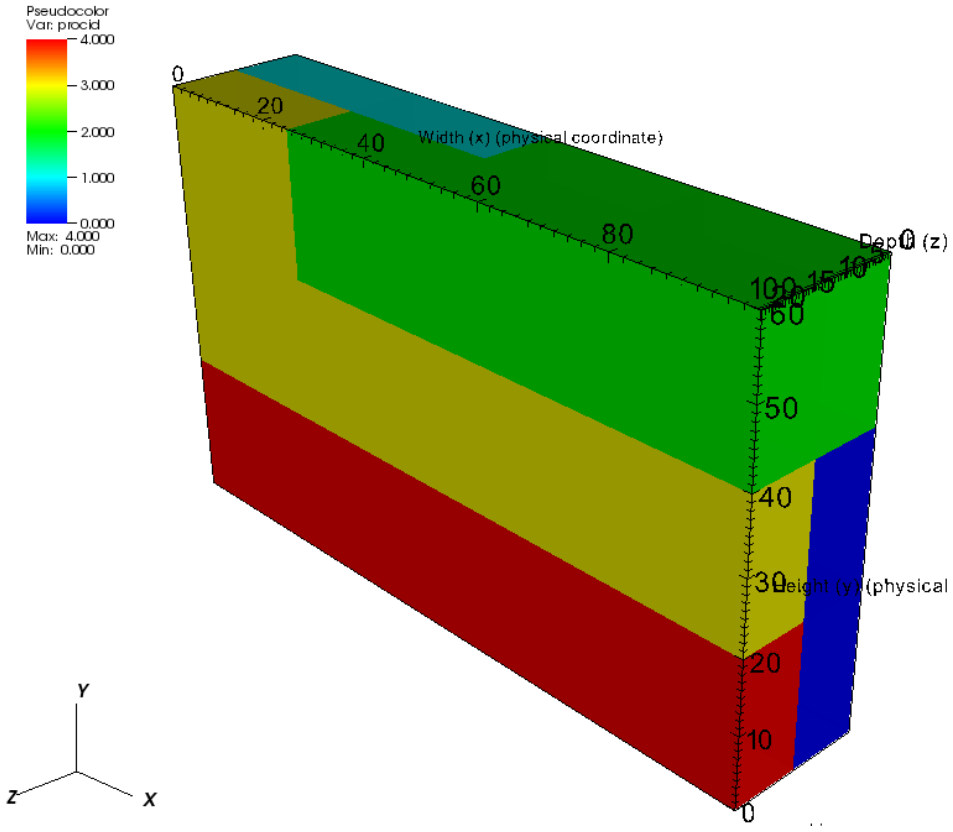

Figure 10: process assignment

Then 100 timesteps were executed. The following two images show the density (figure 11) and velocity (figure 12) values after these timesteps. The plot used for density is a so called pseudocolor plot, which maps scalar variable values to colors, which are then drawn onto the mesh. The velocity variable uses the vector plot, which shows the velocity cell data as vectors. Size and colors of the vectors depend on their magnitudes. In the plots, one can see how the SimpleUBB boundary condition influences the fluid. It models a wall that moves with a speed of 0.2 along the x-axis. So the fluid also moves along the wall, which causes lower density values in the top left corner, as the fluid is carried away by the wall. Higher density values occur on the top right corner as the fluid is pressed against the wall, which has the NoSlip boundary condition.

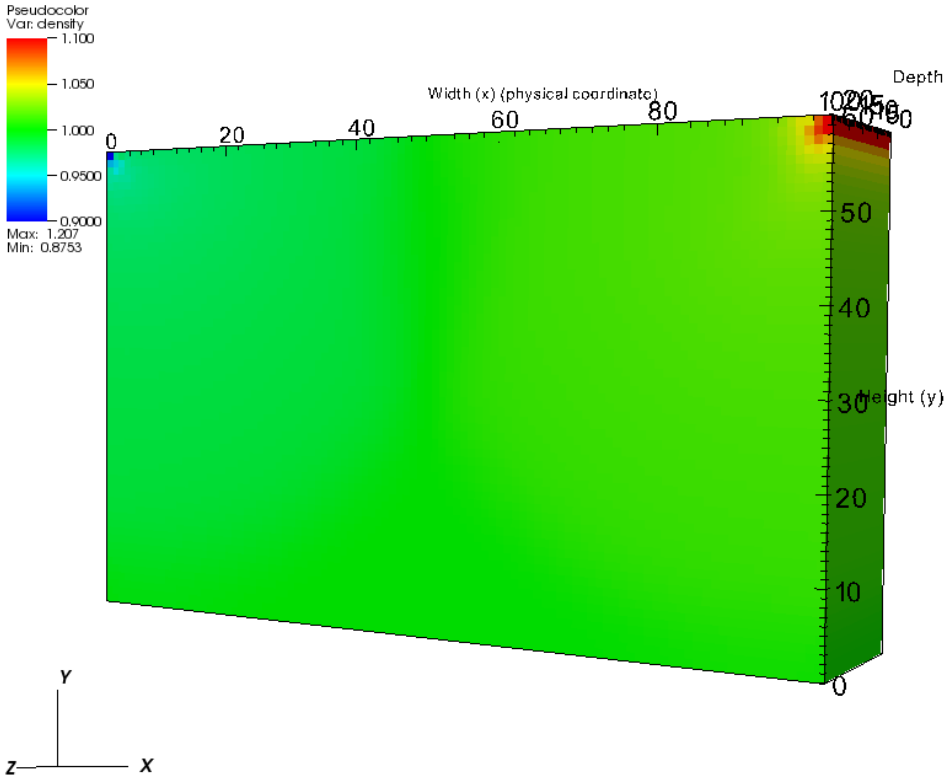

Figure 11: density pseudocolor plot

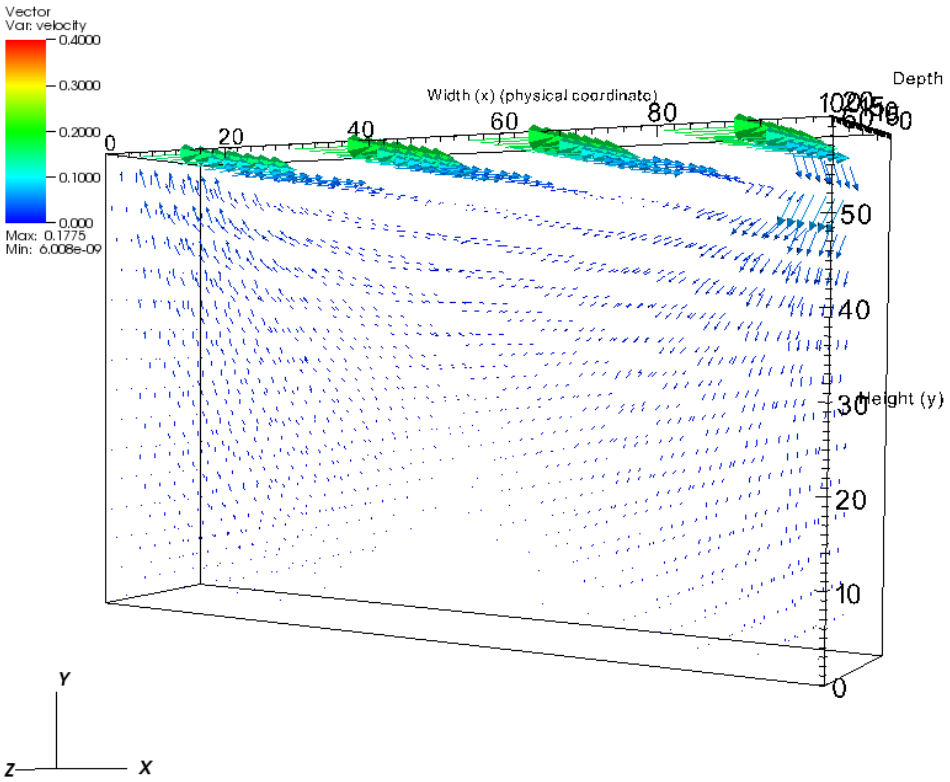

Figure 12: velocity vector plot

In the same timestep, there were two obstacles set, with the NoSlip boundary condition, which was done by using the "Draw Selection" and "Set Obstacle" commands. So first the selection was set via the box operator and then the cells inside the selection were set to contain the NoSlip boundary conditions. This can be seen in figure 13. It shows the velocity plot, shown before, and a pseudocolor plot of the FlagField. The threshold operator was applied to this plot. This operator restricts the value range, which the plot contains. In this case only cells are displayed, which contain the flag for the NoSlip boundary condition (four in this case), in order to visualize the obstacle. Also the current selection area can be seen. It is visualized by a mesh plot of the area and the currently active box tool, which is used to manipulate the box operator with the cursor.

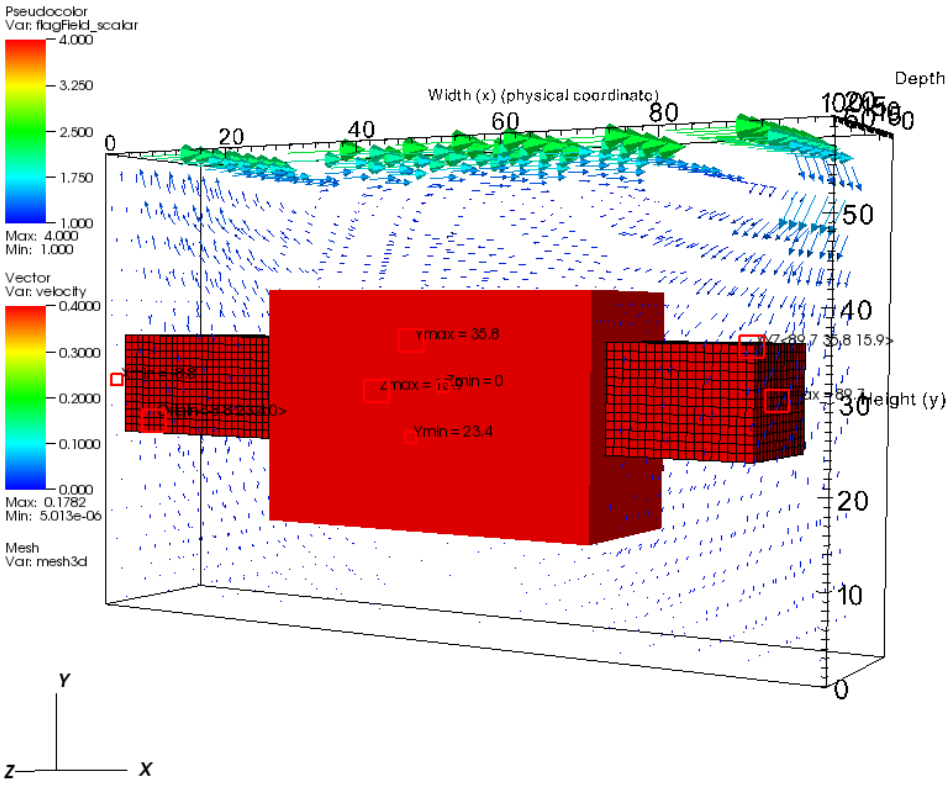

Figure 13: after placing two obstacles

Figure 15 shows the simulation after 200 further timesteps. It is visualized how the fluid flows around the obstacle and how density and velocity values are influenced by it. Figure 14, shows VisIt's Plots window, which contains all plots that are currently used. The threshold operator was applied to the density and velocity plots. It can also be set up to use a different variable than the plot it is assigned to. In this case it removes all cells from the density and velocity plots, which contain a boundary condition in the flag field. Furthermore the isosurface operator was applied to the density plot. An isosurface is a surface, on which every point has the same value. So the 3D data is sliced into such surfaces.

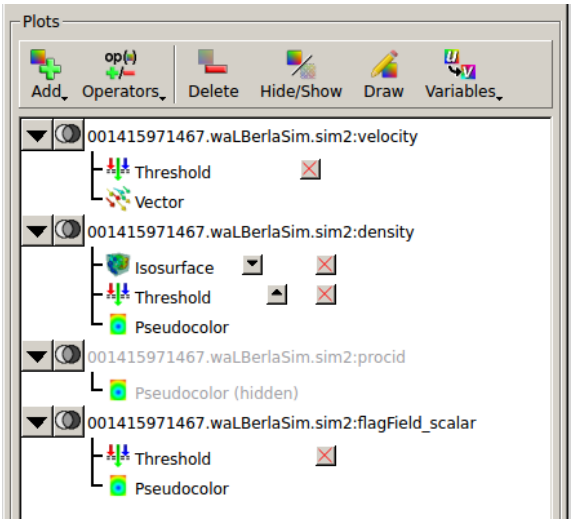

Figure 14: VisIt's plot window

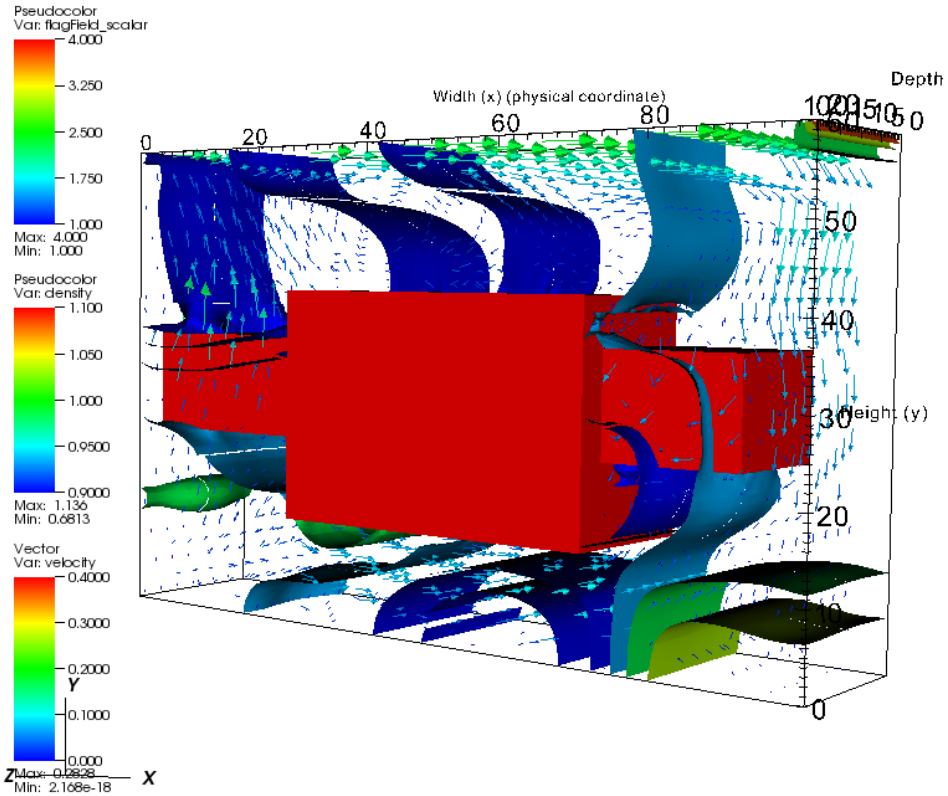

Figure 15: density isosurface plot after 300 timesteps

Finally density and velocity values were set. This can be seen in figure 16 (density = 1100 and velocity = <150, 150, 100>). For that, a new selection area was specified, via the box operator, in the bottom right corner. Every cell in the selection area contained the specified value, after the command to set the variables was executed. The impact on the simulation (40 timesteps later) can be seen in figure 17. The higher velocity and density values created turbulence, which will relax in further timesteps.

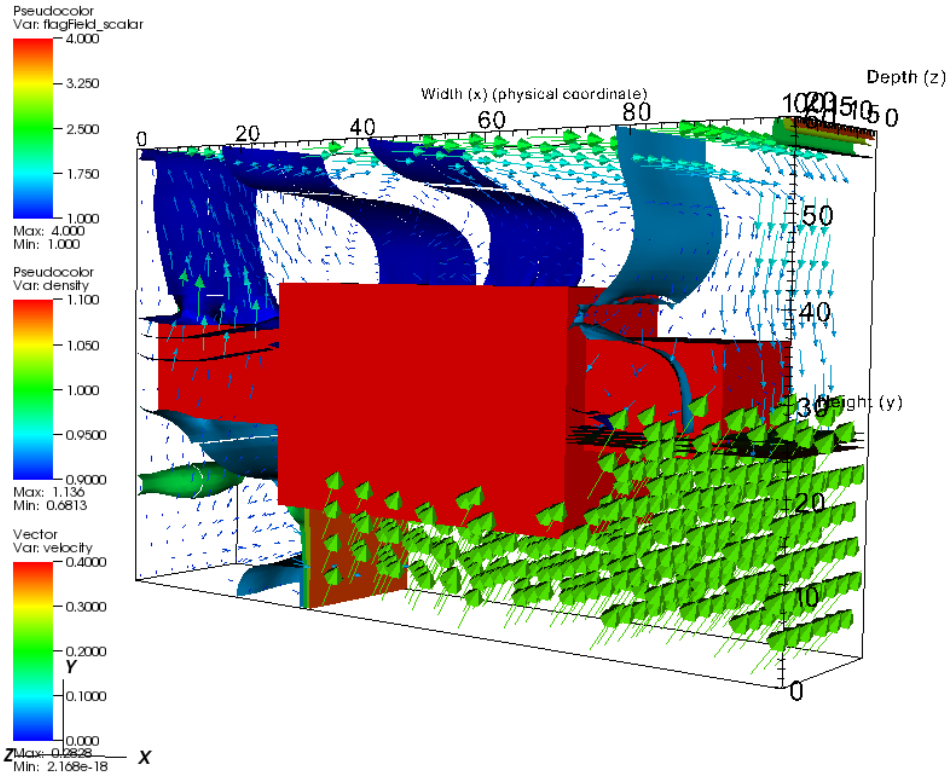

Figure 16: setting density and velocity variables

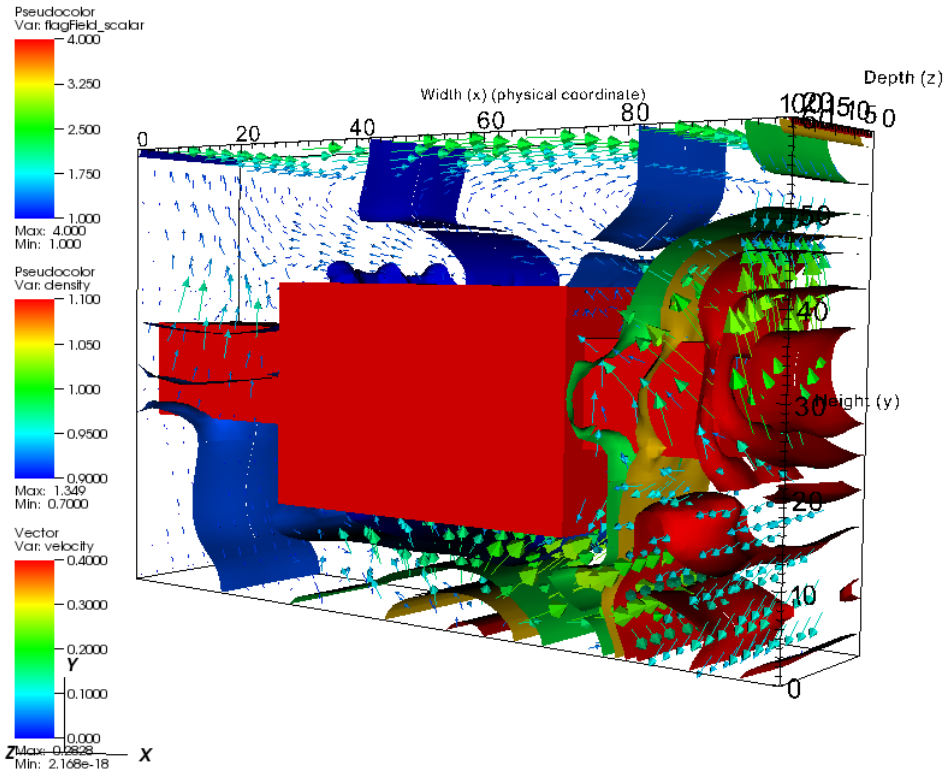

Figure 17: after setting the variables

7 Future Work

The implementation currently has the following limitations:

1. WaLBerla's fields internally use padding to store the data, in order to increase performance. The VisIt functions, that are used in the implementation for setting variable data, use an array without padding. So the current solution is to simply copy the data into a new array, which costs computation time and memory. This could be optimized by finding a way to expose the data without copying or at least speed up the copy process.

2. The PdfFieldInteractions class can only manipulate PdfFields. Other subclasses of the FieldInteractions class could be implemented, in order to be able to manipulate other types of fields. The PdfFieldInteractions class currently assumes, that there is a flag for the boundary condition with the name "NoSlip". If this flag is not found, no boundaries are updated. The code could be extended, so that other boundary conditions can be set.

3. Currently, the callback functions for the Qt-widgets can only register integers, as VisIt provides only callback functions for Qt-widgets, which use integers.

4. Another problem arises from the domain decomposition. In cases where interpolation is needed to perform a visualization operation, the connections between the subdomains become visible. In order for VisIt to interpolate correctly, one has to provide additional ghost data. There are functions for sending ghost data to VisIt, which are described in the Getting Data Into VisIt manual [21]. These could be implemented in the future.

5. Furthermore, as mentioned earlier, waLBerla is able represent the blocks in an octree. The current implementation is only able to visualize blocks with a fixed cell size. This is because a VisIt rectilinear mesh was used, which is populated with the cell data. However, VisIt offers more complex mesh types, which should be able to represent data, with changing cell sizes. The AMR (**A**daptive **M**esh **R**efinment) mesh could be used, which is a structured mesh in which rectangular parts can be subdivided in regions where more detail is required [21].

References

[1] E. Wes Bethel, Hank Childs, and Charles Hansen. *Chapman and Hall/CRC Computational Science : High Performance Visualization : Enabling Extreme-Scale Scientific Insight*. Chapman and Hall/Crc Press, Boca Raton, FL, October 2012.

[2] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94:511–525, May 1954.

[3] Hank Childs, Eric Brugger, Brad J Whitlock, Jeremy Meredith, Sean Ahern, Kathleen Bonnell, Mark Miller, Gunther H. Weber, Cyrus Harrison, David Pugmire, Tom Fogal, Christoph Garth, Allen Sanderson, E. Wes Bethel, Marc Durant, David Camp, Jean M. Favre, Oliver Rubel, Paul Navratil, Matthew Wheeler, and Paul Selby. Visit: An end-user tool for visualization and analyzing very large data. In E. Wes Bethel, Hank Childs, and Charles Hansen, editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, Chapman & Hall, CRC Computational Science, pages 357–372. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, October 2012.

[4] Hank Childs, Kwan-Liu Ma, Hongfeng Yu, Brad Whitlock, Jeremy Meredith, Jean Favre, Scott Klasky, Norbert Podhorszki, Karsten Schwan, Matthew Wolf, Manish Parashar, and Fan Zhang. In Situ Processing. In E. Wes Bethel, Hank Childs, and Charles Hansen, editors, *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, Chapman & Hall, CRC Computational Science, pages 171–198. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, October 2012.

[5] Hank Childs and Mark Miller. Beyond meat grinders: An analysis framework addressing the scale and complexity of large data sets. In *SpringSim High Performance Computing Symposium (HPC 2006)*, pages 181–186, 2006.

[6] Henry R. Childs, Eric Brugger, Kathleen S. Bonnell, Jeremy S. Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *IEEE Visualization'05*, 2005.

[7] Christian Feichtinger, Stefan Donath, Harald Köstler, Jan Götz, and Ulrich Rüde. Walberla: Hpc software design for computational engineering simulations. Technical Report 3, Chair for System Simulation, Erlangen, 2010.

[8] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56:1505–1508, Apr 1986.

[9] I. Ginzburg, F. Verhaeghe, and D. d'Humieres. Study of simple hydrodynamic solutions with the two-relaxation-times lattice boltzmann scheme. *Communications in computational physics*, 3:519 – 581, 2008.

[10] Christian Godenschwager, Florian Schornbaum, Martin Bauer, Harald Köstler, and Ulrich Rüde. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 35–1 – 35–12, 2013.

[11] Lawrence Livermore National Laboratory. *VisIt User's Manual*, October 2005. version 1.5.

[12] 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.

[13] Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, PVG '01, pages 85–92, Piscataway, NJ, USA, 2001. IEEE Press.

[14] Christoph Pflaum. lecturenotes simulation and scientifc computation. `https://www10.informatik.uni-erlangen.de/~pflaum/pflaum/SiwiR_II/skript_siwir.pdf`. Accessed: 2014-11-17.

[15] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[16] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[17] M. Summerfield. *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*. Pearson Education, 2007.

[18] Nils Thürey, Thomas Pohl, Ulrich Rüde, Markus Oechsner, and Carolin Körner. Optimization and stabilization of lbm free surface flow simulations using adaptive parameterization. *Computers and Fluids*, pages 934–939, 2006.

[19] Official visit webpage. `https://wci.llnl.gov/simulation/computer-codes/visit`. Accessed: 2014-10-27.

[20] Official walberla webpage. `http://www.walberla.net/index.html`. Accessed: 2014-10-27.

[21] Brad Whitlock. *Getting Data Into VisIt*. Lawrence Livermore National Laboratory, July 2010. version 2.0.0.

[22] 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*, EGPGV '11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[23] Brad Whitlock, Jakob van Bethlehem, and Hank Childs. *VisIt Python Interface Manual*, 2011. version 2.5.2.

[24] Dieter Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction*, volume 308. Springer, Berlin, 2000.