RWTH Aachen University

Diploma Thesis

CUDA-Based Particle Tracing in Time-variant Tetrahedral Grids

by

Michael Bußler

RWTH Aachen University

Diploma Thesis

CUDA-Based Particle Tracing in Time-variant Tetrahedral Grids

for the degree of Dipl.-Inform. in Computer Science

by

Michael Bußler
Student Id.: 252 137

Prof. Dr. Torsten Kuhlen
Lehrstuhl für Hochleistungsrechnen
Virtual Reality Group

Prof. Dr. Leif Kobbelt
Lehrstuhl für Computergrafik

Supervisor: Dipl.-Inform. Tobias Rick

Date of issue: 27.08.2010

Statement

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig im Rahmen der an der RWTH Aachen üblichen Betreuung angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I guarantee herewith that this thesis has been done independently, with support of the Virtual Reality Group at the RWTH Aachen University and no other than the referenced sources were used.

Aachen, August 27, 2010 .....

Contents

CHAPTER 1

INTRODUCTION

The tracing of massless particles through the unsteady flow field of a time-variant domain is a convenient way to visualize the flow data resulting from computational fluid dynamics. This thesis presents an approach for the particle tracing in time-variant domains resulting from real-world simulations, whereas the movement of the particles is calculated entirely on the GPU using the CUDA framework. The simulation data is given by several temporal states of the time-variant domain which are represented each by a tetrahedral grid with embedded flow field velocities. Inside an immersive Virtual Reality environment, the time-variant flow can be interactively explored while an intuitive user interface is provided at the same time for the creation of particles within domain under investigation.

1.1 Motivation

With the rising capabilities of computer systems, the complexity of models for modern numerical simulations increases which leads to an immense amount of data resulting from the calculation. Especially for the results of computational fluid dynamics (CFD) simulations, which are normally given as a discrete distribution of flow field velocities, an appropriate scientific visualization technique is required in order to depict the behavior of the simulated fluid in a meaningful manner and to detect and analyze important flow features and characteristics. The usage of a Virtual Reality (VR) environment hereby alleviates the interactive exploration as it allows to immerse into the visualization of the time-variant flow by providing an almost holographic depiction while also offering an intuitive user interface with full six-degrees-of-freedom.

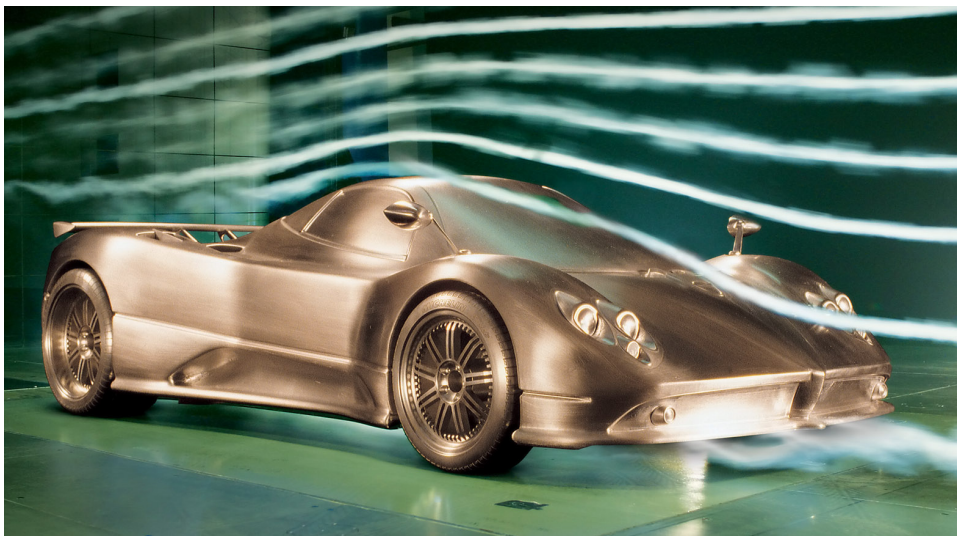


Figure 1.1: Real-world examination of a sports car prototype in a wind tunnel [Pag05]. The airstream is visualized by smoke released from several dies.

While the flow calculation using CFD simulation models can not completely replace real-world flow experiments, e.g. the examination of a prototype in a wind tunnel while visualizing the airflow with an almost massless matter like smoke (cp. Figure 1.1), the synthetic reproduction of such settings is much cheaper and less time-consuming while also providing additional degrees of interaction, e.g. by stopping the progression of the simulation time in order to analyze the three-dimensional scene from various perspectives. In addition, the flow phenomena within certain highly dynamic processes are only hard to observe, like the injection and compression of the combustible mixture within an internal combustion engine. Such processes can be examined much easier using simulated models, in view of the fact that the calculation of the compressible fluid using CFD gives a highly accurate reproduction of the real flow.

In order to achieve a visualization quality almost equivalent to real-world experiments, the simulated flow is illustrated by animating the movement of a vast amount of massless particles, whose trajectories are traced through the flow field of discrete velocities (cp. Figure 1.2). The particles may be seeded at any location within the flow field domain and at an arbitrary point in time within the simulation time frame in order to explore the unsteady flow within different regions of the domain over time. The seeded particles immediately follow the flow, just like smoke emitted from a die.

The advection of a large particle population introduces high computational cost, especially for simulated flow field domains discretized by unstructured grids and defined by several temporal states resulting in a high data volume. Especially in an immersive VR environment, it is crucial to achieve interactive frame rates at all times, which allows only a small time frame for the calculation of the particle's movement, a task of which mostly exceeds the capabilities of CPU-based particle tracers. Therefore, the approach

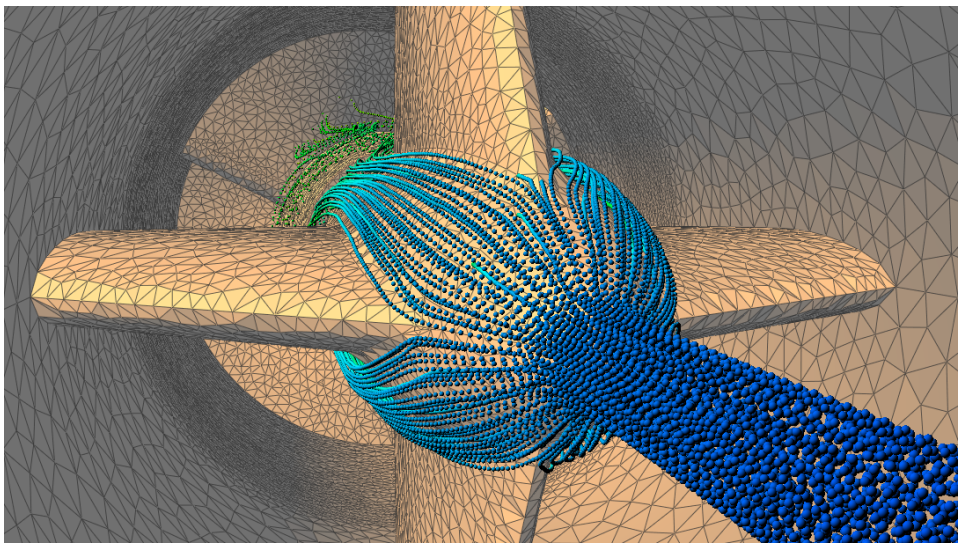


Figure 1.2: Visualization of a simulated flow field depicted by the movement of massless particles.

for particle tracing in unstructured time-variant tetrahedral grids presented in this thesis exploits the capabilities of today's programmable graphics hardware which features many core processors with highly parallel, multithreaded computational power by using the CUDA framework for the implementation.

1.2 Contributions

This thesis aims at the interactive exploration of complex, unsteady flow fields embedded in time-variant tetrahedralized domains by simultaneously achieving interactive frame rates for the visualization in immersive virtual environments. The following contributions are presented:

- The dealing with the original or decimated dataset of time-variant flow field domains represented by several tetrahedral grids whereas each grid describes the state of the domain for one instant in time.

- An efficient two-phase point location scheme including the traversal of a kd-tree, which is entirely performed on the graphics hardware.

- A detailed description of the flow field integration process using different numerical integration methods as well as an adaptive step size adjustment procedure based on the embedded Dopri-5 integration scheme.

1.3 Outline

This thesis is structured as follows. Chapter 2 discusses the simulation domain discretization using unstructured grids with embedded flow field velocities and presents related work about interactive flow field visualization. Chapter 3 describes the tetrahedral grid data structure in detail and presents a solution to the point-location problem on such grids using an efficient two-phase search procedure, whereas different methods for the kd-tree traversal are described and compared in terms of run duration, accuracy and implementation effort. Chapter 4 describes the particle advection procedure which requires the integration of the time-variant flow field. Several approaches for the adaptive adjustment of the integration step size are discussed and some performance figures are given for the flow field integration using different integration schemes when performed either on the CPU or the GPU as well as for different step sizes and particle counts. An introduction to the CUDA programming model featuring a comparison of the CPU and GPU architectures as well as a detailed description of the GPU-based implementation of the entire particle tracing process is given in Chapter 5, including the usage of the ViSTA VR Toolkit to allow the interactive exploration within an immersive virtual reality environment. Chapter 6 examines the usability of the interactive particle tracing approach on the real-world Engine dataset by measuring the performance of the implementation for different particle populations. Chapter 7 finally gives a conclusion of the thesis and a perspective for future work.

CHAPTER 2

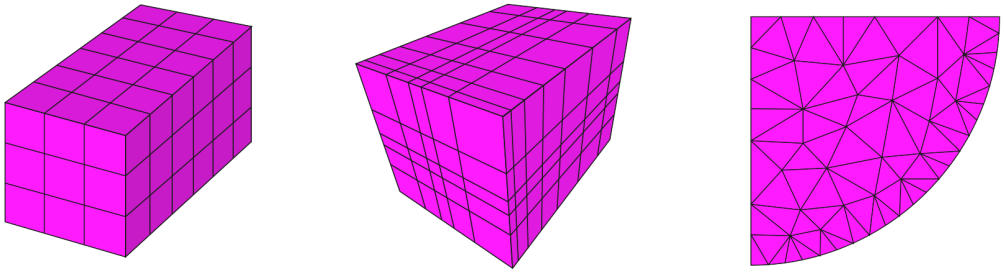
FOUNDATIONS AND RELATED WORK

2.1 Simulation Domain Discretization

The kind of simulation data regarded in this thesis is the result of computational fluid dynamics (CFD), one of the branches of fluid mechanics, which allows calculating the flow field of a fluid using a numerical method, e.g. the Navier-Stokes equation. CFD is widely used for the simulation of fluids, e.g. the airflow in a wind tunnel in order to analyze the aerodynamic behavior of prototypes. Simplified CFD simulations are also used in films or video games to create special effects.

The equations used for CFD describe the continuous behavior of a fluid. One of the key challenges in computing such a fluid is how to discretize the domain in which the simulation is calculated. This can be done by dividing the domain into small cells that form a volumetric grid. The flow of the fluid is then calculated for each node of the grid, resulting in an approximation of the continuous flow field as a solution of the flow equation.

The simulation data regarded in this thesis is calculated on domains discretized by tetrahedral grids. Those grids belong to the class of unstructured grids, which consist of a set of points, the grid nodes, and an additional index structure that describes how the nodes are connected to form the grid cells. In tetrahedral grids, those cells are geometric primitives called tetrahedrons. In contrast to regular grids, where all cells are equally shaped, tetrahedral grids have the significant advantage, that the shape of the cells can be arbitrarily chosen. Some parts of the domain can be simulated at finer scale than others and round shapes can be adapted by using an appropriate distribution scheme

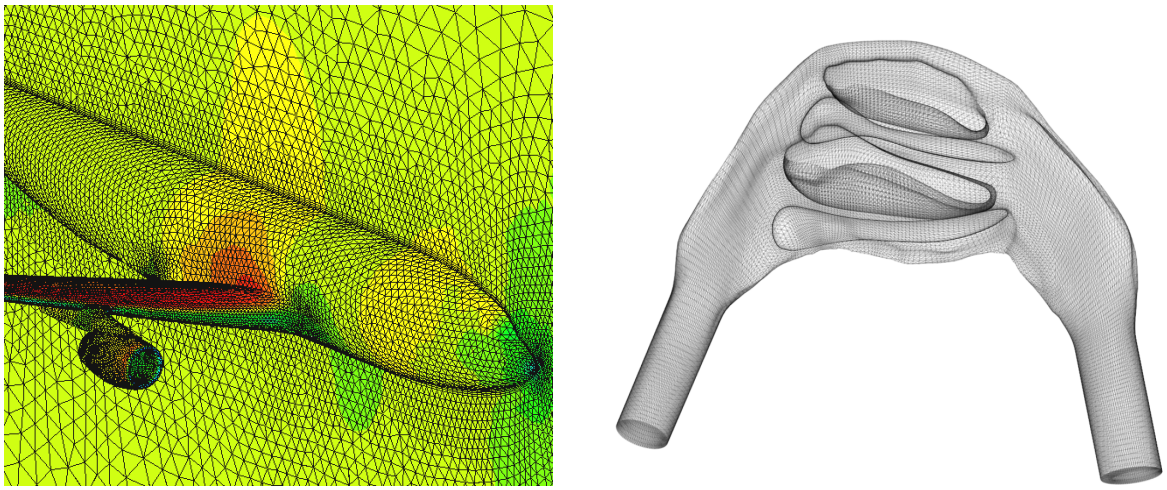


(a) (b) (c)

Figure 2.1: Examples of different domain discretizations using regular (a), rectilinear (b) or unstructured grids (c) [Wik09].

for the grid nodes, which also affects the resolution of the grid whereas switching to a finer resolution in a regular grid affects the size of all grid cells (cp. Figure 2.1).

As a further advantage, unstructured grids can smoothly adapt to the boundaries of the domain as well as to the outer hull of tessellated objects situated inside the domain. As an example, Figure 2.2a shows the simulated pressure distribution for a tessellated aircraft model embedded in a tetrahedralized domain [VDBO97]. Unstructured grids are especially useful for domains with a complex shape, e.g. with a genus greater than zero. By using an unstructured grid for the domain discretization, the grid nodes are only distributed on the hull and within the domain, whereas many of the grid nodes would be situated outside of the domain, if it is discretized using a regular grid. (cp. Figure 2.2b).



(a) (b)

Figure 2.2: By using tetrahedral grids for the domain discretization, the boundaries of tessellated objects within the domain can be smoothly adapted, e.g. an aircraft in a wind tunnel [VDBO97] or the complex shape of the human nasal cavity [Sch08].

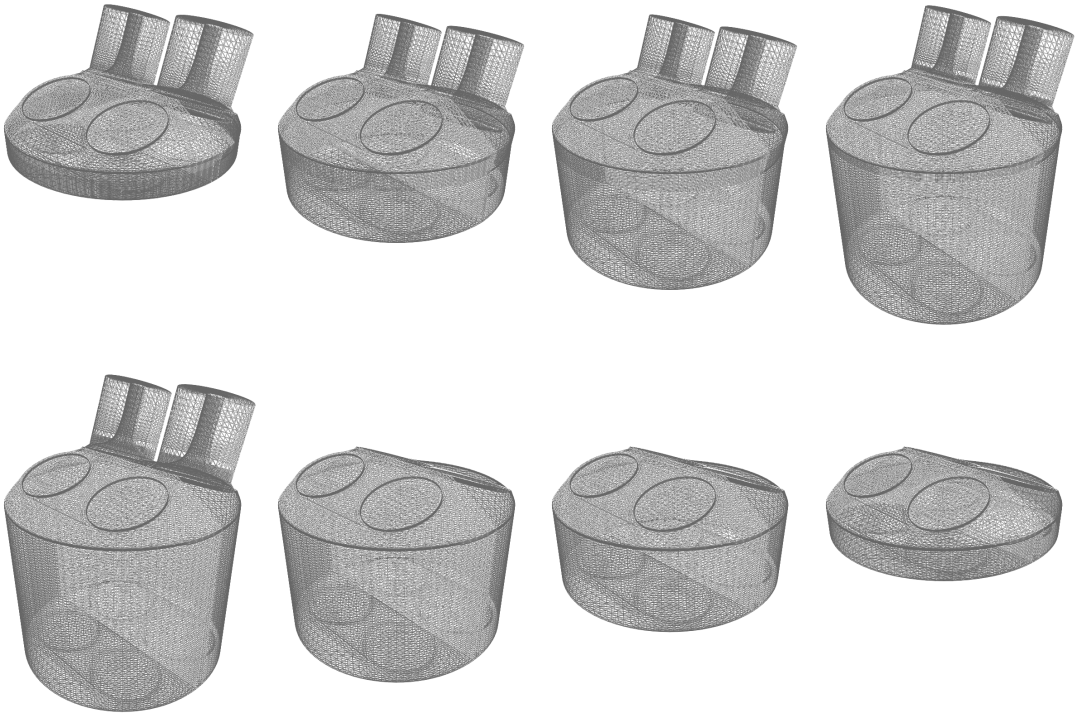


Figure 2.3: The Engine dataset [Abd98] describes the flow of the combustible mixture inside a four-stroke internal combustion engine over several temporal states.

The methods presented in this thesis consider the original or decimated simulation data of time-variant tetrahedralized domains. Those domains dynamically change their appearance over time to account for the simulation of highly dynamic processes. This implies, that also the discretization of the domain changes over time. To handle this alteration, the regarded time-variant domains are described by several temporal states, where each state is represented by a tetrahedral grid with its own respective grid node distribution, describing the state of the domain and the flow for a specific point in time.

As an example dataset, resulting from the computation of an unsteady flow within a time-variant domain, Figure 2.3 depicts some of the temporal states of a simulated four-stroke internal combustion engine [Abd98], courtesy of the Institute of Aerodynamics (AIA) at RWTH Aachen University. This dataset simulates the flow of the combustible mixture while the intake and compression stroke. Despite the opening and closing of the engine's valves, the size of the combustion chamber changes, as the piston moves from the top of the cylinder to its bottom, which leads to a highly dynamic process simulated over several temporal states. Each domain state of the Engine dataset is discretized by a tetrahedral grid with respective grid node distribution and cell shapes.

2.2 Time-variant Flow Field Velocities

The unsteady flow field of a time-variant tetrahedralized domain is defined at the nodes of several tetrahedral grids, each one representing the state of the flow for one instant in time. It is calculated by computational fluid dynamics which yields the flow for each domain state, e.g. as a solution of the Navier-Stokes equation.

The simulation data embedded in a tetrahedral grid can be described as a mapping

$$u(\mathbf{x}, t) : \Omega \times \Pi \rightarrow \mathbb{R}^M \tag{2.1}$$

of discrete locations $\mathbf{x}$ within the flow domain $\Omega \subseteq \mathbb{R}^n$, e.g. the nodes of the grid, and discrete time values $t \in \Pi \subseteq \mathbb{R}$ defined by the temporal states of the domain to M-dimensional attributes, where M equals three for the flow velocity data.

In order to calculate the flow field velocity for positions in between the grid nodes, the velocity defined at the grid nodes needs to be interpolated. To determine, which nodes to consider for the velocity interpolation, the cell structure of the tetrahedral grid is used. Hence, the surrounding cell of the position to evaluate the flow field velocity for needs to be estimated. If that cell is known, the velocity of the respective position is calculated by interpolating between the velocities of the four grid nodes forming the tetrahedron cell.

Several tetrahedral grids are used to describe the time-variant nature of the regarded domains. This also affects the simulated flow, which is defined separately for each temporal state and changes over time. To account for the altering flow field velocities, a temporal interpolation for the position to evaluate needs to be performed. This is realized by interpolating between each two consecutive temporal domain states, as the exact solution of the CFD is only known at that discrete time intervals [Lan93].

As depicted in Figure 2.4, the position-dependent interpolation of the flow field velocities in each tetrahedral grid together with the temporal interpolation between two consecutive flow field states leads to a continuous distribution of the flow velocity over the entire domain, which can be evaluated for every point in time within the simulation time frame:

$$v(\mathbf{x}, t) : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3 \tag{2.2}$$

where $t_0 \leq t \leq t_N$ for a dataset with N distinguished temporal domain states. For positions outside of the flow field domain, the velocity is defined as $\vec{0}$ for each moment in time.

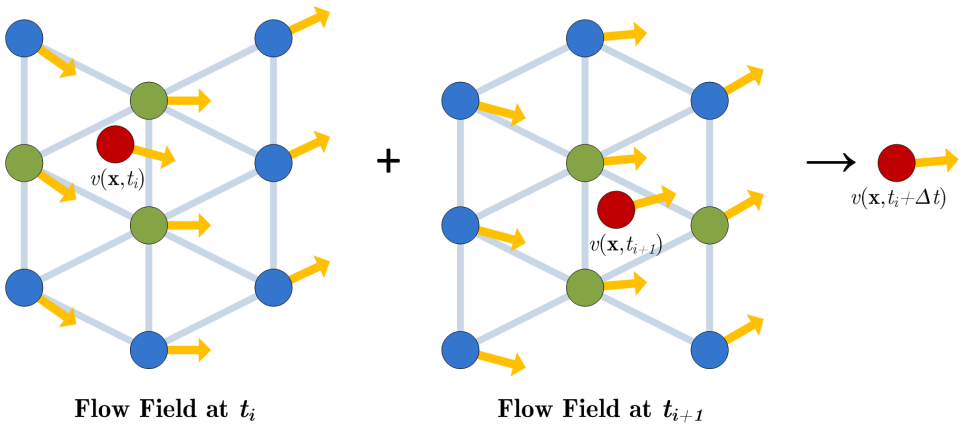


Figure 2.4: The flow field velocity for a position within the domain is obtained by interpolating between the discrete velocity data defined at the grid nodes and by interpolating between two consecutive domain states.

2.3 Interactive Flow Field Visualization

The goal of the interactive visualization of time-variant flow field data is to depict the behavior of the flow over time in a meaningful manner in order to detect and analyze important flow features and characteristics. The major problems to solve in this research area are the handling of large datasets resulting from complex simulations over several time steps, the intuitive interaction with the dataset and the computation time on unstructured grids.

A wide spectrum of flow field visualization techniques was developed for two- and three dimensional datasets according to different purposes [PVH+03]. An overview about the research that has been done in the field of integration-based geometric flow field visualization for the last two decades is given in [MLP+09].

An application for different visualization techniques on unstructured, time-varying CFD grids with adaptive resolutions is presented in [GLT+06], where different visualization techniques are presented to illustrate the swirl and tumble motion patterns of a simulated Engine dataset, including particle trajectories and stream surfaces [GTSS04]. The flow field visualization using stream surfaces can be enhanced by using texture-based advection [LGSH06], which maps a streamline-like flow illustration to the surface by filtering noise textures along the underlying flow field.

A similar approach was used by [VFWTS08] to represent the structure of smoke within turbulent flow fields as semi-transparent streak surfaces. [BFTW09] presents a technique to visualize unsteady flows as streak surfaces using adaptive flow field integration and rendering in real-time by utilizing the computational power of present GPUs. An alternative approach to interactively generate time- and streak surfaces on large time-variant

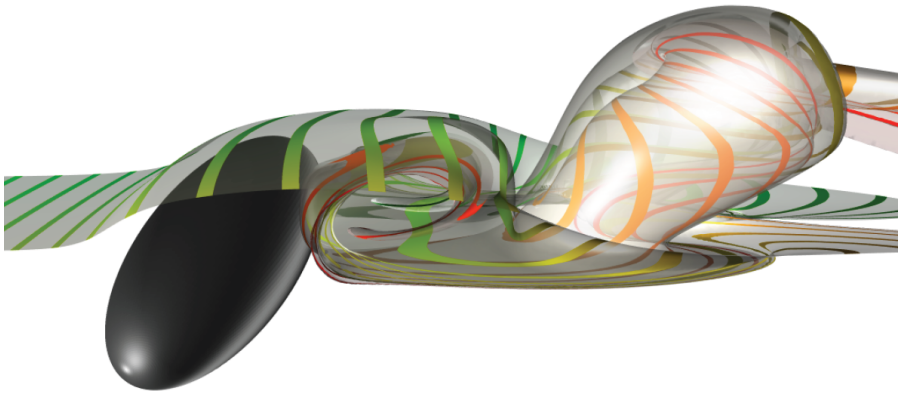


Figure 2.5: A textured streak surface illustrating the flow behind an ellipsoid [KGJ09].

datasets is presented in [KGJ09] (cp. Figure 2.5).

The methods to generate streak surfaces are based on the visualization technique used in this thesis, the interactive tracing of massless particles through the time-variant flow field [BPSS02], a kind of visualization approach, which is adopted from the flow analysis of real world experiments, as presented in [SL00], for example. This method enables the user to intuitively explore the flow field of unsteady domains. Particles can be seeded at every point inside the domain using different seeding strategies. The seeded particles are immediately carried away by the flow, similar to a torch, which emits smoke in a real turbulent environment, or ink, injected to water. As the particles follow the flow, the continuous animation of their movement gives an intuitive impression of the flow behavior in the course of time.

Despite the depiction of the instantaneous particles' positions, additional information is obtained from depicting different kinds of particle trajectories, which also depends on the handling of time information:

- *Streamlines* are the tangents of the flow field velocities for one moment in time. They change their entire appearance over time in case of a time-varying flow field.

- *Pathlines* depict the paths of the particles through the time-varying flow field. They give an impression about how the particle trajectories change over time.

- *Streaklines* are created by connecting the positions of particles seeded continuously at a fixed position. The visual appearance of the streaklines is similar to the continuous release of smoke from a die in real wind tunnel experiments.

- *Timelines* are connections between particles, which were released at the same moment in time, but at different positions.

The advection of particles through the flow domain requires numerical integration of the flow field velocity as well as adaptive step size adjustment. The integration should be performed using a higher-order method like the third- or fourth-order Runge-Kutta integration scheme, as the simple first-order Euler integration shows some inadequacies [Bun88]. To calculate a higher-order accurate integration of the flow field, several evaluations of the flow field velocity are needed, which are computational expensive on unstructured grids. A specialized Runge-Kutta integrator for steady unstructured flow fields was introduced by Ueng in [USM96]. His approach requires some pre-processing, but the calculation of the fourth-order accurate solution requires only a matrix-vector multiplication and a vector-vector addition.

Also for steady unstructured vector fields, Kenwright and Mallinson [KM92] presented a method which completely avoids the flow field integration by calculating the intersection of two stream surfaces. This yields the exact path of a particle that is located at the intersection. Another approach to avoid the integration calculation in steady vector fields is presented by Kipfer in [KRG03] as an extension of the locally exact path calculation approach of Nielson and Jung [NJ99]. Kipfer's approach describes the interpolation within a tetrahedron cell as a linear mapping and uses Eigen decomposition to calculate the exact points where a particle enters and leaves a cell. Unfortunately, the locally exact particle tracing approaches cannot easily be adopted to time-variant domains as they do not account for the temporal interpolation.

In [KL96], Kenwright and Lane describe a method to compute particle trajectories in time-varying flow fields on moving curvilinear grids, which are decomposed into tetrahedrons. They also present a solution to the point-location problem on tetrahedral grids, which is similar to the cell searching method presented in [USM96]. Their method computes the natural coordinates for a point w.r.t. the cell, in which the point is located. The natural coordinates can also be used to interpolate the simulation data at that point, which gives equal results compared to physical space interpolation and volume weighted interpolation.

A solution to the point-location problem on large unstructured grids is given by Langbein et al. in [LST03]. Their approach uses a kd-tree in order to index the grid nodes and to perform a fast nearest-grid-node search. A lookup-table is then used to find the cells that belong to a certain vertex. Still, the point-location problem on unstructured grids remains a challenging task, as a vast amount of particles is needed to convey a visual impression to some extend similar to real-world experiments.

Most of the current research in flow field visualization using particle tracing techniques relies on the computational power of modern graphics processing units (GPUs), to handle a huge amount of particles at interactive frame rates. The particle tracing approach of Krüger et al. [KKKW05] considers steady 3D flow fields on uniform grids and exploits the graphics rendering pipeline for GPU-based computations. The computations are hereby mapped to graphical elements and performed by invoking vertex- and fragment shader

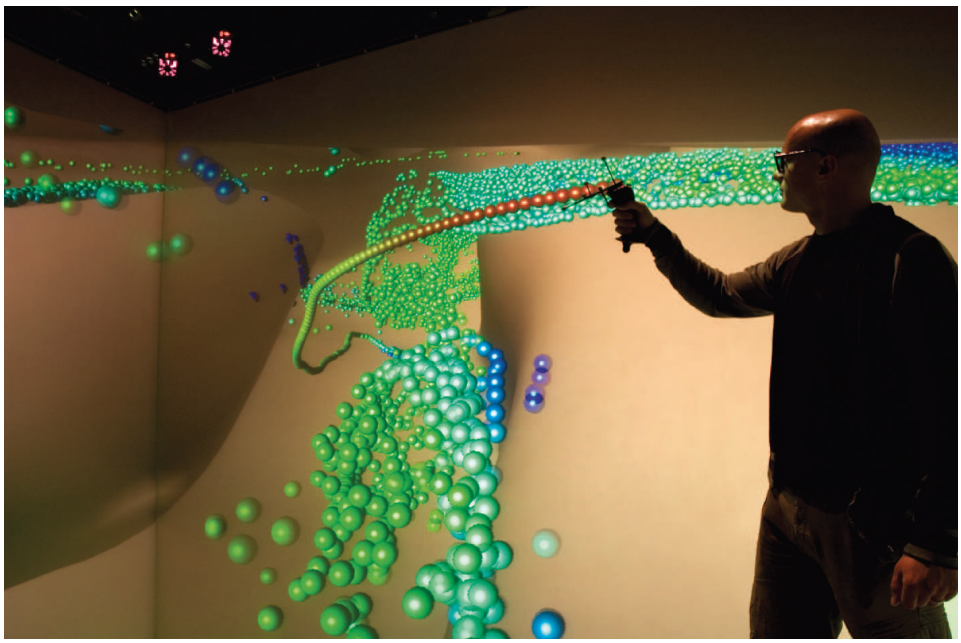


Figure 2.6: Interactive particle seeding in an immersive virtual environment [Sch08].

programs within the rendering process. By performing depth-sorting on the GPU, the transfer of particle data from the GPU to the CPU and vice versa is avoided.

In his Ph.D. Thesis [Sch08], Schirski presents a similar approach for GPU-based particle tracing using the graphics rendering pipeline. He introduces the usage of 3D textures to store the uniform grid of unsteady flow fields, which speeds-up the flow field integration process, as velocities are directly interpolated within graphics hardware. Schirski also presents a method for particle tracing on steady tetrahedral grids with time-varying flow fields, as well as an efficient particle rendering approach for immersive VR environments using user-centered billboards for the depiction of the instantaneous particle positions (cp. Figure 2.6).

When dealing with GPU-based particle tracing, a major problem is the limited amount of memory provided by graphics devices. In his discussion about time-varying flow fields, Schirski states, that the streaming of data to the GPU is a non-avoidable bottleneck, as all data must pass the shared system bus between the host system and the graphics adapter. He presents a demand-driven method featuring region-of-interest, to handle large datasets using a HPC back-end and to convert tetrahedral grids into uniform grids reducing the data that needs to be streamed to the GPU in order to calculate the particle advection.

To face the problem of the bottleneck between the CPU and the GPU when transferring large tetrahedral grids, Weiler [WMKE04] describes a method to reduce the amount of tetrahedron cell data by storing the cells as texture-encoded strips, similar to how

it is done with triangular strips. Another approach to reduce the amount of data is the decimation of the initial tetrahedral grids [VGVW99], whereas the simulation data embedded in the grid needs to be considered as a factor for the decimation operator, as proposed by Chopra and Meyer [CM02] for scalar values. Still, how to perform tetrahedral grid decimation on a HPC back-end remains an open task.

CHAPTER 3

TETRAHEDRAL GRID PROCESSING

This chapter describes the methods that are used to handle the simulation data represented as unstructured tetrahedral grids. The storage of those grids is depicted and an efficient localization scheme is presented, which is used to find the surrounding cell of an arbitrary position within the grid.

3.1 Tetrahedral Grids

In this thesis, time-variant flow field domains are considered, which are discretized by tetrahedral grids. As each tetrahedral grid describes the state of the simulated domain for a certain point in time, time-variance is achieved by using several tetrahedral grids. Therefore, every point in time introduces a complete dataset including the additional overhead for the search structure that is used for the point location within the unstructured grid. This typically leads to a high amount of data that needs to be dealt with. For example, a typical tetrahedral grid for the simulation of computational fluid dynamics can raise the memory requirements to store a single temporal state of the flow field domain to one gigabyte and above. This exceeds the memory capacity of today's graphics card's memory. Such datasets need to be restructured in order to fit the memory limitations that are introduced by GPU-based computations.

The state of the flow field for one instance in time is defined as a discrete distribution of velocity vectors. This distribution results from the positions of the grid nodes, as the flow field velocities are defined at those positions. To evaluate the velocity at an arbitrary location within the flow field domain, the velocities at the grid nodes are

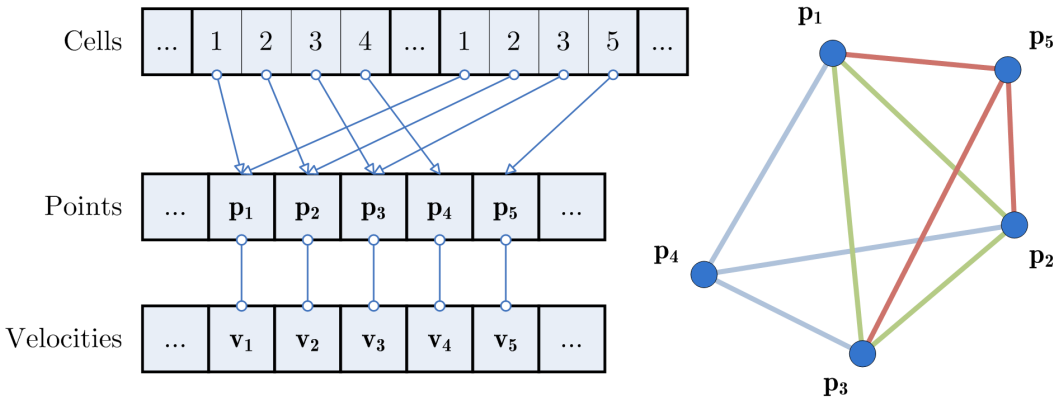


Figure 3.1: A Fragment of a tetrahedral grid which consists of five vertices and two adjacent cells sharing a common face.

linearly interpolated, which leads to a continuous distribution of the flow field velocity. To decide, which grid nodes need to be considered for the velocity interpolation, the cell structure of the tetrahedral grid is used. If the cell which includes the position to evaluate is known, the interpolation can be calculated by determining the natural coordinates of the position w.r.t. the cell nodes. The natural coordinates are then used as factors to weight the velocities defined at the four cell nodes.

For the internal representation, a tetrahedral grid consists of a list of vertices and a list of references, that describe how the vertices are connected to form the tetrahedron cells, as well as a list of velocity vectors defined at the grid nodes. The references are stored for each cell as the indices of the four nodes that belong to the cell (cp. Figure 3.1). As adjacent cells share common nodes, the number of cells is typically much higher than the number of nodes.

In graphics device memory, data structures should be aligned to 16 bytes in order to maximize efficiency of the memory access and therefore maximize memory throughput [NVI10a]. Thus, for tetrahedral grids stored in graphics card memory, coordinates and velocities are stored using 4 floating-point values, whereas each floating-point and integer value requires 4 bytes of memory. As only three float values need to be stored for three-dimensional point and vector coordinates, the fourth component can be used to store an additional per-vertex scalar. Altogether, an unstructured tetrahedral grid for one instance in time with n nodes and m cells is internally represented as:

- An array of coordinates for the grid nodes, represented by $4n$ floating point values.
- An array of velocities, which is also represented by $4n$ floating point values.
- An array of indices, which consists of $4m$ integer values and represents the tetrahedron cells.

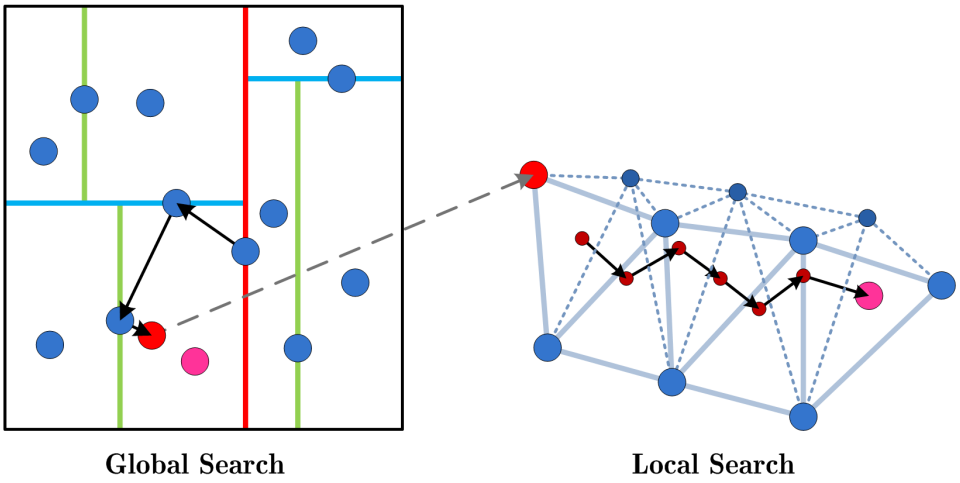


Figure 3.2: To locate the surrounding cell of a query position, a two-phase point location scheme is used. First, a global search is performed using a kd-tree to find a grid node near the query positions. Then a short tetrahedral walk is performed to locate the surrounding cell.

Hence, the total amount of memory, that is needed to store a dataset of k time steps is $k \cdot (32n_j + 16m_j)$ bytes, where n_j is the number of grid node and m_j the number of tetrahedron cells of the j'th time step and $1 \geq j \geq k$. For example, a dataset with 50 time steps, 200K nodes and 1M cells requires approx. 1GB of storage.

3.2 An Efficient Point Location Scheme

Unstructured grids have many advantages when used to discretize a domain, like arbitrary point distribution and adaption of boundary cells to the domain's hull as well as to objects situated inside the domain, which makes this data structure favorable for CFD simulations. But this degree of freedom comes at a greater cost for the visualization of the results, as the particles used to visualize the flow field must be located inside the unstructured grid. Also, while the particles are advected through the flow field domain, the information which cell currently surrounds a certain particle, needs to be continuously updated.

As depicted in Figure 3.2, the point location scheme used in this thesis is divided into two phases. In the first phase, a rough localization within the unstructured grid is performed using a kd-tree, which yields a grid node near to the query position. A node-to-cell lookup table is then used to obtain a cell associated with the grid node. In the second phase, a short tetrahedral walk is performed to find the cell that contains the query position. This point location scheme is similar to the one presented in [Sch08].

The two-phase point location scheme introduces additional index structures for each tetrahedral grid:

- A kd-tree structure indexing the nodes of the grid to perform the nearest-grid-node search, which consists of a floating point array and an integer array, each with four bytes per entry, as well as a character array with one byte per entry, yielding a storage amount of $9 \cdot 2^{\lceil \log_2 n \rceil} \approx 9n$ bytes in total, if every leaf of the tree contains a single grid node.

- A list of cell neighbors to perform the tetrahedral walk. The list contains four integer values per cell and requires another $16m$ bytes for storage.

- An additional node-to-cell lookup table to query the cell index given a grid node index. As only one integer index per node is needed, the table requires $4n$ bytes within memory.

The additionally required memory per time step for the index structures is $\approx 13n+16m$ bytes in total. For the example given above, this results in approx. 2.5MB for the kd-tree and 15.3 MB for the cell neighbors per time step. The kd-tree and cell neighbors can be pre-computed and stored to speed-up the data loading.

3.2.1 Kd-Tree Construction

A kd-tree is a space-partitioning search structure, which is used to organize a set of k-dimensional points and to perform an efficient nearest-neighbor search of a given query position. Here, the kd-tree structure is used to organize the set of nodes of a tetrahedral grid, in order to find the nearest grid node of a given position inside the domain. The kd-tree construction procedure used in this thesis is similar to the one described by Langbein in [LST03]. The kd-tree is created by repeatedly defining planes, so that half of the points lie on each side of every plane, separating the current set of points into two half sets.

The tree is internally represented by three arrays (cp. Figure 3.3). The inner nodes of the tree are stored as two arrays: An array D of char values holding the dimensions of the splitting planes by which the point set was separated in each partitioning step, as well as a floating point array S, which stores the coordinate value of the splitting plane according to the chosen dimension. The third array L represents the leaves of the tree, in which the point indices of the grid nodes are stored.

The kd-tree is build-up recursively with an (optional) level factor to constrain the number of levels of the final kd-tree, which also affects the number of point indices in each leaf. The list of input nodes is copied, whereas the original index of every node of the

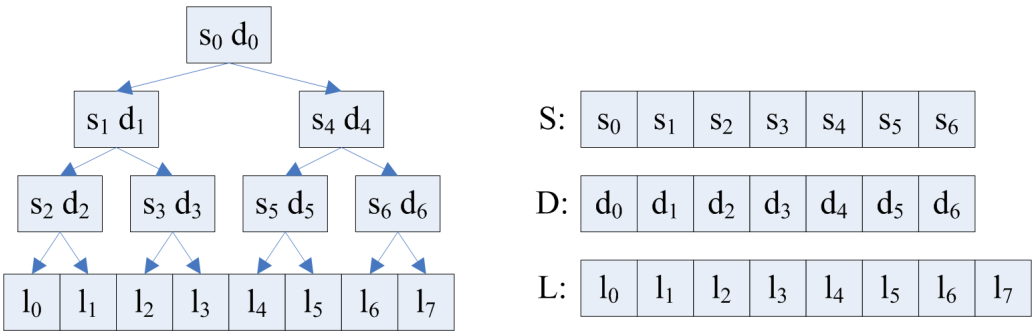


Figure 3.3: The kd-tree search structure is represented by two arrays for the inner nodes and another array for the leaf nodes.

tetrahedral grid is stored. If the number of input nodes is not a power of two, some of the original nodes at inserted twice to fill the gap.

As criteria for the split, Langbein proposes a mixture of the largest axis of the current box with the largest axis of the bounding box of thousand randomly chosen vertices inside the current box. These values are multiplied and the largest axis is chosen as the dimension of the split. Then, all vertices are sorted according to the split dimension using the *quickselect* method proposed in [Dev98]. The component of the median vertex according to the chosen dimension defines the plane, by which the point set is separated in this stage.

This procedure is repeated recursively for the two half-sets resulting from the split while the level factor l is decremented by one to give a breaking condition for the recursion. On the last level, where $l = 0$, the indices of the boundary nodes of the current bucket are stored as the leaves of the current sub-tree.

After the recursive build-up process has finished, the resulting tree structure is traversed breadth first in order to copy the actual split- and dimension values to the corresponding arrays S and D. Once a leaf node is reached, the point index stored in this leaf is appended to the index array L.

3.2.2 Kd-Tree Traversal

For the point-location scheme, the kd-tree index structure is used to locate the nearest grid node of a query point $\mathbf{q} = (x_q, y_q, z_q)$. This is done by traversing the tree structure and making a binary decision on each level of the tree which sub-tree to traverse next. Several methods exist, which describe how to traverse the kd-tree in order to find the nearest grid node. The methods use different approaches for the tree traversal procedure, which leads to certain advantages and disadvantages. In this thesis, three methods are

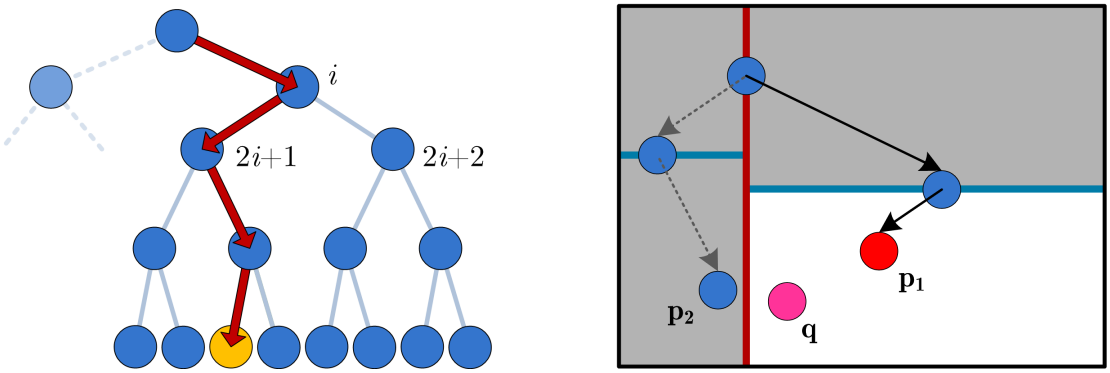


Figure 3.4: The Single-Pass nearest-neighbor search traverses the tree once from top to bottom. It finds a grid node near the query point, but not always the nearest one.

examined: The Single-Pass method, the Randomized method and the Backtracking method. All methods are distinguished in terms of complexity, runtime behavior and accuracy.

Single-Pass The Single-Pass nearest-neighbor search method is the simplest approach to traverse the kd-tree structure. As depicted in Algorithm 1, the method traverses the tree once, from top to bottom, starting at the root node of the tree with array index $i = 0$. On every level of the tree, the value of the current split plane, stored in S_i, is compared to the respective component q_{D_i} of $\mathbf{q}$ given by the dimension stored in D_i, which is either 0 for x_q, 1 for y_q or 2 for z_q. To advance to the next level of the tree, the index i is multiplied by two and an offset value is added to i, which is 1, if q_{D_i} is smaller than S_i or else 2 (cp. Figure 3.4a). Once the last level of the tree is reached, the index of the respective leaf node in L is calculated by subtracting the total number or inner nodes from i. The point with index L_i is then returned as the nearest grid node.

This method has a runtime of $O(\log n)$, where n is the number of points, stored in the tree. On every tree level, the set of possible grid nodes is halved. After $\log n$ steps the last level of the tree is reached, where the index of the nearest grid node is stored. The Single-Pass algorithm is very straightforward to implement, as it only iterates through the arrays once while incrementing an index variable. This is a desired behavior in a multi-threaded environment, as all threads starting together finish at the same time.

Unfortunately, the Single-Pass traversal method it is not guaranteed to find the nearest grid node to the query point $\mathbf{q}$. The decision, which part of the kd-tree is traversed next, relies solely on the position of the current splitting plane. There are occasional situations, in which the nearest grid node is located in the other half space that is not traversed. Such a situation is depicted in Figure 3.4b. Here, the algorithm returns $\mathbf{p_1}$ as nearest grid node, rather than $\mathbf{p_2}$, which is even nearer to the query point $\mathbf{q}$. Hence, to

Algorithm 1 SINGLE-PASS-NEAREST-NEIGHBOR-SEARCH($\mathbf{q}$)

Require: A kd-tree given as split values S, split dimensions D and point indices L.
Require: A query point $\mathbf{q}$.
Ensure: $\mathbf{p}$ is the nearest neighbor to $\mathbf{q}$.

```
i ← 0
for all tree levels do
  if q_{D_i} < S_i then
    i ← 2i + 1
  else
    i ← 2i + 2
  end if
end for
i ← i − 2^N
return Point p with index L_i
```

find $\mathbf{p_2}$ as the nearest neighbor, a procedure is required, which also traverses alternative paths in the tree. Still, as several experiments have shown, the grid node found by the Single-Pass traversal method is mostly quite near to the nearest neighbor.

Randomized The Randomized kd-tree traversal method proposed in [Pan08] is an enhanced version of the Single-Pass method. An approach called random restart is used to perform the kd-tree traversal several times (cp. Figure 3.5a). In every iteration, the nearest-neighbor search is performed for a randomly chosen query point $\mathbf{q}'$ near the original query point $\mathbf{q}$. Hence, the Randomized traversal method facilitates the examination of different paths within the kd-tree, which may include the nearest grid node.

The procedure of the Randomized nearest-neighbor search is described in Algorithm 2. At first, a Single-Pass nearest-neighbor search is performed for the query point $\mathbf{q}$ to estimate the distance d to the current nearest grid node. The next query point $\mathbf{q}'$ is randomly chosen to lie on a sphere around $\mathbf{q}$ with radius d, which is the distance of the original query point to the current nearest grid node found so far (cp. Figure 3.5b). The Single-Pass nearest-neighbor search is then repeated for the new query point $\mathbf{q}'$. If the distance of the original query point $\mathbf{q}$ to the grid node $\mathbf{p}'$ found in this iteration is lower than the distance to the previously found grid node $\mathbf{p}$, the procedure chooses $\mathbf{p}'$ as nearest neighbor. This procedure is repeated several times for randomly chosen query points $\mathbf{q}'$ to find the nearest grid node of the original query point.

By repeating the Single-Pass nearest-neighbor search several times with randomly chosen query points, the problem that this method might fail to find the nearest grid node is somewhat relaxed. The binary decision, which sub-tree to traverse in the next step, can now lead to the traversal of a different part of the kd-tree, which may include the

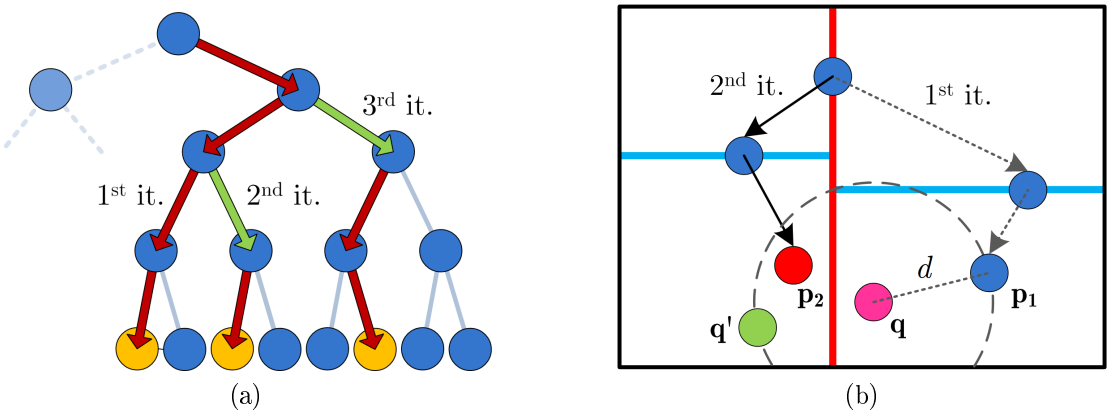


Figure 3.5: The Randomized search method traverses the tree several times with altering query points. This leverages the possibility that the nearest neighbor of the given query point is found, as also alternative paths are examined.

nearest grid node of $\mathbf{q}$. The more often the random restart is performed, the more increases the possibility, that the nearest grid node is found. For the implementation, this method inherits the simplicity of the Single-Pass method enhanced with a random restart approach. The runtime of this method is derived from the runtime of the Single-Pass method as $O(m \log n)$, where m is the number of random restarts.

Backtracking The third method to traverse the kd-tree is the Backtracking nearest-neighbor search ([Ben90]). This method is guaranteed to always find the nearest grid node of the given query point $\mathbf{q}$, as every sub-tree that may include the nearest neighbor is traversed using a backtracking mechanism (see Figure 3.6a). While the Backtracking method is usually recursively implemented, the algorithm presented in this thesis describes an iterative implementation of the method. This is due to the fact, that the CUDA GPU programming interfaces doesn't allow recursive function calls [NVI10a]. An iterative implementation of this traversal method on the other hand can be ported to the CUDA framework without much effort.

The iterative search procedure is depicted in Algorithm 3. It uses a stack St (a last-in first-out data structure) for the backtracking performance, as well as a queue Q (a first-in first-out data structure) which stores the indices of those sub-tree root nodes, that are alternatively traversed. The root node of the kd-tree with index 0 is initially stored in Q to start the procedure.

At first, the kd-tree is traversed top to bottom, similar to the procedure described in Algorithm 1, whereas the index of each visited node is stored in St. The distance of the query point $\mathbf{q}$ to the found grid node $\mathbf{p}'$ is stored as the current best distance d and p' is stored as the current nearest neighbor. Then, the backtracking step is performed using the stack St, to visit all tree nodes again in reversed order. In this phase, every

Algorithm 2 RANDOMIZED-NEAREST-NEIGHBOR-SEARCH($\mathbf{q}$)

Require: A kd-tree given as split values S, split dimensions D and point indices L.
Require: A query point $\mathbf{q}$.
Require: The number of iterations m.
Ensure: $\mathbf{p}$ is the nearest neighbor to $\mathbf{q}$.

```
p ← SINGLE-PASS-NEAREST-NEIGHBOR-SEARCH(q)
d ← ‖ p − q ‖
for i = 1 to m do
    q′ ← Random point on sphere around q with radius d
    p′ ← SINGLE-PASS-NEAREST-NEIGHBOR-SEARCH(q′)
    d′ ← ‖ p′ − q ‖
    if d′ < d then
        p ← p′
        d ← d′
    end if
end for
return p
```

split value S_i is checked for whether the nearest grid node may be found on the opposite half space of the current splitting plane. As depicted in Figure 3.6b, this is done by testing whether $q_{D_i} + d > S_i$ if $q_{D_i} \leq S_i$ (the left half space was chosen before), or else $q_{D_i} - d < S_i$, where q_{D_i} is the D_i'th component of $\mathbf{q}$. If one of the conditions is fulfilled, the index of the root node of the alternative sub-tree is stored in the queue Q and the backtracking process is continued. After all nodes were visited again, the tree is traversed once more from top to bottom starting at the next root node from the queue. Each alternatively found grid node is tested and the current best distance d and the current nearest neighbor p are updated accordingly. This process is repeated, until Q is empty, which indicates that no more alternative sub-trees need to be traversed and $\mathbf{p}$ is the nearest grid node to $\mathbf{q}$.

The backtracking step ensures, that all half spaces of the kd-tree are traversed which may include the nearest grid node to the query point $\mathbf{q}$. In every iteration, the current best distance is updated to minimize the number of alternative paths that need to be traversed. Overall, this method always returns the nearest grid node after the procedure finishes. Unfortunately, the runtime of the backtracking nearest-neighbor search is exponential in the worst case [LW77]. Also, even the described iterative implementation needs adjustments to the code when ported to the CUDA framework, as this framework does not yet provide built-in LIFO and FIFO data structures.

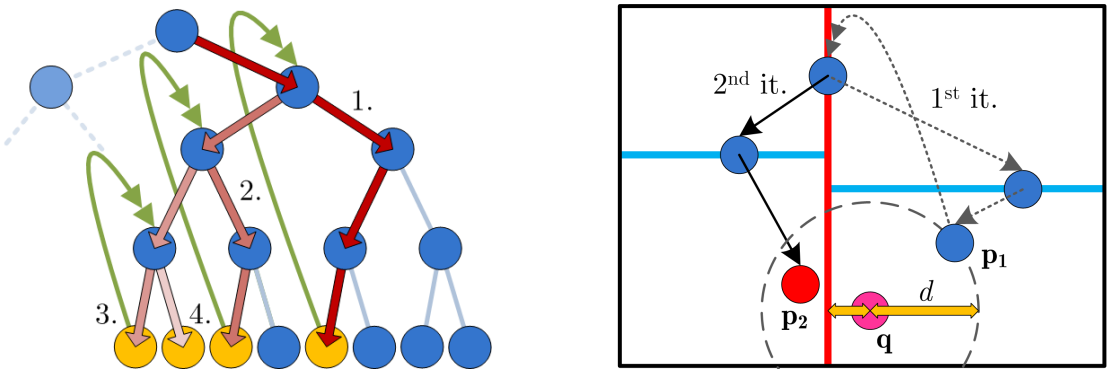


Figure 3.6: The Backtracking nearest-neighbor search performs several backtracking steps to traverse every sub-tree that may include the nearest neighbor of the given query point.

Algorithm 3 BACKTRACKING-NEAREST-NEIGHBOR-SEARCH($\mathbf{q}$)

Require: A kd-tree of N levels, split values S, split dimensions D and point indices L.
Require: A query point $\mathbf{q}$.
Require: A FIFO data structure Q and a LIFO data structure St.
Ensure: $\mathbf{p}$ is the nearest neighbor to $\mathbf{q}$.

$Q \leftarrow 0$
while Q is not empty **do**
  $i \leftarrow Q$
  while $i < 2^N$ **do**
    /* *Traverse tree top to bottom from node i and store each node index in St* */
  end while
  $\mathbf{p}' \leftarrow$ Point with index L_{i-2^N}
  $d' \leftarrow \| \mathbf{p}' - \mathbf{q} \|$
  if $d' < d$ **then**
    $d \leftarrow d', \mathbf{p} \leftarrow \mathbf{p}'$
  end if
  while St is not empty **do**
    $i \leftarrow St$
    if $q_{D_i} > S_i \wedge q_{D_i} - d < S_i$ **then**
      $Q \leftarrow 2i + 1$
    end if
    if $q_{D_i} \leq S_i \wedge q_{D_i} + d > S_i$ **then**
      $Q \leftarrow 2i + 2$
    end if
  end while
end while
return $\mathbf{p}$

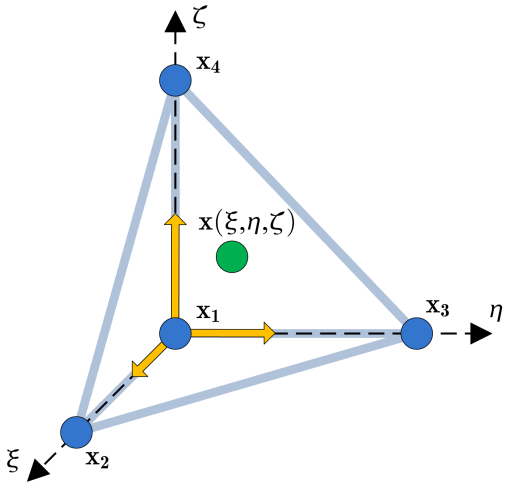


Figure 3.7: In the natural coordinate system of a tetrahedron cell, a position is described as linear combination of the four cell nodes

3.2.3 Tetrahedral Walk

Both, the nodes of the tetrahedral grid and the query positions for the cell search, are located in the physical coordinate system of the domain. To check, whether a query position lies inside a tetrahedron cell, the coordinates of the position needs to be transformed to the local coordinate system of that cell, called the natural coordinates. In natural coordinate space, testing whether a point lies inside a cell can simply be done by checking whether the calculated natural coordinates are valid. This approach can be extended to a point location scheme as proposed by [KL95].

In natural coordinate space, a position $\mathbf{x}$ is described as

$$\mathbf{x}(\xi,\eta,\zeta) = \mathbf{x_1} + (\mathbf{x_2} - \mathbf{x_1})\xi + (\mathbf{x_3} - \mathbf{x_1})\eta + (\mathbf{x_4} - \mathbf{x_1})\zeta \tag{3.1}$$

where $\mathbf{x_i} = (x_i, y_i, z_i)^T, i \in \{1, \ldots, 4\}$ are the positions of the nodes of a cell in physical coordinate space (cp. Figure 3.7). This function can be inverted, to calculate the natural coordinates of a query point $\mathbf{q} = (x_q, y_q, z_q)^T$ given in physical coordinates. Therefore, the above equation is rewritten as equation system

$$\begin{pmatrix} x_q - x_1 \\ y_q - y_1 \\ z_q - z_1 \end{pmatrix} = \begin{pmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{pmatrix} \cdot \begin{pmatrix} \xi \\ \eta \\ \zeta \end{pmatrix} \tag{3.2}$$

and solved by multiplying the inverse of the 3×3 matrix on the right to both sides, which yields:

$$\begin{pmatrix} \xi \\ \eta \\ \zeta \end{pmatrix} = \frac{1}{V} \cdot A \cdot \begin{pmatrix} x_q - x_1 \\ y_q - y_1 \\ z_q - z_1 \end{pmatrix} \tag{3.3}$$

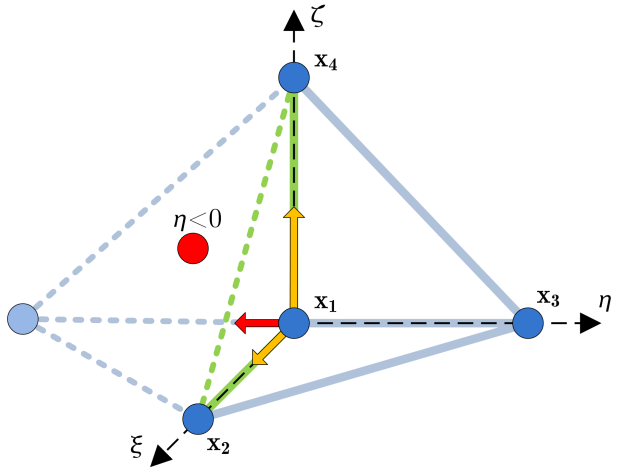


Figure 3.8: If one or more of the natural coordinates of a position are lower than 0, the position is outside of the cell.

where the rows of the (constant) matrix A are given by

$$\begin{aligned} A_{1,-} &= ((\mathbf{x_3} - \mathbf{x_4}) \times (\mathbf{x_4} - \mathbf{x_1}))^T \\ A_{2,-} &= ((\mathbf{x_1} - \mathbf{x_2}) \times (\mathbf{x_4} - \mathbf{x_1}))^T \\ A_{3,-} &= ((\mathbf{x_1} - \mathbf{x_2}) \times (\mathbf{x_2} - \mathbf{x_3}))^T \end{aligned} \tag{3.4}$$

and the determinant V of the matrix is given by:

$$V = (\mathbf{x_2} - \mathbf{x_1}) \cdot ((\mathbf{x_3} - \mathbf{x_1}) \times (\mathbf{x_4} - \mathbf{x_1})) \tag{3.5}$$

The calculation scheme described in Equation (3.3) can be used to directly calculate the natural coordinates of a query point $\mathbf{q}$ in physical coordinates for a given cell c. If c contains $\mathbf{q}$, the following four conditions are fulfilled:

$$\xi \geq 0,\ \eta \geq 0,\ \zeta \geq 0,\ 1 - \xi - \eta - \zeta \geq 0 \tag{3.6}$$

If one or more of the conditions are violated, the query point $\mathbf{q}$ lies outside of c and another cell needs to be tested (see Figure 3.8). Therefore, the natural coordinates also indicate the cell c_{next} that should be tested afterwards. If, for example, the condition $\xi \geq 0$ is violated, the cell c_{cont} that contains $\mathbf{q}$ may share a common face with c, the face where $\xi = 0$. Similarly, this holds, if $\eta \geq 0$ or $\zeta \geq 0$ are violated. If the last condition, $1 - \xi - \eta - \zeta \geq 0$ is violated, then c_{cont} may share the diagonal face with c, where $\xi + \eta + \zeta = 1$.

This consideration is used for the tetrahedral walk cell location scheme. In each iteration, the natural coordinates of $\mathbf{q}$ are calculated for the current cell. If one or more of the conditions given by Equation (3.6) are violated, the worst violator is determined and the cell, that shares the face defined by the worst violator is tested in the next iteration.

Even, if c_{cont} is not found in the next iteration, the worst violator still gives a good heuristic, which cell to test next. This procedure is iterated until all conditions of Equation (3.6) are fulfilled or a boundary cell is reached, in which case no further cells can be tested. In this case, a cell for the query position could not be located and the query is rejected. In the former case, the cell containing the query point is found and returned.

Once, the cell c_{cont} containing the query point $\mathbf{q}$ is found, the natural coordinates of $\mathbf{q}$ w.r.t. c_{cont} are used for the linear interpolation, i.e. the attribute field u at position $\mathbf{q}$ is given by

$$u(\mathbf{q}) = u_1 + (u_2 - u_1)\xi + (u_3 - u_1)\eta + (u_4 - u_1)\zeta \tag{3.7}$$

where $u_1 \dots u_4$ are the attributes defined at the grid nodes $\mathbf{x_1} \dots \mathbf{x_4}$.

This cell location scheme is only useful, if the number of cells that need to be tested can be limited. Otherwise, too many iterations must be performed to locate the right cell, especially if the number of grid cells is large. To avoid those long tetrahedral walks, the starting cell for the walk is estimated by calculating the nearest grid node of the query point $\mathbf{q}$ using one of the kd-tree based nearest-grid-node search methods described in Section 3.2.2. One of the cells associated with the nearest grid node is then used as starting cell for the tetrahedral walk, which is obtained by the node-to-cell lookup table described in the following.

3.2.4 Node-to-Cell Lookup Table

The kd-tree search structure is used to organize the grid nodes, as their number is considerably smaller compared to the number of cells. It is used in the global search phase to locate a grid node near the query point while the tetrahedral walk procedure is used to locate the surrounding cell in the local search phase. In order to perform the tetrahedral walk efficiently, the index of a cell in the neighborhood of the query positions is required. Therefore, a link between the nearest grid node and a cell that references this node is needed. This node-to-cell assignment is ambiguous, as grid nodes are usually indexed to belong to more than one cell.

The construction of the node-to-cell lookup table used in this thesis is rather simple. The table is represented by two lists: The first list N stores an offset for each grid node and is used to access the second list C which stores the indices of the cells referencing the grid node (cp. Figure 3.9).

Two passes over all cells are required to construct the lookup table. In the first pass, a temporary array is filled for each grid node with the number of cells referencing this

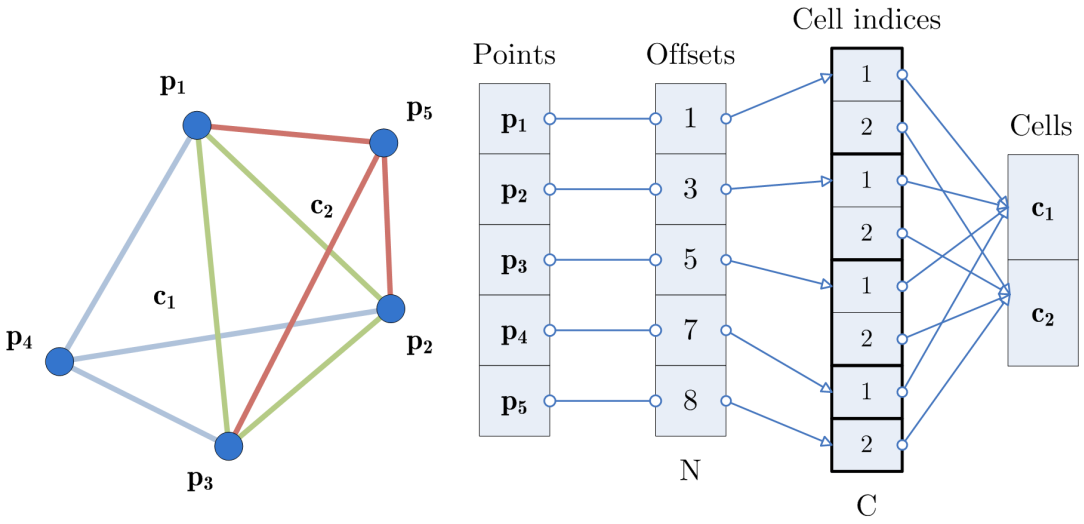


Figure 3.9: The table that is used to look-up the cell index for a grid node consists of two lists: one that stores the cell indices and another one that stores an offset for each grid node.

node. Based on this values, an offset is calculated for each node and stored in N. The sum of all cell counts is used as the length of the second array C. In the second pass, C is filled with the indices of the referencing cells. Hereby, N is used to access C while an additional temporary array stores the number of already inserted cell indices for each grid node.

The ambiguity of the node-to-cell assignment is resolved by choosing for each node the respective cell with the lowest cell index, which is referenced directly by the offset stored in N. One problem that arises from the ambiguity is that the chosen cell is not always the optimal one to start the tetrahedral walk at. In rare occasions, the tetrahedral walk needs to traverse several of the cells referencing the chosen grid node, until the surrounding cell is found. One way to handle this problem is the usage of a kd-tree which yields cell indices directly, e.g. by storing the centers of gravity of all cells in the tree structure. On the other hand, the amount of memory required to store this kd-tree would be much increased, as the number of cells generally exceeds the number of grid nodes by half an order of magnitude.

3.3 Search Evaluation

While the Single-Pass kd-tree traversal method does not always find the nearest grid node of a given query point, it has the advantage of simplicity and gives a fixed run duration determined by the number of tree levels. The Backtracking method always finds the

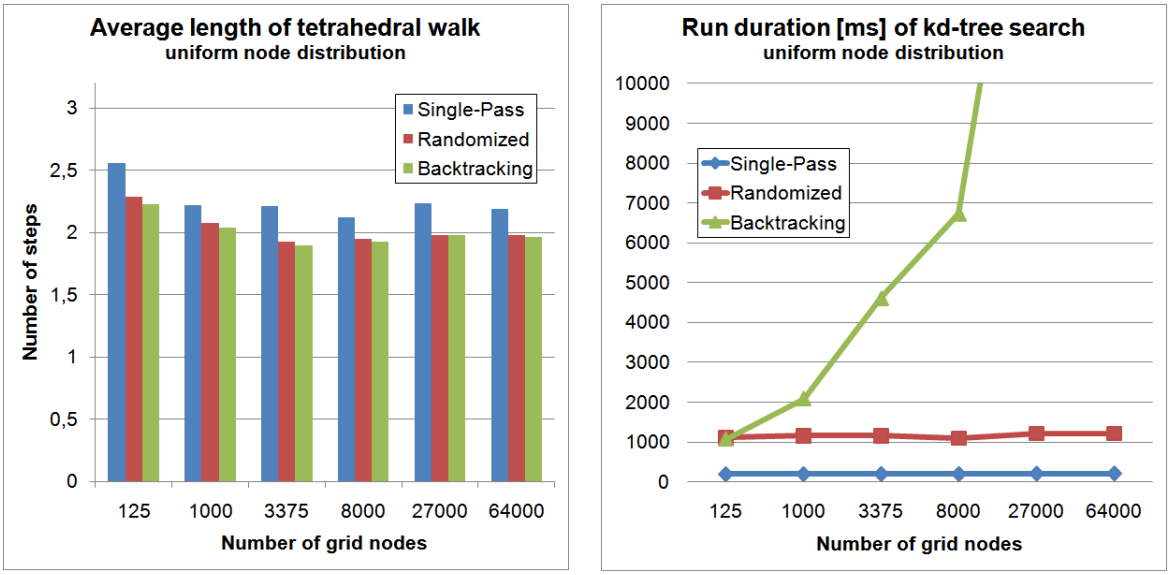


Figure 3.10: Average number of tetrahedral walk steps and run duration of the different kd-tree search methods to perform the point-location scheme on a tetrahedral grid with uniformly distributed grid nodes.

nearest grid node, but at a cost of eventually exponential runtime. The Randomized method inherits the simplicity and fixed runtime properties from the Single-Pass method but has a higher probability to find the nearest grid node of a given query point. Overall, there exists a tradeoff between accuracy, run duration and implementation effort.

For the evaluation of the presented kd-tree traversal methods, the entire point-location scheme described in Section 3.2 was performed for 100K random query points initially created inside the domain of a cube with an edge length of two. Several synthetic tetrahedral grids with a different number of grid nodes were created for this domain as described in Section 3.4, to analyze the impact of different grid structures on the runtime behavior and accuracy of the described traversal methods. The grids were generated by distributing the grid nodes on regular grids of various dimensions, which were then tetrahedralized to create the cell structure. Some randomness in the distribution of the grid nodes was also introduced to generate unstructured grids with varying cell sizes. For each grid, the kd-tree was constructed as described in Section 3.2.1. As metric for the accuracy of the different tree traversal methods, the average number of tetrahedral walk steps needed to locate the cell that contains the query point is used. The tetrahedral walk hereby starts at the cell obtained from the node-to-cell lookup table for the grid node which was found by each of the kd-tree traversal methods. This should describe the impact of the accurary of the different nearest-grid-node search methods on the entire point-location procedure quite well.

The results of the evaluation of the kd-tree traversal methods for uniformly distributed grid nodes are shown in Figure 3.10. For this setting, the average number of tetrahedral

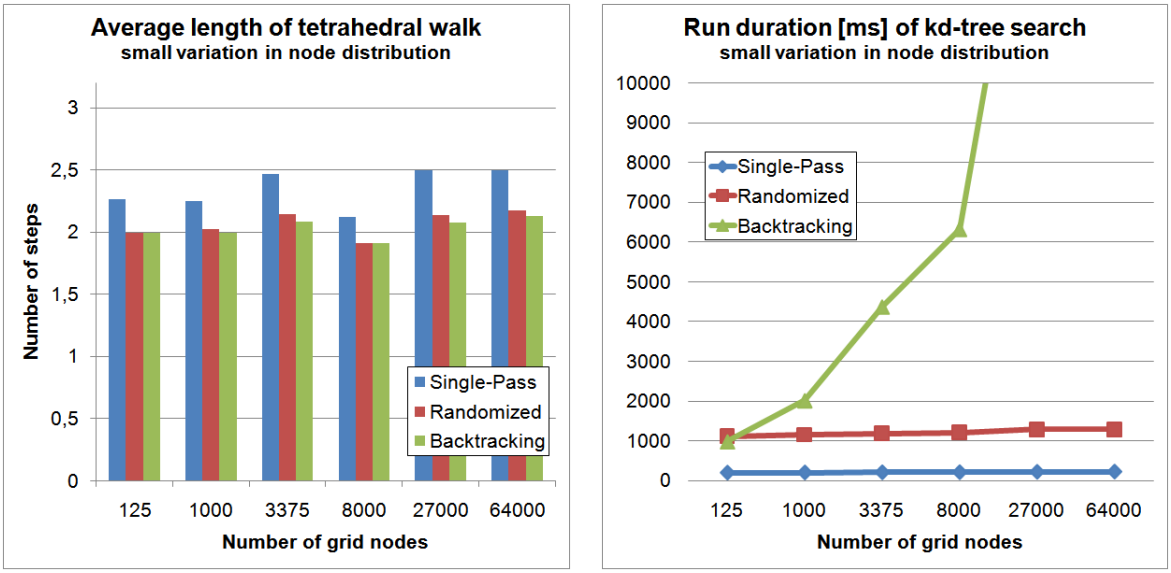


Figure 3.11: Number of tetrahedral walk steps and runtime behavior, when adding a small variation to the grid node distribution.

walk steps is almost the same for the different grid sizes when using the Randomized or the Backtracking kd-traversal methods, whereas the number of steps using the Single-Pass method is slightly higher. Regarding the whole run duration to locate the nearest grid nodes of all query points, the duration of the Single-Pass and the Randomized method on different grid sizes are almost constant, whereas the time needed to perform the search using the Backtracking method grows exponentially with an increased number of grid nodes. If a small variation in the distribution of the grid nodes is introduced, the average number of tetrahedral walk steps is still almost the same for the Randomized and the Backtracking method (cp. Figure 3.11). When using the Single-Pass method for the point location, on average half of a tetrahedral walk step needs be additionally performed compared to the other methods. The run duration of the search methods for this setting is similar to the previously described setting with no variation in the grid node distribution.

When the point-location scheme evaluation is performed on tetrahedral grids with a large variation in the grid node distribution, the average number of traversed cells is much more increased for larger grid node counts compared the other settings with little or no variation (cp. Figure 3.12). This is due to the large variation of the tetrahedron cell sizes as well as the ambiguity of the node-to-cell assignment. The run duration of the methods is comparable with the previous results, whereas the Backtracking method performs slightly better for this setting, as the kd-tree search structure is more efficient when indexing randomly distributed grid nodes. For this setting, the Backtracking procedure performs fewer backtracking steps, as lesser grid nodes are located directly on or near the splitting plane compared to a uniform grid node distribution.

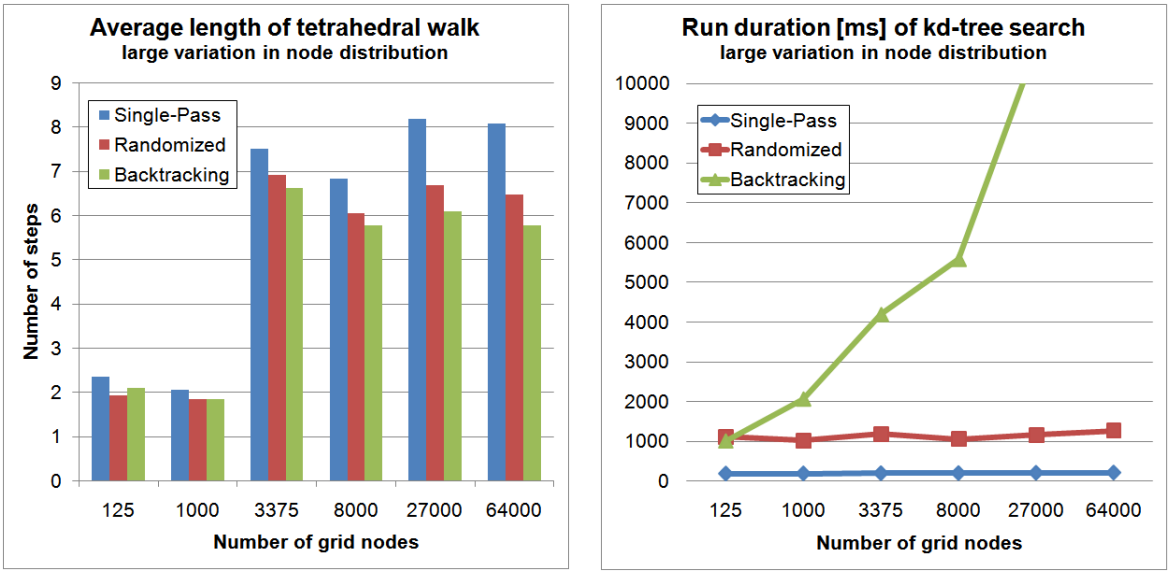
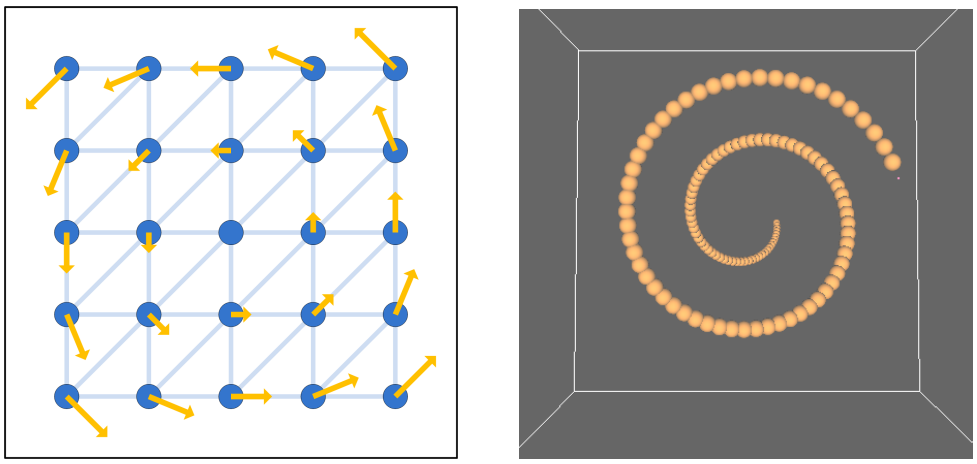


Figure 3.12: For a large variation in the grid node distribution, the average number of tetrahedral walk steps is increased. The runtime behavior of the Backtracking method is slightly better, as lesser backtracking steps are performed.

Overall, it can be concluded that the given accuracy of the evaluated kd-tree traversal methods doesn't have a great impact on the average number of tetrahedral walk steps which are required to locate the cell that contains the query point. Even when using a starting cell referencing the nearest grid node, as obtained by the Backtracking method, the average number of steps is not explicitly lesser, which is due to the ambiguity of the node-to-cell lookup table as discussed in Section 3.2.4. The exponential runtime behavior disqualifies the Backtracking method, as the improved accuracy using this method does not outweigh its computational cost. The Single-Pass method shows the best performance, while the low accuracy of this method has only little influence on the overall performance of the point-location scheme. A good tradeoff between accuracy, computational cost and implementation effort is achieved using the Randomized method. This procedure is almost as accurate as the Backtracking method while the measured run duration grows only linearly with the number of grid nodes.

The implementation of the Randomized algorithm using the CUDA framework is problematic, as the framework does not provide a build-in method to generate random numbers on the GPU. Substantiated by the evaluation results, the Single-Pass method was chosen for the CUDA-based implementation in order to traverse the kd-tree structure directly on the GPU. This method shows the best runtime performance while the algorithm is very straightforward to implement. As the evaluation has shown, the accuracy of the Single-Pass method is sufficient to find a grid node near the query position. A further advantage of the Single-Pass method is the reduced amount of required memory, as the resulting cell indices can be stored directly within the leaf nodes of the kd-tree, which dismisses the additional node-to-cell lookup table.



(a) A helical synthetic flow field embedded in a tetrahedral grid.

(b) Particles visualizing a synthetic inwards spiraling flow.

Figure 3.13: A procedure was developed for testing purposes which allows the generating of arbitrary tetrahedral grids and synthetic flow fields.

3.4 Synthetic Datasets

For the development and implementation of the algorithms, a method was developed, by which different tetrahedral grids and synthetic flow fields can be generated. This allows the testing of the algorithmic implementation within a predictable environment, as the synthetically generated data facilitates the comparison of the visual output with foreseeable results. The generated tetrahedral grids are arbitrary in the number and distribution of grid nodes, which allows to experiment with differently sized tetrahedral grids and different grid node distributions.

The synthetic dataset is generated by defining a set of points, either on a regular grid or randomly distributed, which represents the nodes of the tetrahedral grid. The TetGen library [WIA09] is utilized to calculate the tetrahedron cells for the point set, which generates a Delaunay tetrahedralization based on the given point distribution. The library also generates the index array of cell neighbors while the tetrahedralization. Besides the tetrahedral grid, the generated synthetic dataset also includes a synthetic flow field, which is generated by evaluating the flow defined by

$$\vec{F}(x,y,z) = \begin{pmatrix} ax - by \\ bx + ay \\ -2az + c \end{pmatrix}$$

for each node position within the grid [KM92]. This creates a helical flow for $a = 0$ and $b, c > 0$, where b defines the intensity of the circular velocity and c defines the velocity in z-direction (cp. Figure 3.13a). By choosing $a > 0$, the resulting flow field equals an inward spiraling flow, with ever decreasing radius in z-direction (cp. Figure 3.13b).

CHAPTER 4

INTERACTIVE PARTICLE TRACING

In this thesis, interactive particle tracing is used to visualize the velocities of a simulated flow field. The flow field velocity data is given by several unstructured tetrahedral grids whereas each grid represents the state of the flow field for one point time. The particles are advected through the time-variant flow field by continuously updating the particles' positions according to the underlying flow which depicts the behavior of the simulated fluid indirectly by how it interacts with debris. Inspired by real flow experiments, the massless particles inside the simulated flow behave like smoke particles in a real turbulent environment, or like ink injected to water.

In an interactive visualization setting, a vast amount of particles is needed in order to convey an adequate impression of the flow field, similar to smoke, which consists of thousands of tiny particles. This leverages the computational cost needed to compute the advection of the particles, as the trajectory of each particle is calculated individually. For an interactive setting, it is absolutely necessary to perform the positions' update of the whole particle population within a short time limit. Fortunately, the position update of each particle for one advection step can be computed independently from the others as the movement depends solely on the velocity of the underlying flow field, which allows calculating the advection of each particle by an individual computational thread. This type of computing is supported by today's graphics processing units which can compute hundreds or thousands of those threads in parallel.

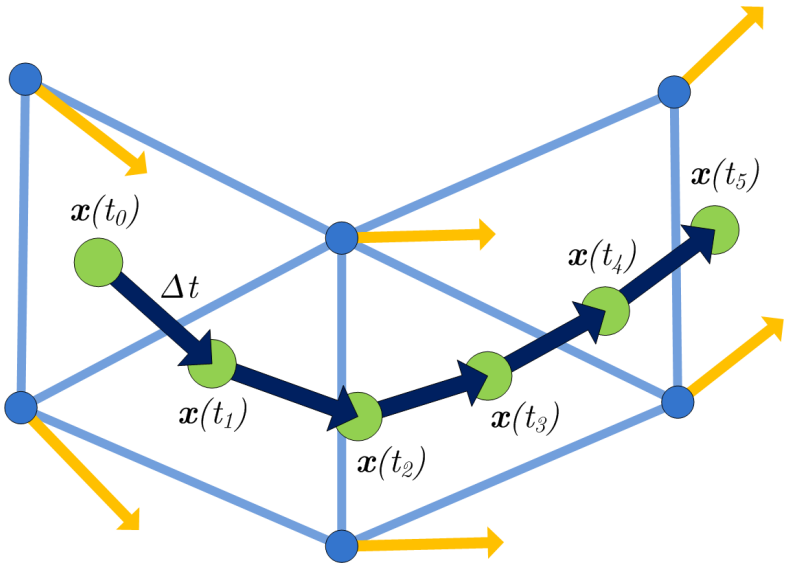


Figure 4.1: The trajectory of a particle is calculated by integrating the velocities of the time-variant flow field which are defined at the nodes of the tetrahedral grid.

4.1 Particle Advection

Particle advection is the process of updating the particles' positions. By repeating this step continuously, an animation of the particles' movement is obtained which conveys a visual impression of the underlying flow (cp. Figure 4.1). In order to calculate the advection of the particles, the flow velocity vector at the current position of each particle must be known. Therefore, the flow field velocities defined at the nodes of the unstructured tetrahedral grid are linearly interpolated by performing the point-location scheme described in Section 3.2 in order to locate the cell which currently surrounds the respective particle. The natural coordinates calculated in this step are then used to interpolate of the flow velocity at the particle's position, as described in Equation (3.7).

The trajectory of a particle through the flow field is given by the ordinary differential equation (ODE)

$$\frac{\partial \mathbf{x}}{\partial t} = \mathbf{v}(\mathbf{x}(t), t) \tag{4.1}$$

in which the initial condition $\mathbf{x}(t_0) = \mathbf{x}_0$ is defined by the seeding position of the particle at time t_0. Integrating both sides of the equation and reformulating yields

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \int_t^{t+\Delta t} \mathbf{v}(\mathbf{x}(s), s) ds. \tag{4.2}$$

This equation allows to calculate the new position of a particle which is currently located at position $\mathbf{x}(t)$ after it has moved through the flow field with a step size of Δt,

by integrating all intermediate flow field velocities between the current and the next position. For the advection calculation, this integral term is approximated by using one of the numerical integration methods described in the following.

4.2 Flow Field Integration

In order to update particles' positions using Equation (4.2), the flow field velocities need to be integrated. The numerical integration schemes which are regarded in this thesis are the first-order Euler integration and the third- and fourth-order Runge-Kutta integration. In addition, the embedded integration scheme proposed by Dormand and Prince [DP80] is used to integrate the flow field velocities which bases on the Runge-Kutta integration and calculates a fourth- and fifth-order accurate solution in one step. This embedded integration scheme can be utilized to adjust the step size adaptively, as described in Section 4.3.

The simplest and fastest method to calculate the particles' advection is the Euler integration scheme [PTVF07], which gives a first-order integration of the flow field velocities. Using the Euler integration scheme, the new position of a particle is calculated by evaluating the flow field velocity $\mathbf{v}(\mathbf{x_t}, t)$ at the particle's current position $\mathbf{x_t}$ at time t. For the particle's advection, this velocity is multiplied by the current step size Δt and added to the old position:

$$\mathbf{x_{t+1}} = \mathbf{x_t} + \Delta t \cdot \mathbf{v}(\mathbf{x_t}, t) \tag{4.3}$$

The velocity $\mathbf{v}(\mathbf{x_t}, t)$ is calculated by interpolating the flow field velocities as described in Equation (3.7). The temporal factor t is taken into account by linearly interpolating between the velocities of the current flow field state T_i and the next state T_{i+1}:

$$\mathbf{v}(\mathbf{x_t}, t) = a(t) \cdot \mathbf{v_{T_i}}(\mathbf{x_t}) + (1 - a(t)) \cdot \mathbf{v_{T_{i+1}}}(\mathbf{x_t}) \tag{4.4}$$

where $a(t) \in (0, 1]$ is the normalized point in time between t_{T_i} and $t_{T_{i+1}}$ calculated as

$$a(t) = \frac{t - t_{T_i}}{t_{T_{i+1}} - t_{T_i}}.$$

By using the Euler integration scheme, the integral factor of Equation (4.2) is approximated only by the velocity at the current particle's position $\mathbf{x_t}$ in order to calculate the new position $\mathbf{x_{t+1}}$ of the particle. This approximation leads to a small error in each advection step as the continuously distributed velocities in between the current and the next position of the particle are not considered. As more steps are performed, this error accumulates, which results in an inaccurate movement of the particle. The impact of this error on the visualization of computational fluid dynamics was examined by Buning in [Bun88].

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_s	a_{s1}	a_{s2}	$\cdots$	a_{ss-1}	
	b_1	b_2	$\cdots$	b_{s-1}	b_s
	$(b'_1$	b'_2	$\cdots$	b'_{s-1}	$b'_s)$

(a) General calculation scheme. The coefficients $b'_1 \dots b'_s$ describe an embedded integration scheme.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

(b) Fourth-order Runge-Kutta integration.

0			
$\frac{1}{2}$	$\frac{1}{2}$		
1	-1	2	
	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{1}{6}$

(c) Third-order Runge-Kutta integration.

Table 4.1: Runge-Kutta integration scheme third- and fourth-order, described as Butcher tableau.

The error of the Euler integration scheme can be reduced by considering the flow velocities in between the current and the next position of the particles. This is the case when using a higher-order integration scheme which calculates the next position of a particle by evaluating the flow velocities at more than one position. The most familiar method to calculate a higher-order integration is the Runge-Kutta method [PTVF07]. For an explicit Runge-Kutta method with s stages, the next position of a particle at position $\mathbf{x_t}$ is calculated by

$$\mathbf{x_{t+1}} = \mathbf{x_t} + \Delta t \cdot \sum_{j=1}^{s} b_j \cdot \mathbf{k_j} \tag{4.5}$$

where the coefficients $b_j \in \mathbb{R}$ are weights for the intermediate results $\mathbf{k_j} \in \mathbb{R}^3$ which represent the additional flow field velocities in between the current and the next position of the particle. The more stages are calculated, the more accurate the flow field integration becomes as more flow evaluations are performed. But this is only applicable for up to fifth-order accurate integration schemes, as the number of required stages rises faster than the order of the integration scheme for $s \geq 5$ [But87]. The additional flow field velocity vectors are calculated as

$$\mathbf{k_j} = \mathbf{v}(\mathbf{x_t^{(j)}}, t + c_j \cdot \Delta t) \tag{4.6}$$

in which the coefficients c_j describe how much the time t advances in each stage while the temporal velocity interpolation is handled as described in Equation (4.4). The intermediate positions $\mathbf{x_t^{(j)}}$, at which the additional flow field evaluations are performed, are calculated in dependency on the former results by

$$\mathbf{x_t^{(j)}} = \mathbf{x_t} + \Delta t \cdot \sum_{l=1}^{s} a_{jl} \cdot k_l \tag{4.7}$$

whereas the previously calculated flow velocities k_l are weighted by the coefficients a_{jl}.

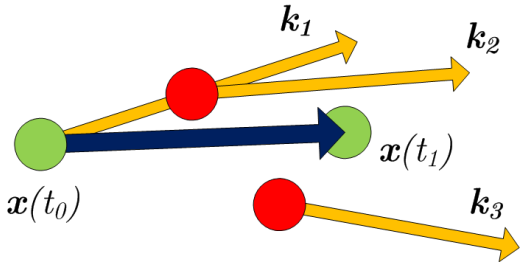


Figure 4.2: The third-order Runge-Kutta integration scheme evaluates the flow field velocity at three different positions.

As an example, the following calculations need to be performed in order to calculate the new position of a particle using the third-order Runge-Kutta integration method:

$$
\begin{aligned}
\mathbf{x_{t+1}} &= \mathbf{x_t} + \Delta t \cdot \left(\frac{1}{6}\,\mathbf{k_1} + \frac{4}{6}\,\mathbf{k_2} + \frac{1}{6}\,\mathbf{k_3} \right) \\
\mathbf{k_1} &= \mathbf{v}\left(\mathbf{x_t}, t \right) \\
\mathbf{k_2} &= \mathbf{v}\left(\mathbf{x_t} + \frac{\Delta t}{2}\,\mathbf{k_1}, t + \frac{1}{2}\,\Delta t \right) \\
\mathbf{k_3} &= \mathbf{v}\left(\mathbf{x_t} - \Delta t\,\mathbf{k_1} + 2\,\Delta t\,\mathbf{k_2}, t + \Delta t \right)
\end{aligned}
$$

The different methods of the Runge-Kutta family are distinguished by their number of stages s, as well as by their coefficients b_j, c_j and a_{jl}. The methods are usually depicted using the butcher tableau [But87], where the coefficients are arranged as shown in Table 4.1a. Table 4.1 also shows the coefficients for the third- and fourth-order Runge-Kutta integration schemes. Embedded Runge-Kutta methods like the Dopri-5 integration scheme are depicted by an additional row of b'_j coefficients.

The integration of the velocities using a higher-order integration scheme introduces several flow field evaluation steps. The third-order Runge-Kutta integration scheme, for example, evaluates the flow field velocity at three different positions in order to determine the new position of a particle (cp. Figure 4.2). This leads to six flow field evaluations per particle in every advection step, as the velocity is interpolated between two consecutive flow field states, according to Equation (4.4).

The flow field velocity evaluation is performed by the local point-location scheme described in Section 3.2.3. For every particle, the index of the cell, in which it was lastly located, is stored, and used as input for the flow field evaluation in the next stage of the integration. Additional tetrahedral walk steps may be required, if the natural coordinates of the position to evaluate w.r.t. the current cell are not valid, according to Equation (3.6). In this case, the natural coordinates of the adjacent cell given by

the worst violator, are calculated and tested. The tetrahedral walk typically requires at most one additional step, as the positions to evaluate are quite near to the current particle's position, except for large step sizes where several cells need to be traversed for each evaluation. Nevertheless, the evaluation of the flow field velocities at different positions is the most time-consuming process during the integration calculation even when executed in parallel on the graphics processing unit.

4.3 Adaptive Step Size Adjustment

As unstructured grids are regarded, it is problematic to advect the particles using a fixed value for the integration step size Δt. Problems arise due to the different sizes of the cells that are traversed during the advection process. If the integration step size is chosen too high, a particle might cross several cell borders at once and omits the flow information of those cells. This is especially a problem in parts of the domain, where particular small cell sizes are used to simulate turbulent areas of the flow. In such areas, the velocities differ highly from cell to cell and the step size needs to be adjusted according to the current situation to achieve an accurate visualization of the flow behavior. On the other hand, it is feasible to choose a larger step size for some areas of the grid without leaving out too much velocity information. An adaptive step size adjustment procedure chooses an independent step sizes for each particle depending on the currently known information. Facts that are used in order to choose an appropriate step size may include the flow field velocity around the position of the particle, the particle's trajectory or the structure of the grid, especially the size of the particle's surrounding cell.

In order to achieve a plausible visualization of the particle advection, all particles must be advected with the same step size Δt. By adaptively adjusting the step size for each particle, the number of advection steps per particle increases. Thus, instead of performing one advection step with step size Δt, n advection steps need to be performed a step size of $\Delta t_i,\ i \in \{1 \dots n\}$ each, so that $\Delta t_1 + \Delta t_2 + \ldots + \Delta t_n = \Delta t$.

4.3.1 Step-Doubling

The step-doubling procedure described in [PTVF07] is the simplest method to adaptively adjust the integration step size. This procedure chooses an appropriate step size by estimating an error value for the integration which considers the flow field velocity.

As depicted in Figure 4.3, the error value for the current step size Δt is determined as the distance between the position $\mathbf{x_1}(t + \Delta t)$ of a particle after one advection step and

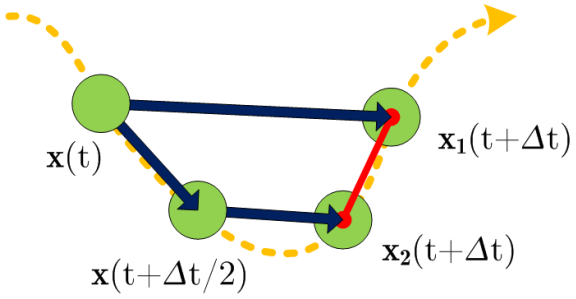


Figure 4.3: Adaptive step size adjustment using the step doubling procedure. The red bar indicates the error value of the current step size when compared with two half-steps.

the position $\mathbf{x_2}(t + \Delta t)$ after two advection steps with a step size of $\Delta t/2$ each:

$$\epsilon = \parallel \mathbf{x_1}(t + \Delta t) - \mathbf{x_2}(t + \Delta t) \parallel \tag{4.8}$$

Both new position updates are calculated using one of the integration schemes described above, e.g. the fourth-order Runge-Kutta integration. The integration step size is adjusted dynamically by defining an upper threshold for the error value ϵ. If the error value exceeds this threshold, the size of the step is halved and the flow field integration is repeated with the reduced step size. The procedure is iterated until the threshold condition is again fulfilled. Likewise, a lower threshold is defined, by which the step size is doubled. The whole step-doubling procedure is iterated until a complete advection step of step size Δt has been performed.

An existing integration procedure can easily be enhanced using the step-doubling procedure as it solely requires three approximations of the flow field integral in order to estimate a suitable step size for the integration. Still, the procedure requires nearly twice the number of computationally expensive integration calculations in order to estimate the step size. By using the fourth-order Runge-Kutta integration scheme for example, 11 flow field evaluations are required per iteration in both current tetrahedral grids. This number even increases, if the error value for an already reduced step size is still higher than the threshold. The step-doubling procedure for particle tracing on tetrahedral grids was evaluated by Kenwright in [KL95]. His results show that the performance using this step size adjustment procedure is worse than using a fixed step size of 5 steps per cell. This dismisses the step-doubling procedure for the adaptive step size adjustment on tetrahedral grids.

4.3.2 Curvature-Based Step Size Adaption

Kenwright also presents an alternative procedure for the adaptive adjustment of the step size called curvature-based step size adaption. This procedure estimates the integration step size for a particle based on the particle's moving direction through the flow field in

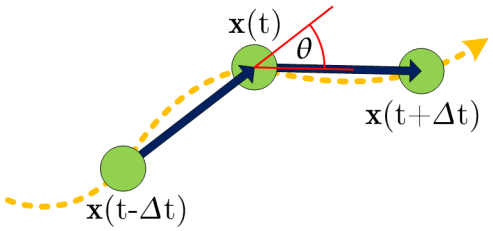


Figure 4.4: The curvature-based procedure estimates the step size based on the angle between the directions of the current and the last advection step.

the previous and the current advection step. As depicted in Figure 4.4, the curvature-based procedure calculates the path line curvature as the angle of the directions of two successive advection steps:

$$\cos\theta = \frac{(\mathbf{x_{i-1}} - \mathbf{x_i}) \cdot (\mathbf{x_i} - \mathbf{x_{i+1}})}{|\mathbf{x_{i-1}} - \mathbf{x_i}| \cdot |\mathbf{x_i} - \mathbf{x_{i+1}}|} \tag{4.9}$$

If the calculated angle exceeds a certain threshold, the current step size is halved and the last flow field integration step is repeated with the reduced step size. With this approach for the adaptive step size adjustment, Kenwright reported almost equal results compared to the step-doubling procedure when the angle is kept in between 3° for the lower and 15° for the upper threshold.

As only the particles' positions of the current and the previous advection steps are required for the step size estimation, which are generated in any case while the positions' update, no additionally gathered information is required for this adaptive step size adjustment procedure. Hence, the number of redundant calculations is significantly reduced compared to the previously described step-doubling procedure. In regions where

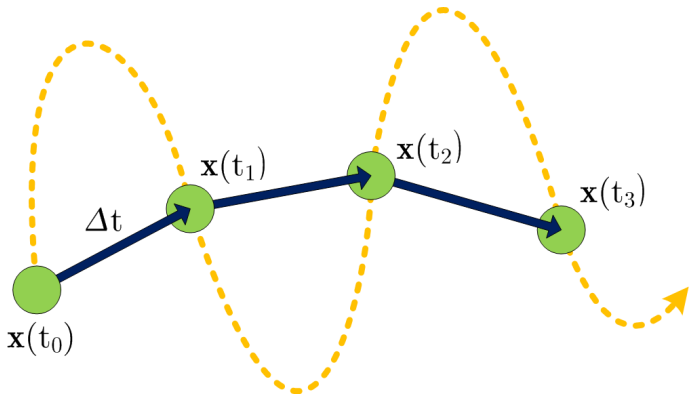


Figure 4.5: The curvature-based step size adjustment procedure is not accurate for rapidly changing flow directions, as the step size estimation considers the flow field velocities only indirectly.

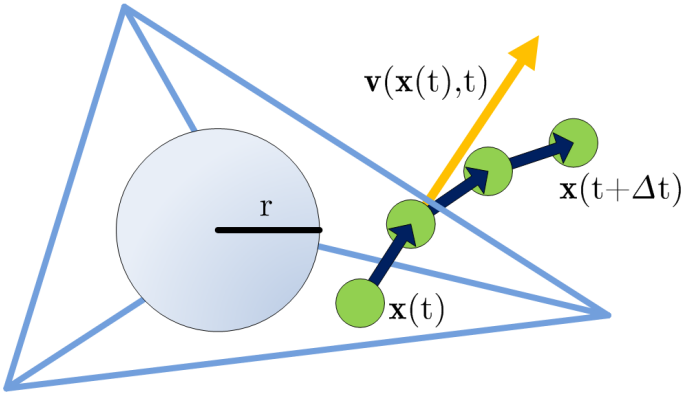


Figure 4.6: Using the in-sphere procedure, the particle advection is performed for n uniform sub-steps with an equidistant step size which is estimated by the in-sphere radius of the surrounding cell.

the flow is turbulent, the particle advection step size is reduced until the curvature of the particle's trail fits the boundary condition, which results in an adequate depiction of the flow in those regions.

But this method also has a severe drawback: Since the step size of the next advection step depends solely on the angle of the particles' directions of movement, the step size estimation is decoupled from the flow field velocities which affect the particle's trail only indirectly. Also, there is no indication about the error that is introduced by the current integration step size. In regions with rapidly changing flow directions, the step size may not be reduced, because the particle trail's curvature still fits the boundary condition for the chosen step size. As an example for such a situation, Figure 4.5 shows the advection of a particle through a turbulent flow of constant velocity but rapidly changing direction, which is depicted by the integral curve of the flow field. In this example, the advection of the particle using the step size estimated by the curvature-based approach does not convey an accurate visual impression of the underlying flow.

4.3.3 In-Sphere Step Subdivision

Another approach for the adaptive step size adjustment within tetrahedral grids, the in-sphere procedure, is suggested by Schirski in [Sch08]. He describes a procedure which relies solely on the topology of the tetrahedral grid in order to divide the current advection step of size Δt into n uniform sub-steps with an equidistant step size of $t_i = \Delta t/n$ each. As depicted in Figure 4.6, the number n of uniform sub-steps is obtained from the length of the flow velocity vector at the particle's position multiplied by the overall step size and divided by the insphere radius r of the current surrounding cell of the particle

within the tetrahedral grid:

$$n = \left\lceil \frac{\|\mathbf{v}(\mathbf{x_t}, t)\| \, \Delta t}{r} \right\rceil \tag{4.10}$$

This step size adjustment procedure is motivated by the fact that those regions within the flow field domain, for which a turbulent flow is expected during the simulation phase, are discretized by a dense grid node distribution with small cell sizes, which results in a higher resolution of the respective regions within the tetrahedral grid. Using the insphere radius of the surrounding cell of a particle ensures that the step size corresponds to the actual topology of the grid at the particle's position. Regions with small cell sizes are traversed with a smaller step size by which more advection sub-steps are performed for a particle crossing this area. If the cells are equally shaped, the velocity is evaluated in every cell that is traversed. The computational effort to determine the step size is quite small, since the insphere radius calculation is simple and the flow velocity $\mathbf{v}(\mathbf{x_t}, t)$ at the current particle's position is evaluated regardless of which method is used for the integration calculation.

As the factor for the uniform step subdivision depends solely on the insphere radius of that cell which contains the position of the particle at the beginning of the sub-step advection, an inaccurate particle advection may arise for regions of the tetrahedral grid with highly varying cell sizes, since too few sub-steps may be performed if the particle resides in a large cell at the beginning while the subsequent cells have a much smaller size. Also, despite the length of the velocity vector at the current particle's position, the flow field velocities are not considered for the estimation of the number of sub-steps. The performance of a number of uniform sub-steps using an equidistant step size for the flow field integration leads to the same problem that was already discussed for the curvature-based approach, as the chosen step size does not depend on how the time-variant flow evolves while the advection sub-steps are performed. Especially in regions with rapidly changing flow directions, the chosen step size may lead to a non-accurate movement of the particle w.r.t. the underlying flow field (cp. Figure 4.5).

4.3.4 Dopri-5 Adaptive Step Size Adjustment

The procedures for the adaptive step size estimation presented so far have either the drawback of a much higher computational effort, like the step-doubling procedure or do not consider the flow field velocities, like the curvature-based and the in-sphere procedures. A reasonable adaptive step size adjustment procedure should be able to adapt the step size to the flow field velocities without introducing too much computational effort.

In this thesis, the procedure of Dormand and Prince [DP80] called Dopri-5 adaptive step size adjustment is used. This procedure utilizes an embedded Runge-Kutta integration

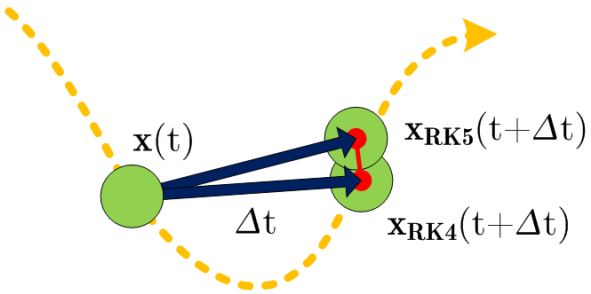


Figure 4.7: The embedded Dopri-5 integration scheme yields a fourth- and a fifth-order accurate solution of the flow field integration in a single step. Both solutions together are used to estimate the error of the fourth-order approximation of the flow field integral.

scheme which performs seven integration stages in order to calculate a fourth- and a fifth-order accurate approximation of the flow field integral in a single step. The approach is similar to the method of Fehlberg [Feh70] which minimizes the error of the fourth-order solution, whereas the coefficients of the embedded Dopri-5 integration scheme are chosen to minimize the error of the fifth-order solution. By using the Dopri-5 integration scheme, the fifth-order solution is used to approximate the integral term of Equation (4.2) in order to calculate the particle advection while the fourth-order solution is used to calculate an error value for the current step size by which the current step size is adaptively adjusted.

The integration is performed as described in Section 4.2 by using the coefficients presented as Butcher tableau in Table 4.2, whereas seven flow field velocity evaluations are performed in both current temporal domain states. The intermediate results of the flow field integration (cp. Equation (4.6)) are hereby weighted with either the b or the b' coefficients in order to calculate the fourth- and fifth-order accurate solutions in a single integration step.

To adaptively adjust the step size of the current integration, an error value is derived from both solutions of the last flow field integration step. This error value describes the error of the fourth-order accurate solution and is given by the difference between the fourth-order solution $\mathbf{x}_{\mathrm{RK4}}(t+\Delta t)$ and the fifth-order solution $\mathbf{x}_{\mathrm{RK5}}(t+\Delta t)$ (cp. Figure 4.7):

$$\epsilon = \parallel \mathbf{x}_{\mathrm{RK5}}(t+\Delta t) - \mathbf{x}_{\mathrm{RK4}}(t+\Delta t) \parallel \tag{4.11}$$

An adaptive step size adjustment procedure which utilizes the embedded Dopri-5 integration scheme is presented in Algorithm 4. This procedure is generally applicable to embedded Runge-Kutta integration schemes which calculate solutions of order p and $p+1$. For the embedded integration scheme of Dormand and Prince, the solution of fifth-order is essentially more exact than the fourth-order solution, so that this solution is used for the particle advection process.

0								
$\frac{1}{5}$	$\frac{1}{5}$							
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$						
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$					
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$				
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$			
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$		
	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$	(fourth-order)
	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	(fifth-order)

Table 4.2: Coefficients of the Dopri-5 method listed as Butcher tableau. The upper row of b coefficients gives the fourth- and the lower row the fifth-order accurate solution.

As input to the procedure, the user defines an error tolerance factor τ_0, a minimal step size $h_{\min}$, a safety factor $\rho \in (0, 1]$ and an increase bound $\eta \geq 1$. In every iteration, the procedure tries to adjust the current step size h', so that the error value from Equation (4.11), divided by the current step size, lies below the given error tolerance τ_0. Otherwise, the step size is halved and another iteration is performed. If the reduction has reached the minimal step size, $h' \leq h_{\min}$, the current step size is accepted despite the error value in order to prevent a dead-lock situation when the error threshold condition cannot be fulfilled by further reducing the step size. If the current step size h' is accepted, the fifth-order accurate solution $\mathbf{x}_{\mathrm{RK5}}(t + h')$ of the flow field integration defines the new position of the particle which is used as the input position in the next iteration.

As proposed in [Mel09], the integration step size to calculate solutions of order p and $p + 1$ in the next iteration is estimated by

$$h' = \rho \left(\frac{\tau_0}{\epsilon} h'^{\,p+1} \right)^{1/p}$$

based on the current step size h', the last error value ϵ and the tolerance factor τ_0. In order to minimize the number of repetitions, a safety factor $\rho \in (0, 1]$ is multiplied which raises the possibility that the new step size proposal is immediately accepted in the next iteration. In the Dopri-5 adaptive step size adjustment algorithm, the equation for the new step size estimation is enhanced, to ensure that the new value for h' does not lie below the minimal step size $h_{\min}$. Also, if the step size increases, the growth rate is restricted by an increase bound $\eta \geq 1$. A typical value would be $\eta = 2$, by which the new step size is restricted to be at most twice the size of the previous step.

To ensure the uniform movement of all particles, the procedure requires the input of the global step size h as well as the current simulation time t by which a maximum time T

Algorithm 4 DOPRI-5 ADAPTIVE STEP SIZE ADJUSTMENT

Require: An error tolerance factor τ_0.
Require: The minimal step size $h_{\min}$.
Require: A safety factor $\rho \in (0, 1]$.
Require: An increase bound $\eta \geq 1$.
Require: The initial step size h and the simulation time t.

$T \leftarrow t + h$
$h' \leftarrow h$
while $(t < T)$ **do**
  calculate $\mathbf{x}_{\mathrm{RK4}}(t + h'), \mathbf{x}_{\mathrm{RK5}}(t + h')$ **using the Dopri-5 integration scheme**
  $\epsilon \leftarrow \| \mathbf{x}_{\mathrm{RK5}}(t + h') - \mathbf{x}_{\mathrm{RK4}}(t + h') \|$
  if $((\epsilon/h') \leq \tau_0 \vee h' \leq h_{\min})$ **then**
    $\mathbf{x}(t + h') \leftarrow \mathbf{x}_{\mathrm{RK5}}(t + h')$
    $t \leftarrow t + h'$
    $h' \leftarrow \max\left\{h_{\min}, \min\left\{\eta h', \rho\left(\frac{\tau_0}{\epsilon} h'^{5}\right)^{0.25}\right\}\right\}$
    if $t + h' \geq T$ **then**
      $h' \leftarrow T - t$
    end if
  else
    $h' \leftarrow h'/2$
  end if
end while

is calculated. In every iteration, the time t is advanced by the current step size h'. If the step size of the next iteration exceeds the maximum time T, the step size is defined by $h' = T - t$, by which a small additional advection step is performed in order to complete the global advection step of step size h.

The Dopri-5 integration scheme can easily be implemented as a modification of the classic Runge-Kutta methods using the coefficients listed in Table 4.2, while the step size adjustment procedure using Algorithm 4 requires little more implementation effort.

4.3.5 Discussion

The presented adaptive step size adjustment procedure using the embedded Dopri-5 integration scheme uses an approach for the step size estimation which is similar to the step-doubling procedure, as an error value for the current integration step size is estimated, by which the size of the next step is adjusted dynamically. If the error value exceeds a certain threshold, the last iteration step is repeated with a reduced step size, until it fits the given conditions. While the step-doubling procedure calculates the

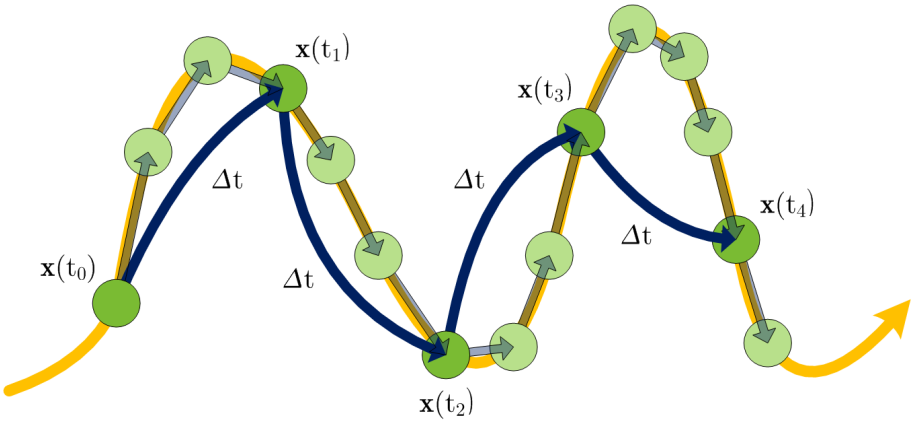


Figure 4.8: By using the Dopri-5 adaptive step size adjustment procedure, the integration step size is continuously adjusted in order to achieve a highly accurate depiction of the underlying flow.

error value of the current integration step as the difference between a single advection step and two half-steps, the Dopri-5 procedure performs an embedded Runge-Kutta integration which yields the error value as the difference between the fourth- and the fifth-order accurate solutions. Compared to the step-doubling procedure, which requires 11 flow field evaluations in each integration step to calculate the error value and a fourth-order accurate approximation of the flow field integral using the fourth-order Runge-Kutta integration, the Dopri-5 step size adjustment procedure requires only 7 flow field evaluations to calculate a fifth-order accurate solution as well as an estimation of the error of the fourth-order accurate solution. Therefore, the Dopri-5 procedure saves 8 computationally expensive flow field evaluations in total, as 14 flow evaluations are sufficient for the embedded Dopri-5 integration scheme to integrate the time-variant flow field in contrast to the step-doubling procedure with fourth-order Runge-Kutta integration which evaluates the flow field velocity at 22 distinct positions.

As depicted in Figure 4.8, the described step size adjustment procedure using the Dopri-5 integration scheme carefully respects the flow field velocities for the estimation of an appropriate integration step size. The Figure presents the previously described turbulent flow of constant velocity but rapidly changing direction which is depicted by the integral curve of the flow field. A continued adjustment of the current integration step size is required to calculate the particles' movement in order to obtain a highly accurate visual depiction of the underlying flow. Using the Dopri-5 integration scheme with adaptive step size adjustment, each particle advection step is performed with an integration step size that complies with the given error tolerance of the fourth-order accurate solution. Therefore, each advection sub-step follows the integral curve of the flow field and the depiction of the particle's position after several sub-steps leads to an accurate visual impression of the underlying flow. As already discussed, such accuracy is hardly achievable by adjusting the step size according the curvature of the particle's trail or by using a

uniform subdivision of the advection step, without decreasing the global step size.

By using the Dopri-5 adaptive step size adjustment procedure for the flow field integration, the precision of the particle's movement can be intuitively controlled by adjusting the error tolerance factor τ_0 and the minimal step size h_{min}. But a too strict error tolerance leads to a much higher computational effort as far more sub-steps need to be performed. This would slow down the whole particle advection process and would make the procedure virtually useless. Therefore, besides the performances of the different integration schemes for different platforms, also the impact of the adaptive step size adjustment procedure on the overall performance is analyzed in the following section.

4.4 Integration Performance Evaluation

The described method of particle tracing in time-variant tetrahedral grids was implemented to run on the CPU as well as on the GPU. Therefore, the performance of both platforms can be directly compared in terms of performance and particle throughput. The focus lies especially on the comparison of the different integration schemes that were used to integrate the flow field velocities in order to calculate the particle advection. Depending on the order of the integration scheme, several flow field evaluations have to be performed using the point-location scheme for tetrahedral grids described in Section 3.2.

The CPU-based implementation was designed to run on a single core. In order to obtain the theoretical peak performance of a system, the particle throughput was measured for a single core and then multiplied by the number of system's cores. The runtime of the CPU-based implementation was measured on an Intel Core 2 Quad Q9550 system with 4 cores, each running at 2.83GHz, as well as on a Dual Intel Xeon E5420 system with a total amount of 8 cores, each running at 2.50Ghz. These systems are referenced as Core 2 Quad and Xeon MP.

The performance of the GPU-based implementation using the CUDA framework was profiled on a GeForce GT240 with 1GB of device memory, as well as on a Quadro FX 5800 with 4GB of memory by utilizing the CUDA Visual Profiler [NVI10d]. The GeForce GT240 features 12 multiprocessors with a total amount of 96 cores, whereas the clock rate of the processors is 1.34Ghz. This GPU represents a typical consumer model with average performance. The Quadro FX 5800 on the other hand represents a professional high-end graphics card, equipped with 30 multiprocessors running at 1.30Ghz and a total amount of 240 cores. These systems are referenced as GT 240 and FX 5800.

Figure 4.9 shows the performance of the different integration schemes for the described platforms. On each platform, the particle advection was performed for several steps

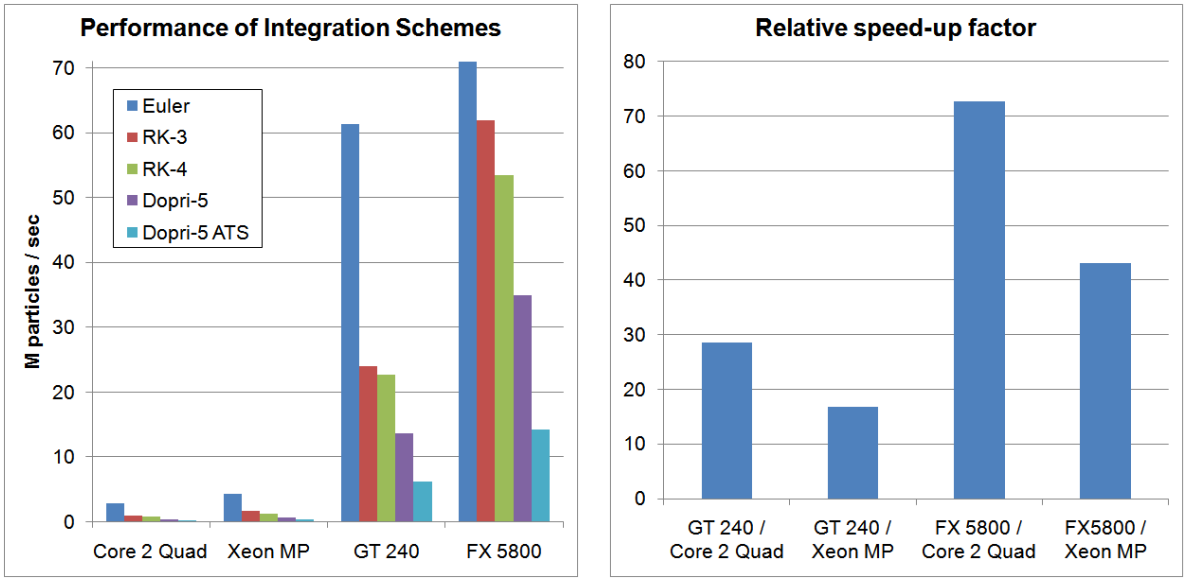


Figure 4.9: Median performance and relative speed up of flow field integration process on various platforms using different integration schemes. The particle throughput of the Quadro FX 5800 using the Euler integration scheme tops off at about 190M particles per second.

using the Euler and the Runge-Kutta integration scheme of third and fourth order, as well as the Dopri-5 integration scheme, once with and once without adaptive step size adjustment, in order to compare the impact of the adaptive step size adjustment procedure on the flow field integration performance. The performance throughput is given as million particles per second, in order to obtain comparability with the performance measurements of previous works, e.g. [Sch08].

For all platforms, the best performance was measured using the Euler integration scheme as this integration scheme requires only one flow field evaluation in both temporal domain states to calculate the new position of a particle. If a higher-order integration scheme, like the third- or fourth-order Runge-Kutta method is used to calculate the particle advection, a higher accuracy of the particle movement is obtained, but at the cost of a much lower particle throughput, as more computational costly flow field evaluations are required. By using the Dopri-5 integration scheme, the particle throughput is again lower, as it performs three additional flow field evaluations per temporal state compared to the fourth-order Runge-Kutta method. The performance of the Dopri-5 integration scheme with adaptive step size adjustment was evaluated using a maximum error tolerance factor $\tau_0 = 10^{-4}$ as upper threshold for the error of the fourth-order accurate solution of the flow field integral. This very low error tolerance shows an explicit impact on the particle throughput performance compared to the Dopri-5 integration scheme without adaptive time step adjustment, as each flow field integration step with an error value above the tolerance is repeated with a reduced step size.

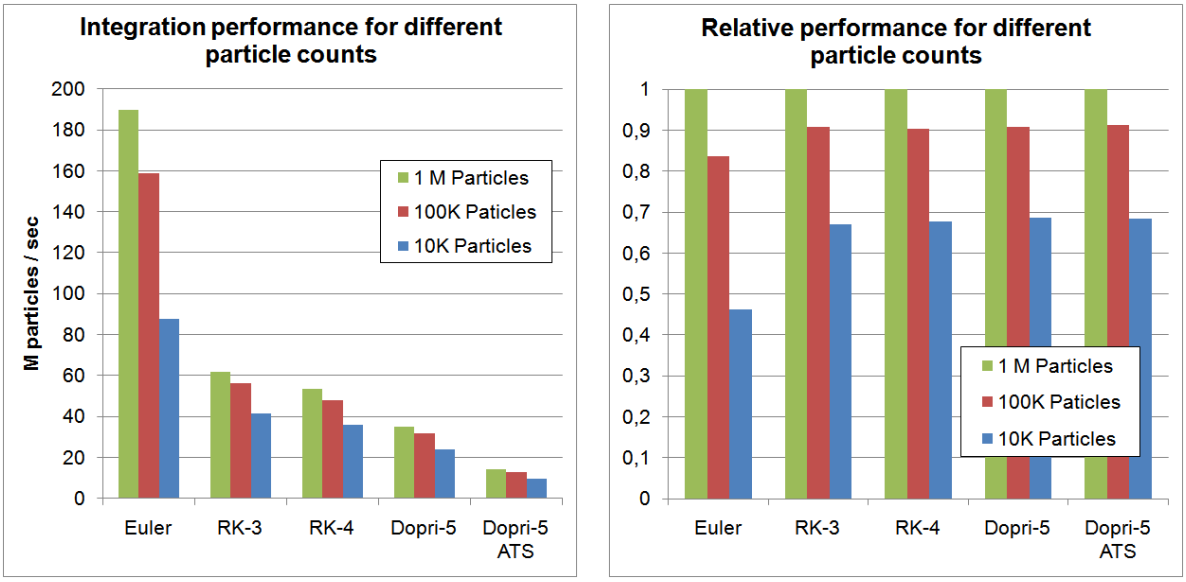


Figure 4.10: The throughput of the GPU-based implementation for different integration schemes also depends on the number of particles that are advected in parallel.

Figure 4.9 also depicts the measured relative speed-up factor of the GPU-based implementation compared to the results of the CPU-based systems. This diagram shows, that a large performance boost can be obtained by shifting computations from the CPU to the GPU as the GPU-based implementation outruns the CPU-based implementation for every measured integration scheme. This leads to the conclusion, that the described tasks are very suitable to be performed on the graphics processing unit.

Another interesting fact is, that the performance of the GPU-based implementation depends on the number of particles that are advected in every step. Figure 4.10 shows the relative and absolute particle throughput for different particle counts for the different integration schemes. As can be seen, a higher throughput is achieved if the integration is calculated for a larger number of particles. This is due to the overhead that is introduced by the invocation of the CUDA-based computations, e.g. by binding the current tetrahedral grids to the texture memory etc. If more complex computations are performed, e.g. by using a higher-order integration scheme, the amount of overhead gets smaller compared to the overall computational workload.

The particle throughput is also affected by the chosen step size of the advection step. The absolute and relative performances of the different integration schemes for an altered step size are shown in Figure 4.11. The highest particle throughput is achieved for a basic step size of $\Delta t = 0.02$. If the step size is increased by a factor of 5, the performance decreases to about 40 percent, and even more if the step size is increased by a factor of 10. The performance break-in can be explained by the increased number of cells that are traversed during the flow field velocity evaluation. If the particles are advected using

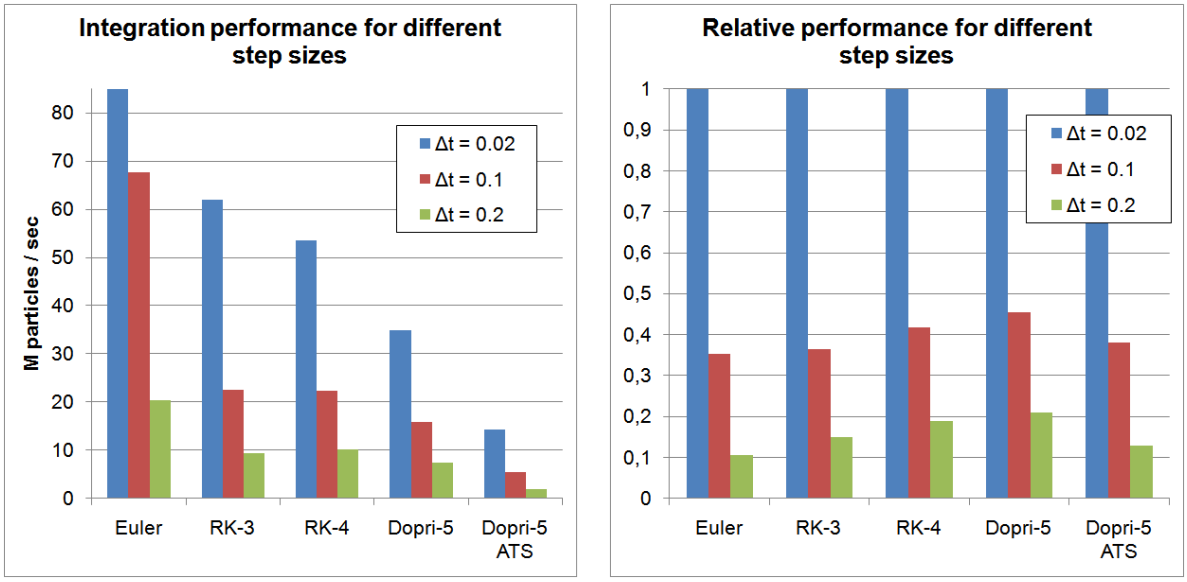


Figure 4.11: A larger step size reduces the particle throughput for all integration schemes, as more cells need to be traversed while the flow field evaluation. The particle throughput of the Euler integration scheme for $\Delta t = 0.02$ tops off at about 190M particles per second.

a large step size, the distance between the evaluated positions increases and therefore longer tetrahedral walks must be performed. The average number of traversed cells for the different integration schemes and step sizes is shown in Table 4.3.

The highest performance loss for the flow field integration with different step sizes was measured when performing the particle advection using the Dopri-5 integration scheme with adaptive step size adjustment. While the overall step size is increased, the size of the sub-steps remains constant in order to attain the desired accuracy. Therefore, much more sub-steps need to be calculated for the global advection step, which also increases the number of flow field evaluations and therefore the computational workload. Figure 4.12 shows the average number of traversed cells as well as the average number of iterations required for different error tolerance values when using the Dopri-5 integration scheme

	Integration scheme				
Step size	Euler	RK-3	RK-4	Dopri-5	Dopri-5 ATS
$\Delta t = 0.02$	0.547	0.583	0.571	0.565	0.33844
$\Delta t = 0.1$	4.561	4.724	4.384	4.231	1.55382
$\Delta t = 0.2$	6.083	10.735	9.574	9.462	2.48518

Table 4.3: Average number of traversed cells for different integration schemes and step sizes.

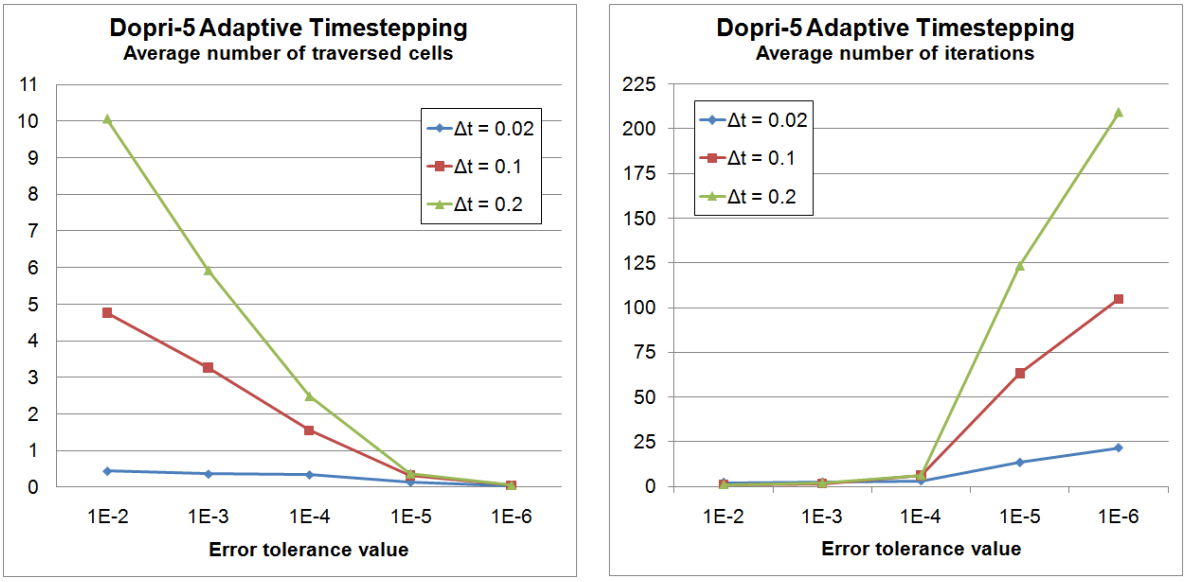


Figure 4.12: Average number of traversed cells and iterations for different error tolerance values using the Dopri-5 adaptive time stepping for different global step sizes.

with adaptive step size adjustment for different global step sizes. If the error tolerance value is chosen too strict, the number of iterations rises quickly while the respective step size tends to zero.

In conclusion, by utilizing modern programmable graphics hardware for the calculation of the flow field velocity integration, millions of new particle positions can be calculated per second, even if the described adaptive step size adjustment procedure with controllable error tolerance is used, which gives a highly accurate solution of the flow field integral by performing several sub-steps of varying step sizes. Nevertheless, the flow field integration is the most time-consuming part of the particle tracing approach in time-variant tetrahedral grids.

CHAPTER 5

CUDA-BASED GPU IMPLEMENTATION

5.1 General Purpose Computations on Graphics Processing Units

General Purpose Computation on Graphics Processing Units (GPGPU) describes the usage of graphics hardware for computations other than their usual purpose, the processing of graphics. With GPGPU, all kinds of algorithms for diverse problems, e.g. in the field of physics, mechanics or economy, are implemented to run on the GPU instead of the CPU. Hereby, an advantage is taken from the rapid growth of GPU performance due to the market demand for ever increased real-time, high-definition 3D graphics, e.g. for the creation of almost photo-realistic scenes within 3D-games.

The application of GPGPU started with the introduction of the programmable graphics pipeline, which is an extension of the standard graphics rendering pipeline (cp. Figure 5.1). This extension allows to specify vertex- and fragment shader programs within the rendering process, e.g. by using a high level shading language like the OpenGL Shading Language GLSL [Ros09].

For computations on the GPU, the algorithms are hereby mapped to the graphics rendering pipeline by defining graphical primitives and specifying textures for the data input. The computations are implemented by writing vertex- and fragment shader programs and performed by executing the rendering process. After the rendering has finished, the results are obtained by reading out the frame buffer. This technique was also used in previous works about GPU-based particle tracing [KKKW05, Sch08].

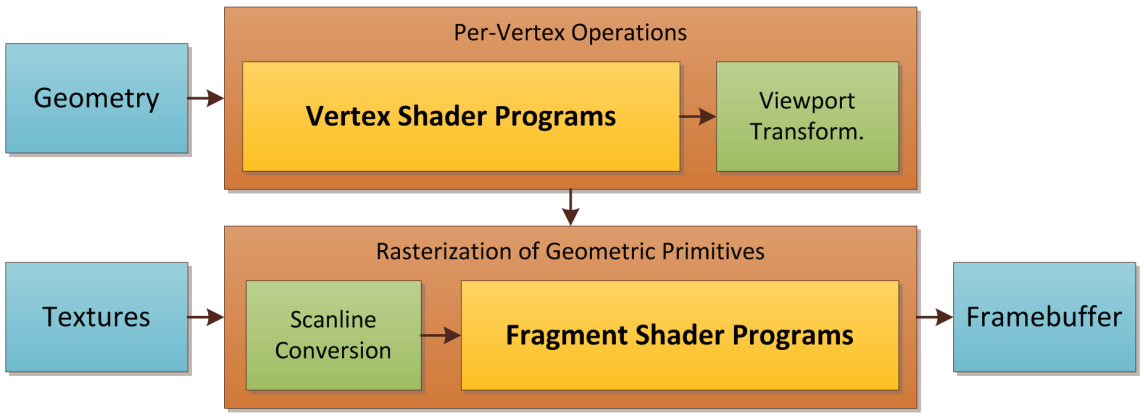


Figure 5.1: The graphics rendering pipeline can be modified and used for GPGPU by specifying vertex- and fragment shader programs.

Using the graphics rendering pipeline for GPU-based computations needs a high level of abstraction, as the algorithms for a specific problem need be implemented as graphical shader programs. With the increasing of GPU performance, this technique was more and more able to boost the computational speed of various implementations for all kinds of problems. This development was also recognized by graphics card manufacturers, which see the chance, to place their products not only for fast graphics rendering, but also as co-processors, e.g. to amplify video encoding performance [Ele10] or to create enhanced physical effects within games [NVI10c].

In order to alleviate the usage of GPU-based computations, the graphics card manufacturers developed special application programming interfaces (API), which allows shifting calculations to the GPU without utilizing the graphics rendering pipeline. The two largest producers of programmable graphics hardware besides Intel, NVIDIA and AMD/ATI, each designed their own API, which was adapted to their respective hardware specifications. These days, also unified APIs for all recent graphics cards are available, including *OpenCL* by the Khronos Group [Khr10a] and *DirectCompute* designed by the Microsoft Corporation as part of their graphics API DirectX 11 [Mic10].

In this thesis, the Compute Unified Device Architecture (CUDA) introduced in November 2006 by the NVIDIA Corporation is used to implement GPU-based calculations. The CUDA framework was specially designed for computations on recent NVIDIA graphics cards, which are widely used in current computer systems. For the implementation, C for CUDA is used, which is an extension of the regular C/C++ language. There also exists a variety of wrappers for other languages, including Java, Python, Fortran and MATLAB. CUDA has some advantages over GPGPU calculations using graphics APIs, as it allows random access to the graphics card memory, features fast shared memory between threads and full support for integer and bitwise operations [NVI10a].

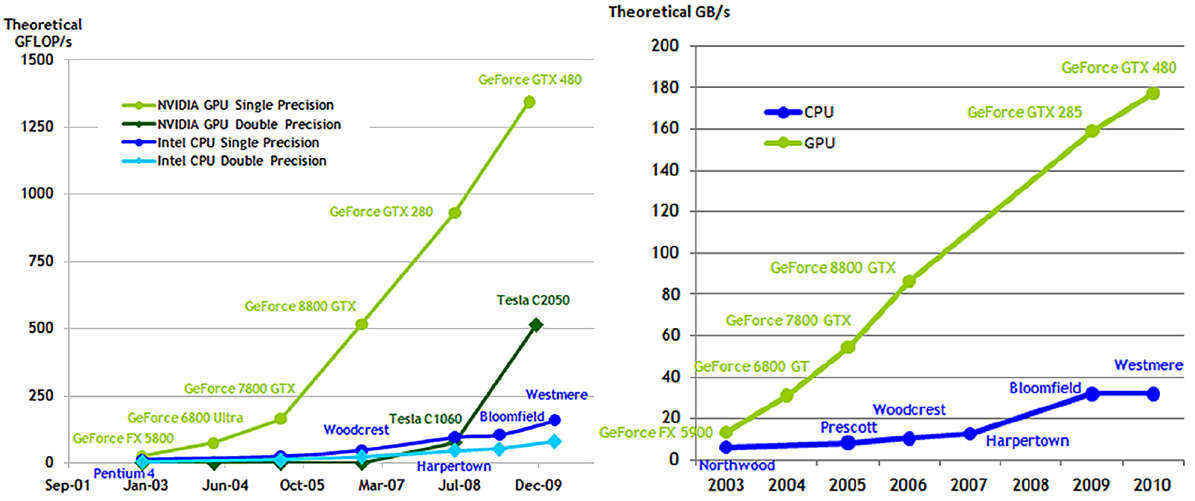


Figure 5.2: The development of theoretical peak performance of floating-point operations per second and memory bandwidth for CPU- and GPU processors [NVI10b].

5.2 GPU Architecture

Today's programmable graphics hardware features many-core processors with highly parallel, multithreaded computational power, as well as high bandwidth to the onboard graphics memory. The theoretical peak performance of floating point and double precision operations per second of CPU- and GPU processors, as well as the maximum bandwidth of (device-) memory is depicted in Figure 5.2, taken from [NVI10b]. It also includes the performance of the latest generation of Fermi- (Geforce GTX 480) NVIDIA GPUs. Of course, these values are determined by summing up the peak performances of all available cores, a state of which is almost never achieved in practice. But, there is a certain performance gap between GPUs and CPUs.

The performance differences of single-precision operations between CPUs and GPUs are based on the distinctions of both architectures. GPUs are specialized for 3D graphics rendering, which introduces highly parallel computations, e.g. in the calculation of picture elements. For the graphics rendering process, it is not that relevant, how long it takes to calculate the color of one pixel on the screen, but to calculate the whole frame within a small period is much more important. This can only achieved by massive parallel processing.

The architecture of current CPUs on the other hand is designed to complete a single task as fast as possible. On modern super-scalar out-of-order CPUs, for example, the control logic of the processor tries to find instructions within the program code, that can be processed in parallel (ILP: Instruction Level Parallelism, [SL05]). Also, it is often necessary to switch between different tasks, e.g. in order to react on interrupts by

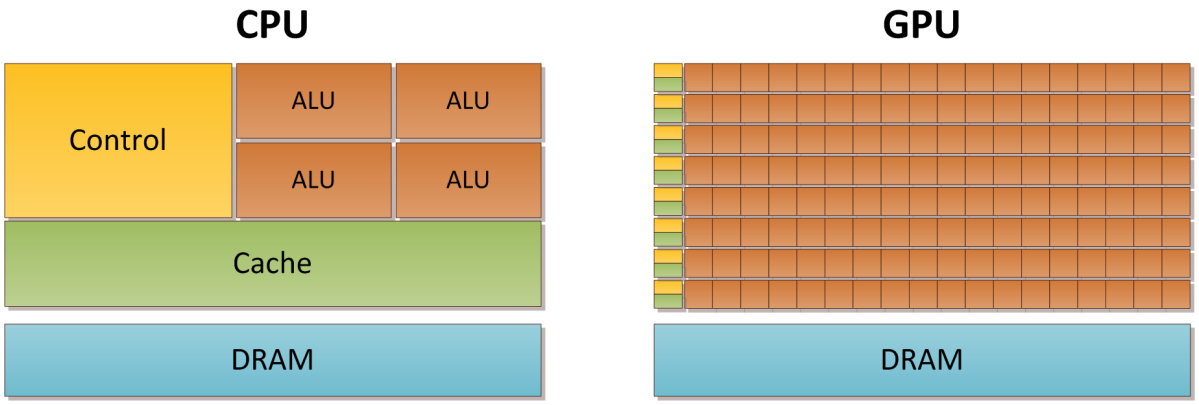


Figure 5.3: A schematic illustration of modern CPU and GPU architectures [NVI10a]. The GPU architecture is especially designed for computations with high arithmetic intensity.

the operating system. Therefore, common CPU architectures feature large units for the control logic, as well as big data caches, in order to switch fast between different process contexts.

In contrast to the CPU architecture, GPUs are designed for fast parallel data processing, where a single program is executed on many data elements, rather than data caching and flow control. As only one program is executed at a time, there is no need for sophisticated control logic units. For programs with a high arithmetic intensity - the ratio of arithmetic- to memory operations - memory access latency can be hidden with calculations instead of big data caches. Therefore, GPU-based computations are most suitable for problems that can be divided into many independent tasks and processed in parallel. Both architectures are schematically illustrated in Figure 5.3.

The structure of modern GPUs is build around several streaming multiprocessors (SMs), each of which can run hundreds of computational threads concurrently. The CUDA framework introduces an architecture called *SIMT* (Single-Instruction, Multiple-Thread) describing thread-level parallelism. This is akin to SIMD (Single-Instruction, Multiple Data) with the main difference that all threads of a warp have their own address counter and register state and are therefore free to branch and execute independently. Hereby, a warp describes the number of threads executed in parallel on a single SM, whereas the size of a warp is equal to 32 on present GPUs. Of course, full efficiency is only achieved, when all threads of a warp agree on their execution path. If some threads diverge, e.g. by a data-dependent conditional branch, they are serially executed while the other threads are disabled until all threads converge back to the same execution path.

On the hardware level, a multiprocessor can have several active warps, between which is switched, depending on whether a warp has active threads ready to execute. Threads of different warps on the same multiprocessor can share data via a fast, on-chip shared

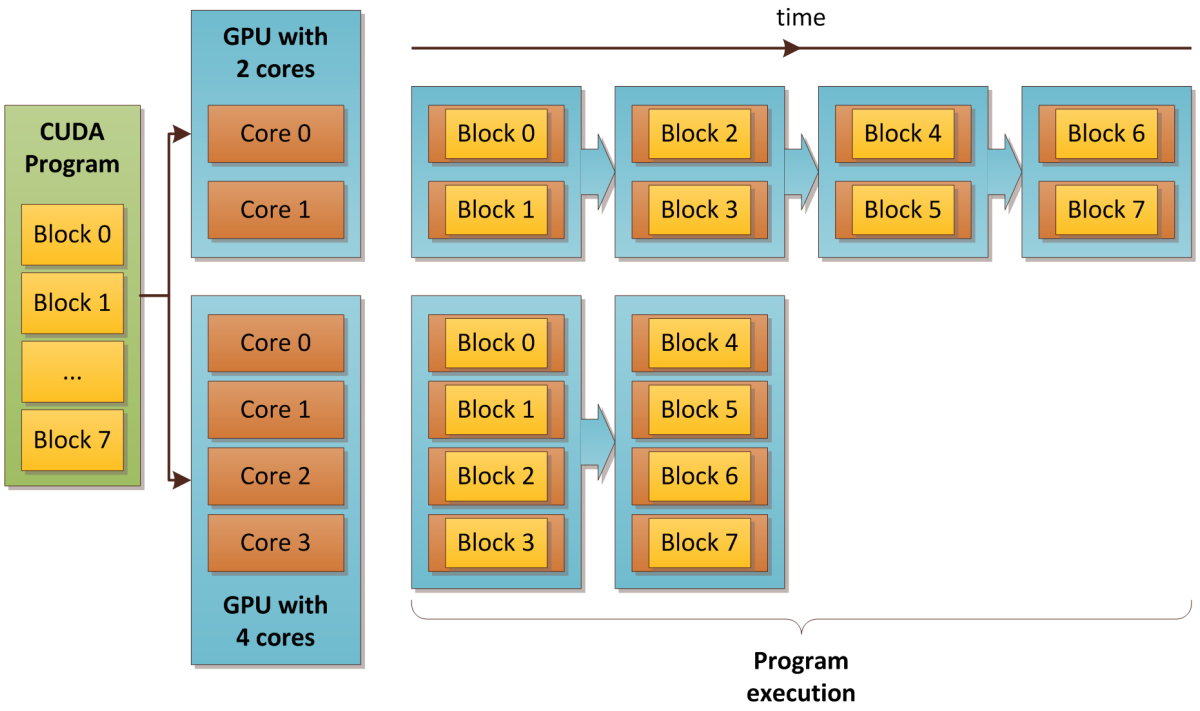


Figure 5.4: The CUDA architecture requires, that a problem can be partitioned into blocks of sub-problems that can be calculated independently from each other. Then, the time needed to solve the problem scales with the number of available processor cores.

memory. The number of warps, that can reside together on a multiprocessor hereby depends on the usage of registers and shared memory by the threads and the register and memory capacities available on the multiprocessor.

In order to map threads to the SMs, the threads are organized as thread blocks and all threads of a block are expected to reside on the same multiprocessor. The capability of the processor hereby defines the maximum number of threads per block. For the execution, a block is divided into warps of 32 parallel running threads, where a block typically has more than one warp, so that the execution of warps can overlap, e.g. to hide memory access latency.

The block- and thread-based CUDA architecture requires, that a problem can be partitioned into coarse sub-problems that can be solved independently from each other by blocks of threads. In each block, the sub-problem must again be dividable into finer pieces, which can be solved in parallel by all threads within the block. As an absolute requirement, it must be possible to run each block of threads independently from the other blocks, on any available processor core and in any order, concurrently or sequentially. This ensures, that the performance of a CUDA-based program scales with the number of available processor cores on the GPU, as long as there are enough blocks to process (cp. Figure 5.4).

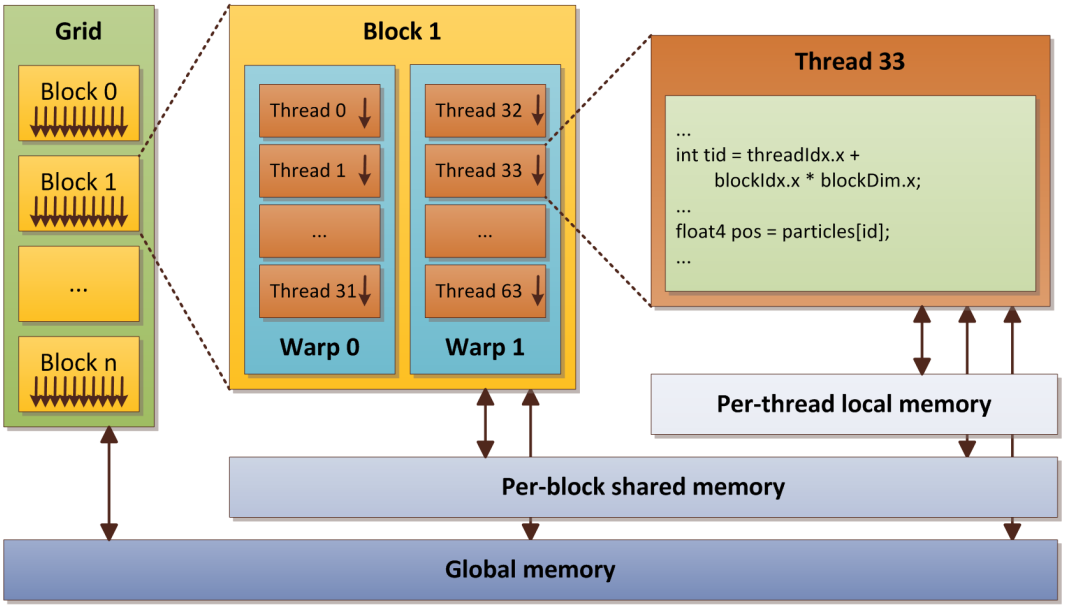


Figure 5.5: The CUDA programming model organizes the threads in blocks and the blocks as grid. The threads of a block are executed in parallel by warps of 32 threads. For each thread, a unique id is obtained by the thread index, as well as the index and the size of its thread block.

5.3 The CUDA Programming Model

Using the CUDA programming model, GPU-based programs are implemented as C functions called *kernels*. When such a kernel is called within the application context, it is executed N times in parallel by N different threads. For the kernel calls, a special syntax is introduced, which defines, how the threads are organized in one-, two-, or three-dimensional blocks. Within each block, the threads are references by a consecutive, three-dimensional index variable.

As depicted in Figure 5.5, the blocks are again organized in a one- or two-dimensional grid structure, by which each block gets its own index. This allows the executing of more threads at once than would fit into a single block. Within the kernel program, special variables allow to access the thread index w.r.t. the current block, as well as the block index and block dimension within the grid. Therefore, the kernel can access arrays by a consecutive, unique id for each thread, e.g. to update the position of each particle stored in an array.

Figure 5.5 also shows the memory hierarchy of the CUDA programming model. Each thread has its own local memory accessible only from within the thread's context. All threads of a thread block can share data by using the per-block shared memory. The global memory is used to share data between all blocks of the grid and is persistent

across launches of different kernels. Therefore it is possible to use within a kernel the calculation results from a previous kernel launch.

The local- and shared memory banks are very limited in size, but they provide much lower access latency than the global memory, as they are located on-chip of each multiprocessor. The global memory, on the other hand, offers a much higher capacity, but also has high access latency. To lower the access latency, CUDA allows mapping areas of the global memory to the on-chip texture memory unit, which acts as a cache. Then, a read access to the global memory is served by the texture memory, if it was already requested before. The global memory of the GPU is also accessible for the host system via special commands in order to copy data to and from the graphics device.

5.4 CUDA-Based Particle Tracing

The CUDA architecture perfectly fits the task of particle tracing through a time-variant flow field, as the movement of each particle can be calculated independently from the others. This is due to the fact, that the update of the particles' positions depends solely on the flow field velocities, as depicted in Figure 5.6. By performing the whole particle advection calculation on the GPU, the overall performance is boosted, as each position update is calculated by a single thread and many of those threads are calculated in parallel. The size of the thread blocks can be chosen arbitrary in order to maximize the usage of processor capacities for various CUDA-enabled GPUs. Thus the time to complete a single advection step for a vast amount of particles scales with the number of available SMs.

The position of each particle is stored in an array within global memory as a vector of 4 floating-point values per entry, holding the three dimensional components of the position inside the domain. Within the fourth component, positive values are used as an optional counter to limit the number of advection steps until a particle is deleted. A negative value within the fourth component marks a particle as out-of-field, by which it can be overwritten with a new particle in the next seeding phase. As the particles are stored

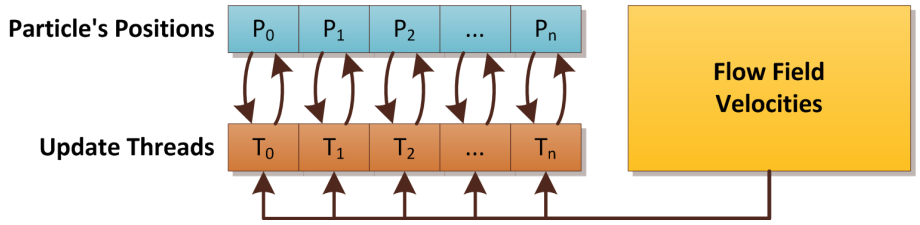


Figure 5.6: The update of the particles' positions depends only on the flow field velocities and can therefore be calculated in parallel for each particle.

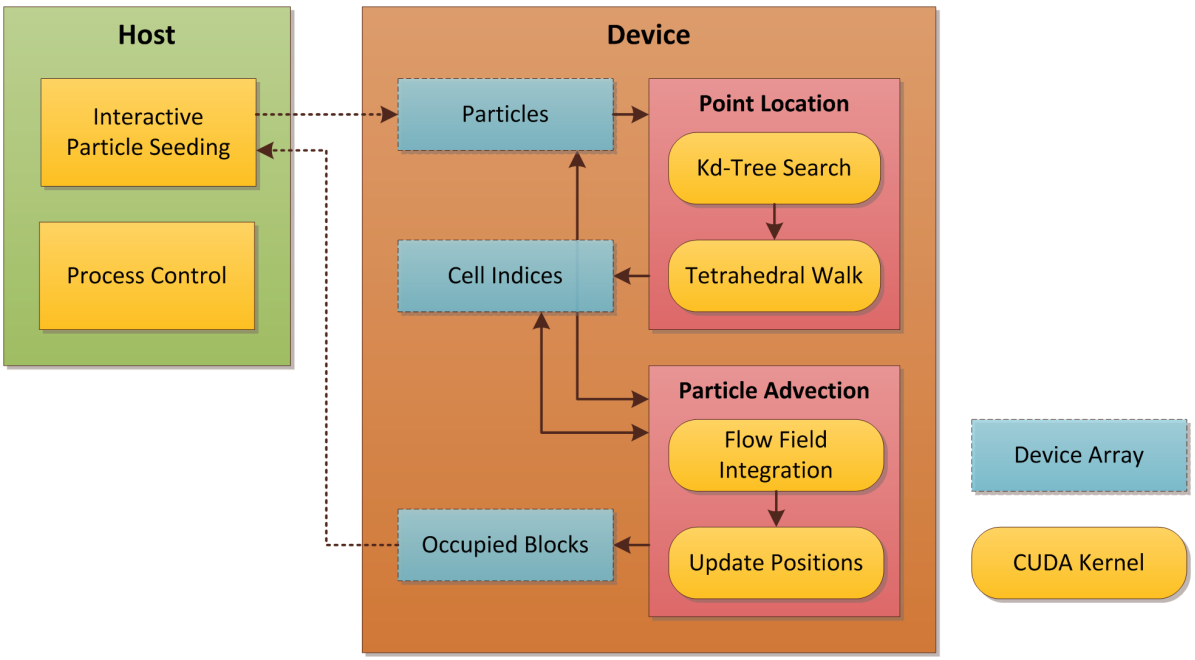


Figure 5.7: All parts of the particle advection process are computed on the GPU. Unlike previous works, this also includes the global search using a *k*d-tree structure for the point location within the unstructured grids of consecutive domain states.

in the graphics device's memory, there is no need to constantly transfer their positions to the memory of the host system and vice versa in order to visualize them. The only exception is the interactive particle seeding part, where new the particle positions are calculated on the host system and then transferred to the array of particle positions on the GPU.

As illustrated in Figure 5.7, the CUDA-based particle tracing process is implemented by four different computing kernels. Whenever new particles are seeded, the indices of the surrounding cells of their positions are computed using the two-phase scheme described in Section 3.2 and stored within extra arrays in global memory for each current time step. The particles' positions array as well as the arrays of cell indices are continuously updated while the particle advection process.

The point-location scheme is implemented as two kernel programs, one for the *k*d-tree-based nearest-grid-node search and another one for the tetrahedral walking procedure. The *k*d-tree traversal is hereby implemented using the Single-Pass method, as it offers the best performance while the accuracy is not explicitly worse than using another search method (see Section 3.3). Also, some device memory and transfer time can be saved by storing the resulting cell indices directly within the leaves of the *k*d-tree, which makes is unnecessary to transfer and store the node-to-cell lookup table.

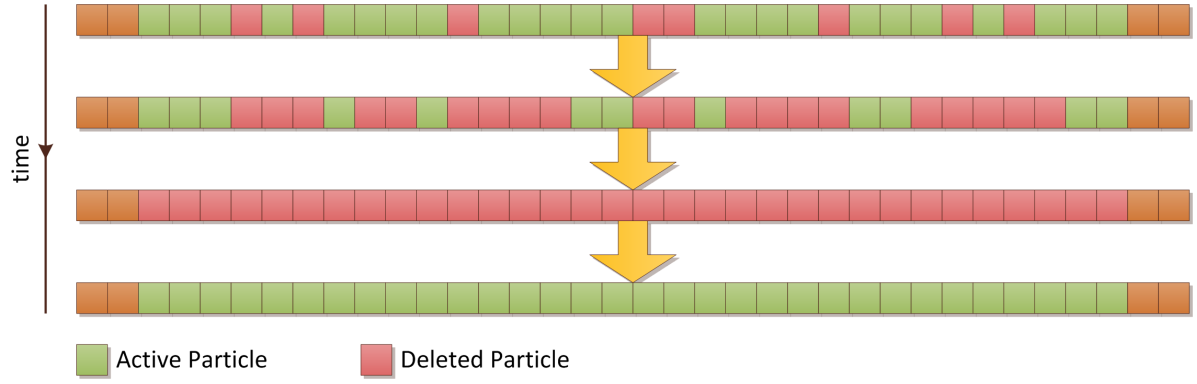


Figure 5.8: The positions of particles that have left the flow field are marked as out-of-field. If all particles of a partition of 32 positions are marked, the whole partition is overwritten by newly seeded particles within the next seeding phase.

The particle advection process is also divided into two phases. In the first phase, the particles' positions are used as input for the flow field integration, which calculates a velocity vector for each particle and stores the results in an additional array. Hereby, one computing kernel was implemented for each integration method that was described in Section 4.2 including the adaptive step size adjustment procedure using the embedded Dopri-5 integration scheme. In each flow field integration kernel, the velocities within both current time steps are evaluated either once using the Euler integration scheme or several times using a higher order integration scheme while the flow field evaluation is embedded into a point location scheme similar to the one described in Section 3.2.3. Particles that could not be located during the flow field integration are marked as out-of-field. Also in each stage of the integration, the velocities are interpolated according to the progress of time. For the Dopri-5 adaptive time stepping method, an additional array within global device memory is used to store the size of the last regular sub-step for each particle (not the last performed step, which is usually very small), which acts as input for the next particle advection step. In the second phase of the particle advection process, another kernel program updates the particles' positions by loading their positions from global memory and adding to each position the respective previously calculated velocity vector multiplied by the current step size. Particles that were marked as out-of-field during the flow field integration are deleted by placing them to a region out of sight.

As the number of positions within the particles' array is constant, the positions of deleted particles are re-used, in order to keep the total amount of particles within the bounds of the array. Therefore, the position update kernel records free partitions within the position's array where all particles have been deleted. As depicted in Figure 5.8, the array is divided into partitions of 32 particles, which matches the size of a warp of concurrently updated positions. To detect whether all particles within a partition are marked as out-of-field, a Boolean value is declared in shared memory, which is initially set to 1. If one of the threads within a warp detects an active particle to update, the value is set to 0, indicating that this block of particles is still occupied. As soon as the

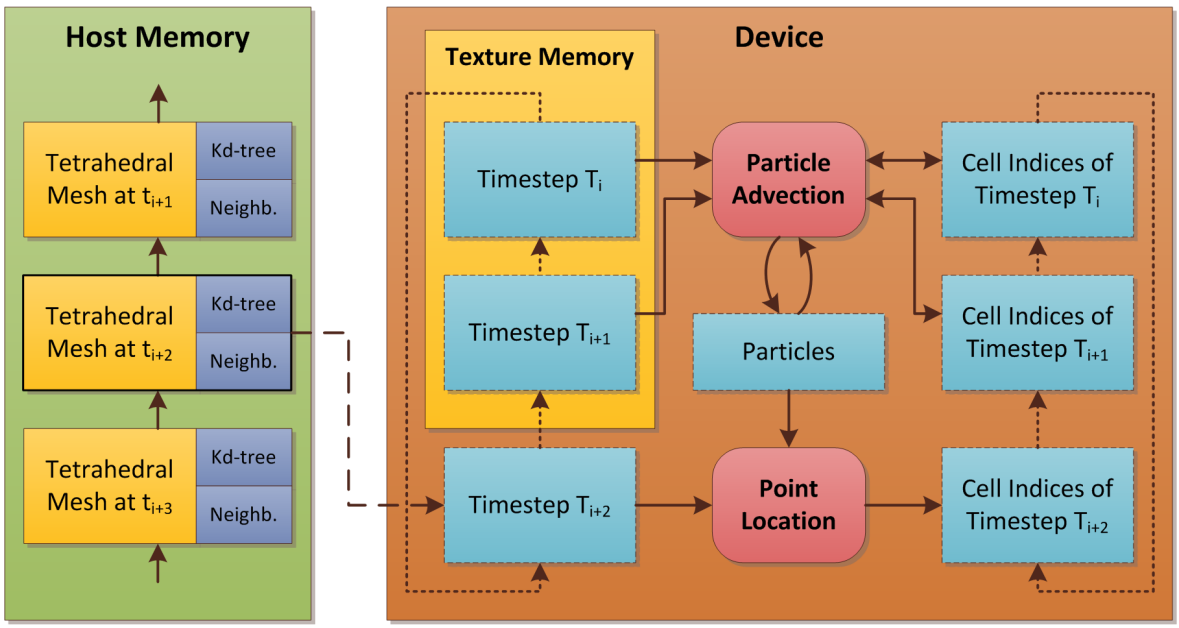


Figure 5.9: The time step data of all regarded domain states is loaded to the host's memory first and then streamed continuously the device's memory, including the additional overhead for the two-phase point location scheme.

warp finishes, the first thread copies the state of the Boolean value from shared memory to an array of occupied blocks in global memory. As depicted in Figure 5.7, this array of occupied blocks is transferred to the host system's memory every time new particles are created in order to distribute the new particles to unoccupied blocks within the array of particles in global device memory. This proceeding allows filling up orphaned regions within the particles' array in order to maintain a dense distribution of positions.

Under normal circumstances, the device's global memory is not capable of holding the entire time-variant dataset off all time states, including the additional overhead of the kd-tree search structure and the cells' neighbors. The host system in contrast features a much higher memory capacity with additional background storage in form of hard disk drives. Therefore, the complete dataset is loaded into the memory of the host system first and then transferred continuously to the device's global memory (cp. Figure 5.9).

In order to calculate the particle advection through the time-variant flow field on the GPU, the partial dataset of at least two consecutive time steps must be available in the device's memory, each given as a tetrahedral grid with embedded velocities as well as the index structures for the point location. The velocities of those two time steps are linearly interpolated while the flow integration to account for the time-varying nature of the flow field according to Equation (4.4). For the data transfer from the host to the graphics device, the dataset of a third time step is hold in the global memory of the GPU, which buffers the upcoming state of the domain in the next time step. Whenever

a new temporal state of the domain is reached, the references to the two current time steps are updated and the particles are located within the new dataset of the third time step. This is done by invoking the point location kernel for the whole particle population in order to update the cell index of each particle in the new time step as starting point of the tetrahedral walk performed by the flow field integration. After that, another time step dataset is transferred to the global device memory, which replaces the data of the previous time step.

On devices that support concurrent copy operations and kernel executions, the data transfer is performed asynchronously with the execution of the particle location and the advection kernels. For the concurrent operations, two different computing stream ids are utilized, one for the kernel execution and another one for the data transfer. Whenever the references to the current time steps are updated, both stream ids are swapped, to ensure that the asynchronous transfer of the next time step has finished. To perform the data transfer asynchronous, the dataset of the next time step on the host's side needs to be located within page-locked memory, a special kind of memory resource provided by the operating system. This kind of memory is a very limited resource of the host system and therefore allocated dynamically and filled by copying the time step data from pageable to page-locked memory before starting the transfer.

The flow field evaluation requires fetching node- and cell data from randomly distributed locations within the global device memory. As this random access would typically lead to high memory latency and therefore reduced performance, the data of both current time steps is bound to texture memory in order to cache reads from the global memory. Hereby, advantage is taken from the fact, that particles, which were seeded together, have similar trajectories through the flow field and therefore cross the same cells within both tetrahedral grids. Hence, caching the data for the nodes, velocities and neighbors of those cells is highly efficient and leads to improved performance and higher particle throughput.

5.5 Host System Implementation

The software implementation of the methods presented in this thesis is divided into CUDA-based components running on the GPU, which perform most of the calculations including the point location, the particle advection and the flow field integration, as well as several components running on the host system. Those components mostly handle the data loading, the control of the data flow and the invocation of CUDA computing kernels.

5.5.1 Data Flow Controller

The data flow controller is the central component of the host system implementation. It controls the whole data flow of time step data to the GPU, handles the visualization time and steers the CUDA-based computations. This component provides functions for adding time step data and binding those data to a certain point in time. The time step data hereby consists of an unstructured tetrahedral grid, a kd-tree, which indexes the nodes of the tetrahedral grid and the indices of the neighboring cells for each tetrahedral cell. The dataflow controller invokes the kernel for the location of newly seeded particles within both current time steps and the kernel which calculates the advection of the particles with the chosen step size using one of the implemented integration schemes.

By binding each time step data to a specific point in time, the intervals between the temporal domain states can be arbitrarily chosen. The data flow controller handles the progress of visualization time during the particle advection and decides, which time step data is actually used for the temporal interpolation of the time-variant flow field. Therefore, three consecutive time steps on the GPU are referenced: The two current time steps T_i and T_{i+1} as well as the next time step T_{i+2} (cp. Figure 5.10). In Addition, the data flow controller holds the references to the respective cell indices of the particles for each time step (cp. Figure 5.9). While the time steps T_i and T_{i+1} are used for the particle advection through the time-variant flow field, the reference to the next time step T_{i+2} is used to simultaneously transfer new data to the GPU.

If the time intervall of the current advection step crosses the point in time defined by T_{i+1}, the advection step is divided into two half-steps. As soon as the first advection step within both current time steps T_i and T_{i+1} has finished, the references to the time step data on the GPU are updated as $T_i = T_{i+1}$ and $T_{i+1} = T_{i+2}$. Then, the point-location kernel is executed to update the cell indices of the particle population within the new current time step T_{i+1} and the second advection step is performed on the new temporal domain states. The former time step T_i is now referenced as T_{i+2} and, as it is no longer used, overwritten by the data of a new time step during the transfer to the GPU. If the visualization time crosses the point in time defined by the last available time step, the visualization time is reset and the first three time steps T_0, T_1 and T_2 are transferred from the host system to the device memory in order the reset the visualization.

For each particle advection step, also several constant values are transferred to the constant device memory, including the current visualization time and the current step size. For the temporal interpolation between the current time steps T_i and T_{i+1}, also the normalized time and the normalized step size are copied, as these values are only known within the context of the data flow controller. The value of the normalized step size is hereby needed to advance the time between the stages of a higher order integration scheme.

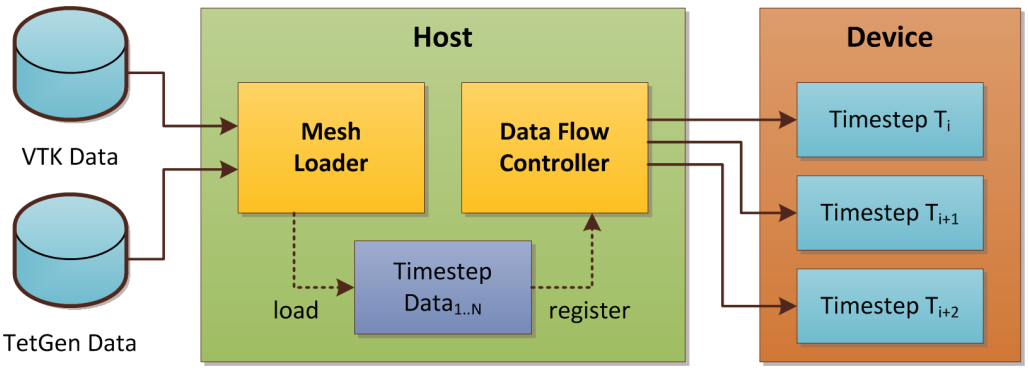


Figure 5.10: On the host side, the mesh loader component loads the time step data from mass storage into the host system's memory, while the data flow controller handles the data transfer to the GPU.

5.5.2 Time Step Loading and Processing

The mesh loader component was also implemented to run on the host system. This component is used to load unstructured tetrahedral grids from files on the hard disk into the host's memory, where each tetrahedral grid is internally represented by a set of grid nodes, the velocities defined at the grid nodes and the tetrahedron cells, as described in Section 3.1. Several file formats for unstructured tetrahedral grids are supported by the mesh loader including the *vtkUnstructuredGrid* file format defined by the Visualization Toolkit VTK [Kit10] as well as the file format of the TetGen library [WIA09].

For each loaded tetrahedral grid, the kd-tree index structure and the neighboring cell information are calculated if required and stored according to the filename of the grid. The mesh loader also recognizes previously calculated search structures by the filename of the tetrahedral grid and loads them as well, which speeds up the loading process. As depicted in Figure 5.10, the loaded tetrahedral grid together with the search structure is internally represented as the data of one time step and registered within the data flow controller by specifying the point in time within the simulation time frame, which is described by the time step.

5.6 Rendering

The described flow field visualization process requires the depiction of several graphical elements, including the flow field domain and the animation of particles, which are used visualize the time-variant flow. In addition, an illustration is given of those tetrahedron cells, which are traversed by the particles while their advection. The whole rendering process is guided by the OpenGL graphics library [Khr10b].

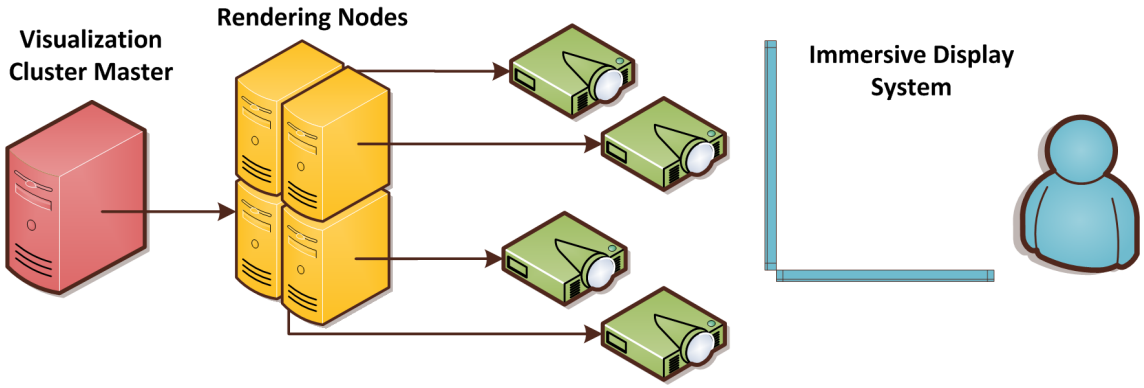


Figure 5.11: The ViSTA VR Toolkit runs on the visualization cluster master and drives several rendering nodes in order to create the stereoscopic projection of the immersive display system.

5.6.1 Virtual Environments and Immersive Display Systems

In this thesis, the ViSTA Virtual Reality Toolkit [Vir09] is used for the continuous depiction of the particle movement through the time-variant flow field, which also allows the application of the given visualization process within an immersive virtual reality setting. Immersive display systems hereby provide a stereoscopic projection, which is dynamically adjusted to the position of the viewer by utilizing a head-tracking device. This yields a holographic depiction of the scene under investigation and provides additional depth information to the viewer. Figure 5.11 shows a schematic illustration of an immersive display system, driven by several rendering nodes which are synchronized by a visualization cluster master system.

The fast drawing of particles using standard billboard rendering techniques is problematic for immersive display systems, as those billboards are typically aligned parallel to the viewing plane. In an immersive virtual reality setting using stereoscopic projection, the viewer is relatively close to the projection plane, so that the projective distortion and angular error become more apparent and the flat nature of the billboards can be recognized more easily. In his Ph.D. thesis [Sch08], Schirski proposed, that this problem can be avoided by aligning the billboards to the viewer instead. His approach was also implemented in the Vista VR Toolkit and is used in this thesis for the depiction of particles within the immersive virtual reality setting. In order to render the particles within this context, the positions of the particles are stored as a *vertex buffer object* (VBO) on the GPU, which yields high rendering performance. In order to update the particles' positions by the respective CUDA kernels, the VBO is registered as a *cudaGraphicsResource* and mapped to an array for the particle advection.

The ViSTA VR toolkit also supports several VR devices, e.g. the SpaceNavigator for easy navigation within the scene, as well as tracking devices which provide an intuitive

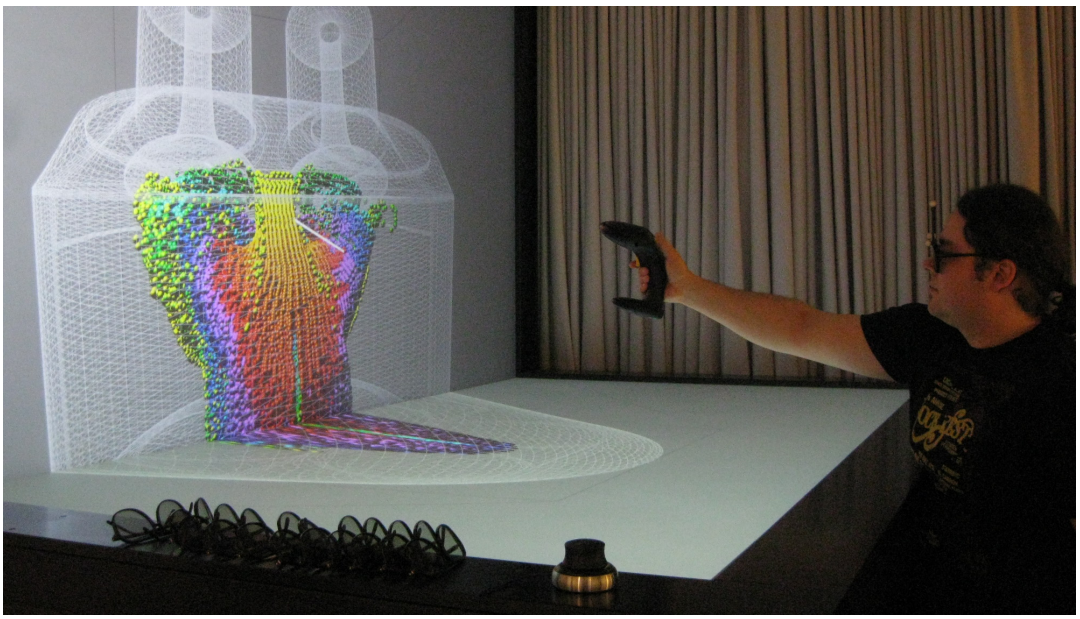


Figure 5.12: Interactive flow field exploration in an immersive virtual reality setting.

user interface for the seeding of particles within the flow field domain (cp. Figures 5.12 and 5.15).

5.6.2 Graphical Representation of the Flow Field Domain

As shown in Figure 5.13, the graphical output also contains an illustration of the current flow field domain, which is given by the outer faces of one of the current tetrahedral grids. The outer faces depict the domain as a wireframe model whereas the backfaces are optionally drawn as solid triangles (cp. also Figure 1.2 on page 5).

The list of outer faces is obtained from the mesh loader component, which tests for each tetrahedron cell while the loading or calculation of the cell neighborhood information whether one of the cell neighbors is undefined. The vertices of respective tetrahedron cell face are copied to an additional VBO on the GPU, whereas also the per-vertex face normals are calculated and stored.

5.6.3 Depiction of the Traversed Cells

Attempts to also visualize the cells of the current tetrahedral grid by simple lines have not shown pleasant results, as many of those lines overlap and single cells are not recognizable, which makes this approach virtually useless. Therefore, another approach

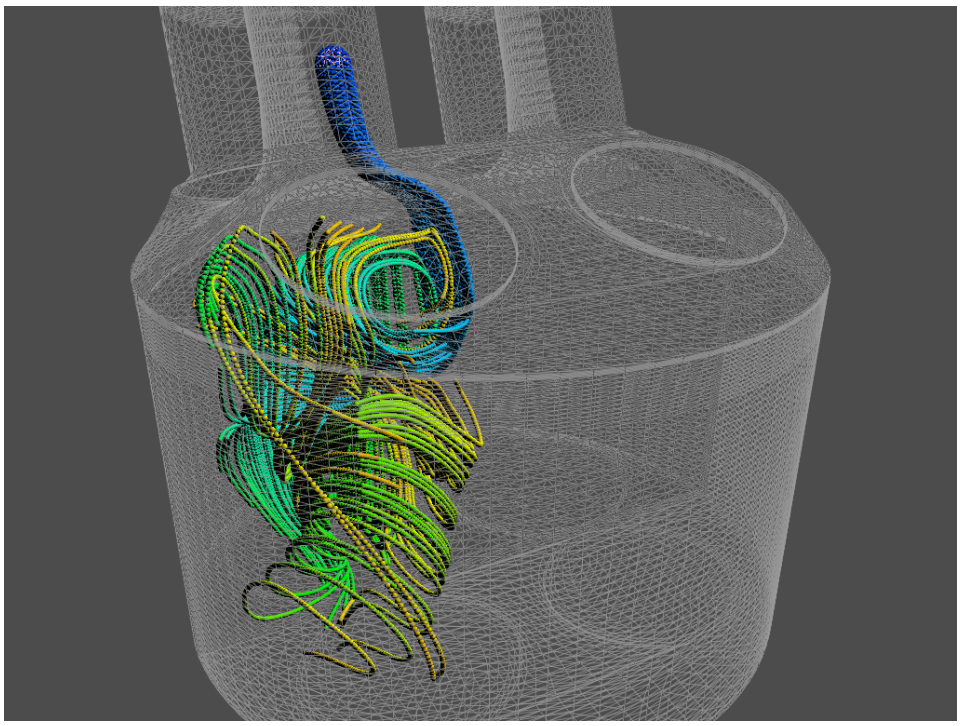


Figure 5.13: A graphical representation of the domain is given by the external faces of the tetrahedron grid. The time-variant flow field is visualized by particles seeded at arbitrary positions within the flow field domain.

was created, by which only those cells are visualized, which are traversed during the tetrahedral walks of the point location and the flow field evaluation (cp. Figure 5.14).

The depiction of the traversed cells gives a direct visual feedback of the performed calculations on the GPU and had been especially useful to detect implementation faults. Also, an illustration of the different cell sizes and the distribution of the cells at different parts of the tetrahedral grid can be obtained by seeding particles in the respective areas.

As the traversed cells would typically overlap each other, it was decided to render those cells transparently. Therefore, the cells need to be depth-sorted and rendered in the right order. As the depth sorting by the centers of the cells is performed on the host system, the indices of the traversed cells need to be transferred to the host memory. After the sorting, the traversed cells are stored in an *index buffer object* that describes each tetrahedron cell by four triangles as well as a color index buffer, which provides alternating colors for the faces. The cells are rendered back-to-front, at first as triangle lines and then as filled triangle polygons with alpha-blending enabled, which yields the transparent appearance.

Of course, this approach could be much improved in terms of performance by performing

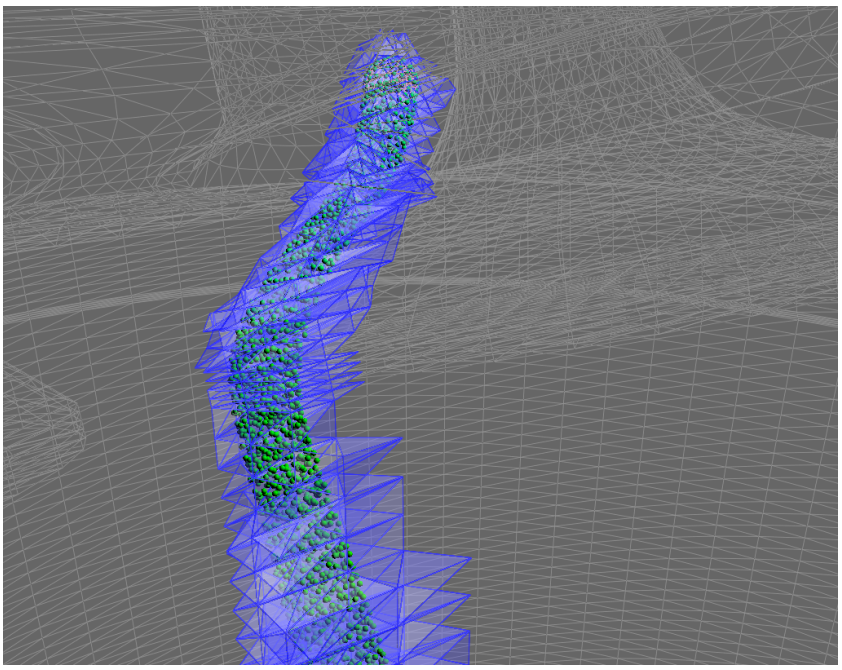


Figure 5.14: The cells of the tetrahedral grid are visualized by rendering only those cells that are traversed during the tetrahedral walk.

the sorting procedure to the GPU, which would also avoid the time-consuming transfer of data from the device to the host and vice versa. But, for small amounts of traversed cells, this already gives quite a good impression without consuming too much performance.

5.7 Discussion

By using the CUDA Framework, the particle advection process could be shifted to the GPU, where it profits from the fast parallel data processing architecture of today's programmable graphics hardware. Once the CPU-based implementation was written using the C++ language, it could be ported without effort to the CUDA programming model, as kernels are written in CUDA-C, a subset of the C/C++ language. Of course, there are some differences concerning both architectures, especially the memory hierarchy model and the massive multithreaded environment of the CUDA programming model, which had made it necessary to rewrite some parts of the implementation.

One of the major drawbacks when using the CUDA framework was the lack of suitable software for debugging purposes. The CUDA debugger *cuda-gdb* for example, which is a console debugger for GPU-based computations under Linux, requires exclusive access to the graphics device and is therefore not capable of running simultaneously to a window server like X11.

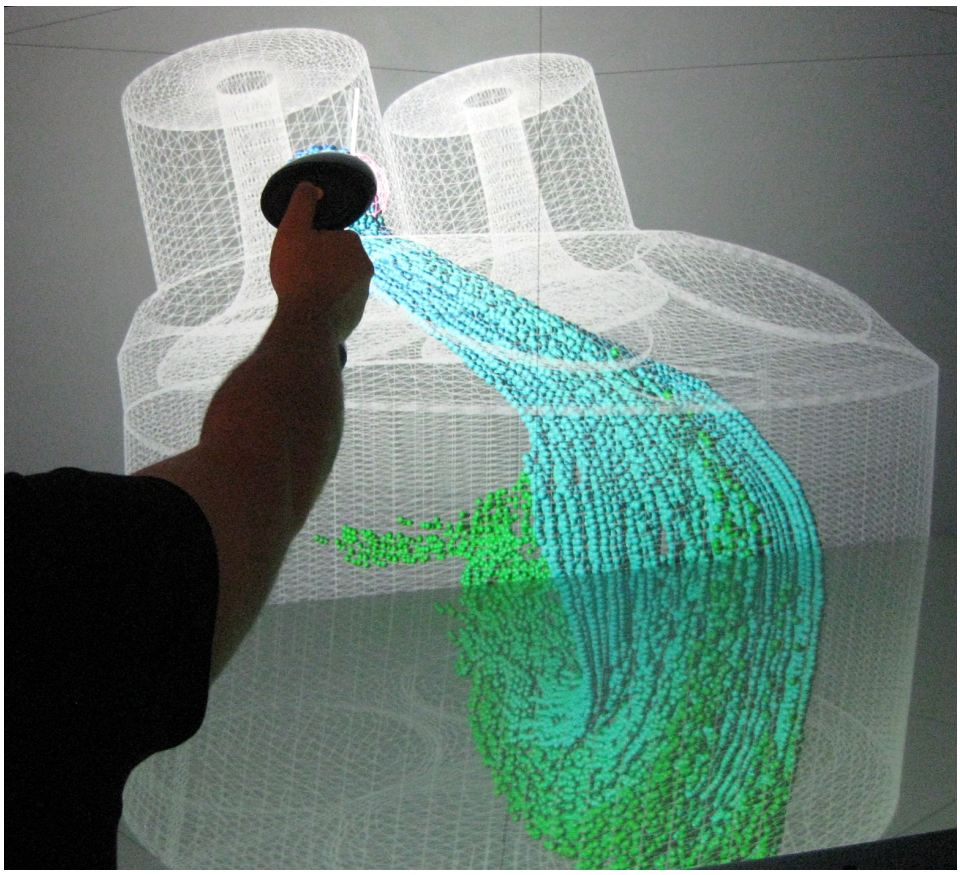


Figure 5.15: The seeding of particles by means of a tracking device with full six-degrees-of-freedom provides a highly intuitive user interface for the interactive exploration of the flow field.

CHAPTER 6

RESULTS

The GPU-based implementation described in the previous chapter was chiefly developed and tested using synthetic tetrahedral grids and flow fields that were generated as described in Section 3.4. This proceeding has the advantage, that the visual result of the particle trajectories can be compared to the predictable behavior of an artificial flow field. The described methods were also tested and evaluated on several real-world time-variant datasets. The dataset of a real-world simulation usually introduces a much higher number of grid nodes and tetrahedron cells compared to a synthetic dataset, as well as a sophisticated domain structured and a turbulent flow field.

This chapter describes the Engine dataset as an example of a real-world flow field simulation. While the complete dataset together with the additional search structure overhead has a size of 2.2 GB, which exceeds to memory capacity of most today's GPUs, the simulation is divided into several temporal domain states, which can be continuously transferred to the GPU, where the time-variant flow field is visualized by the described particle tracing approach.

While the previous chapters mostly describe each part of the implementation in detail, this chapter regards the entire procedure and presents an analysis of the approach in terms of usability and performance.

6.1 System Overview

The approach of particle tracing in time-variant tetrahedral grids presented in this thesis allows the interactive exploration of time-variant flow fields given by several tetrahedral grids resulting from real-world simulations. The user can seed particles at an arbitrary position within the flow field domain, which are immediately taken away by the flow. This proceeding is especially useful in an immersive virtual reality environment, where the particles are seeded by a tracking device with six degrees-of-freedom. The stereoscopic projection of the virtual environment as well as the dynamic adjustment of the visualized scene using head tracking give an additional depth information to the user and allows him to immerse into the flow field visualization.

To achieve interactive frame rates while visualizing the time-variant flow field using particle tracing, the integration of the flow field as well as the movement of the particles are calculated entirely on the GPU using the CUDA framework, as described in Section 5.4. Especially the calculation of the flow field integration experiences an immense speed-up when shifted to the graphics device, as the procedure fits well the massive parallel capabilities of present programmable graphics hardware (cp. Section 4.4).

Several computation kernels were implemented for the different integration schemes, which are distinguished by their order and can be dynamically replaced while the visualization proceeds. For particular precise visualization concerns, also an adaptive step size estimation procedure was implemented, using the Dopri-5 embedded integration scheme with adjustable error tolerance in the distance of the fourth- and fifth-order accurate integration results (cp. Section 4.3).

Unlike previous works, this thesis describes particle tracing on the original or decimated dataset of time-variant tetrahedralized domains, which change their appearance over time. To deal with such dynamically changing domains, the point-location scheme described in Section 3.2 was implemented to run entirely on the GPU, which allows to perform the location of many query positions in parallel. The procedure is hereby divided into two phases, a global search phase, which yields the rough position of the surrounding cell by traversing a kd-tree and a local search phase, which performs a short tetrahedral walk in which the cell neighborhood information is used to navigate through the unstructured grid.

For the traversal of the kd-tree, three methods were compared in terms of accuracy, runtime behavior and implementation effort, in order to decide which method is most suitable for the GPU-based implementation. As stated in Section 3.3, the Single-Pass method hereby fits best the requirements, as it shows the highest performance while also giving good results in terms of the described accuracy metric.

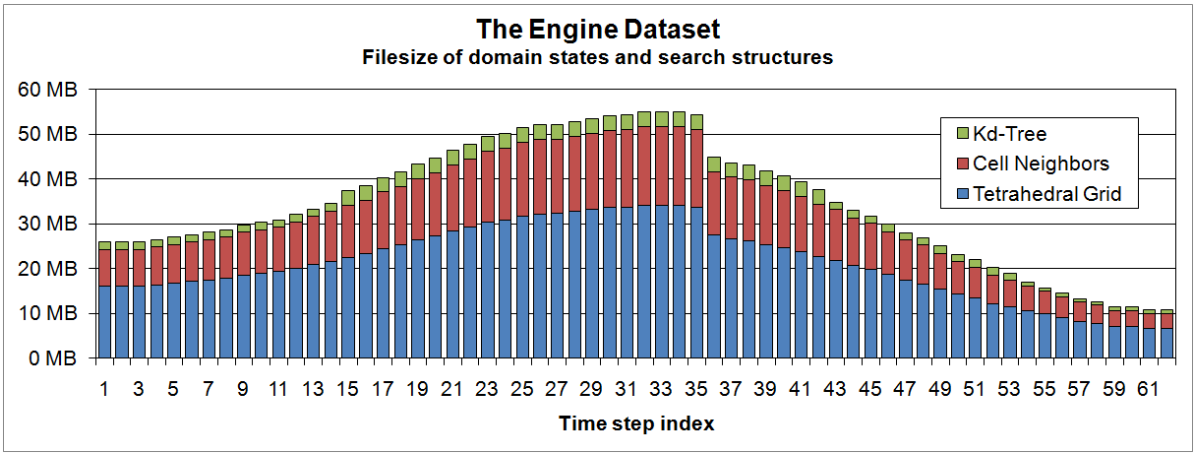


Figure 6.1: The Engine dataset consists of 62 single states. Each state is discretized by a tetrahedral grid with additional overhead introduced by the search structure.

6.2 The Real-World Engine Dataset

The methods presented in this thesis consider time-variant domains with different temporal states, where each state in time is discretized by an unstructured tetrahedral grid. To test the methods on real-world simulations, the *Engine* dataset [Abd98] was chosen, which contains the simulation results for the intake and compression strokes of a four-stroke internal combustion engine, courtesy of the Institute of Aerodynamics (AIA) at RWTH Aachen University (cp. also Figure 2.3 on page 9). This dataset was originally simulated using a multi-block grid and later converted into a series of tetrahedral grids.

The Engine dataset describes a domain which changes its appearance over time. It is defined by 62 single domain states, whereas each state is given by a tetrahedral grid and embedded flow field velocities, defined at the nodes of the grid. All temporal states of the Engine dataset together describe a time-variant tetrahedral grid with an embedded unsteady flow field. Figure 6.1 shows the file size of each domain state as well as the additional amount of memory required per state, to store the search structures needed for the point location within the unstructured grid. With approx. 32%, most of the overhead is added by the indices of the cell neighbors, which are needed to navigate through the grid during the tetrahedral walk. The kd-tree index structure needed to perform the global search on the GPU on the other hand requires only approx. 6.5% of additional memory per domain state.

The first half of domain states of the Engine dataset simulates the intake stroke, in which the size of the combustion chamber increases from state to state, as the piston moves down. Also simulated in this phase is the movement of two engine's valves, which unbolt to induct the combustible mixture and then close again to facilitate the

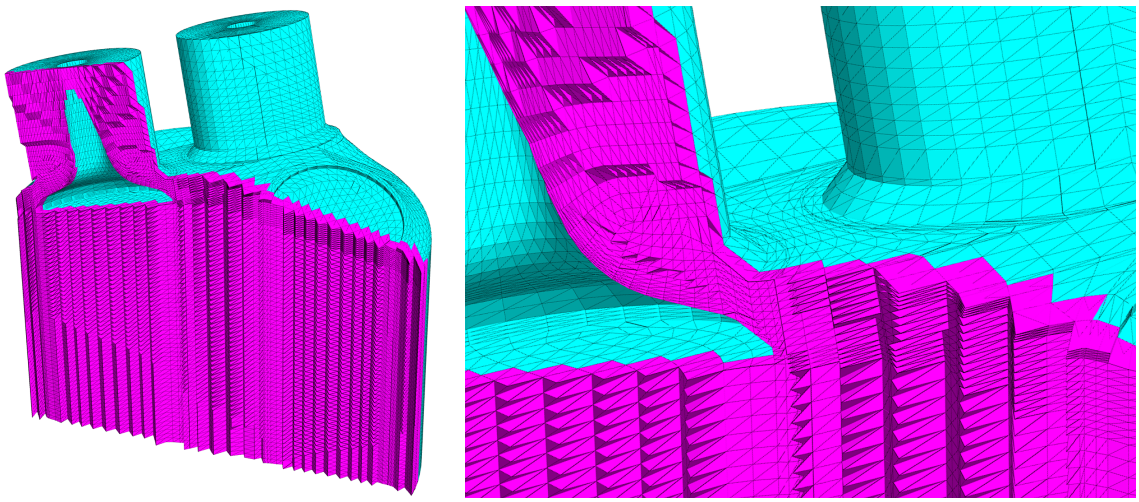


Figure 6.2: A diagonal cut through one of the engine's states shows the different tetrahedron cell sizes used for the domain discretization. Especially small cells were used to simulate the injection flow around the opening and closing valves.

compression stroke. While the valves are opened, the downstroke of the piston reduces the pressure inside the cylinder, which leads to a highly turbulent flow created by the combustible mixture rushing into the combustion chamber.

The compression stroke of the internal combustion engine is simulated by the second half of domain states. In this phase of the four-stroke cycle, the piston moves up again, compressing the injected fuel-air mixture, which reduces the size of the combustion chamber in each new state. The valves are no longer simulated in this phase, as they are closed anyway while the compression stroke, by which the amount of nodes and cells required for the discretization is additionally reduces.

The domain of the Engine dataset is discretized using several unstructured tetrahedral grids with highly varying cell sizes. Figure 6.2 shows a diagonal cut through one of the engine's states as well as a detailed view of the section around the valves. While the distribution of the grid nodes inside the combustion chamber is mostly uniform, leading to homogeneous tetrahedron cell sizes, small cells are used to discretize the area around the moving valves, as strong flow field velocities were calculated by the simulation within this section and the position of the valves dynamically changes.

The number of grid nodes and tetrahedron cells changes with the growing and shrinking of the combustion chamber. At its largest extension, the domain is discretized using 201K grid nodes and 1.15M cells, whereas the first domain state is given by 97K nodes and 540K cells. With the changing number of cells in each state, also the indices of the cells change. Hence, also the indices of surrounding cells of positions within the domain change whenever a new domain state is reached, which makes it necessary to locate the cells of those positions within the new grid.

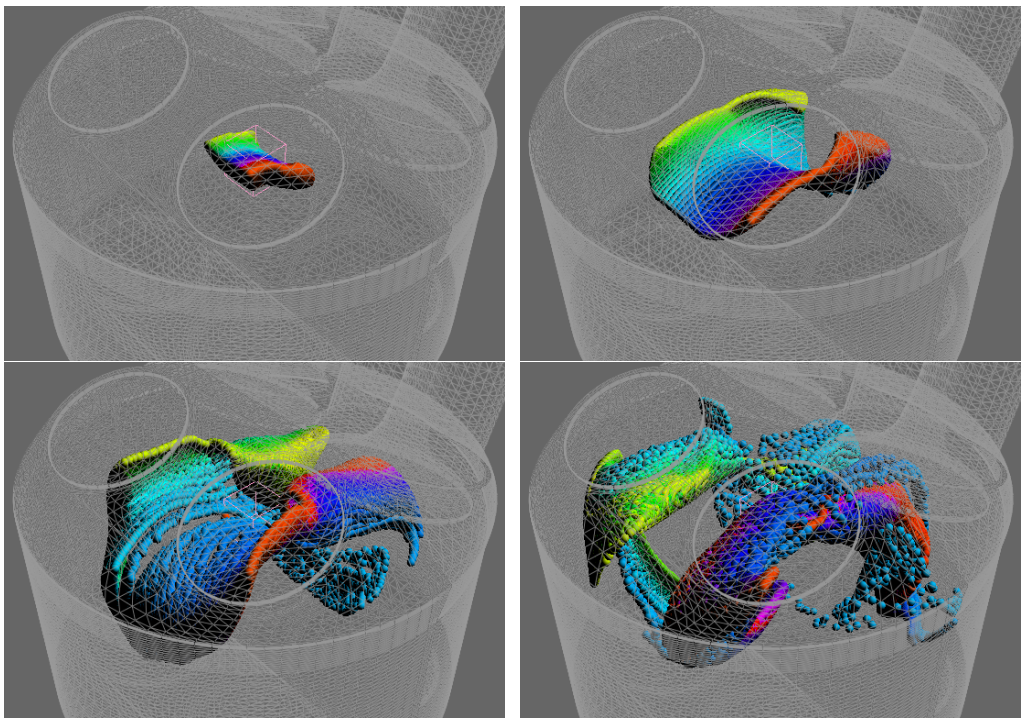


Figure 6.3: The interactive exploration benchmark is performed by creating different particle populations within the domain, which are tracked over several advection steps.

6.3 Interactive Exploration Benchmark

The approach of particle tracing in time-variant tetrahedral grids presented in this thesis is suitable for dynamically changing domains with different cell indices in each state. This was achieved by performing the point location entirely on the GPU, which allows updating the cell indices of a huge particle population within very short time, due to the parallel computation capabilities of present graphics devices, which performs many point-location queries concurrently.

In order to evaluate the usability of the presented flow field visualization approach for interactive exploration of the real-world Engine dataset, the performance was benchmarked for different amounts of particles. A good usability is hereby achieved at interactive rendering frame rates by simultaneously tracing of a vast amount of particles through the flow field.

The benchmark was performed by creating different particle populations within the engine's combustion chamber using a cubic particle seeding strategy. As depicted in Figure 6.3, the particles are tracked over several advection steps in order to visualize the turbulent flow field inside this part of the domain.

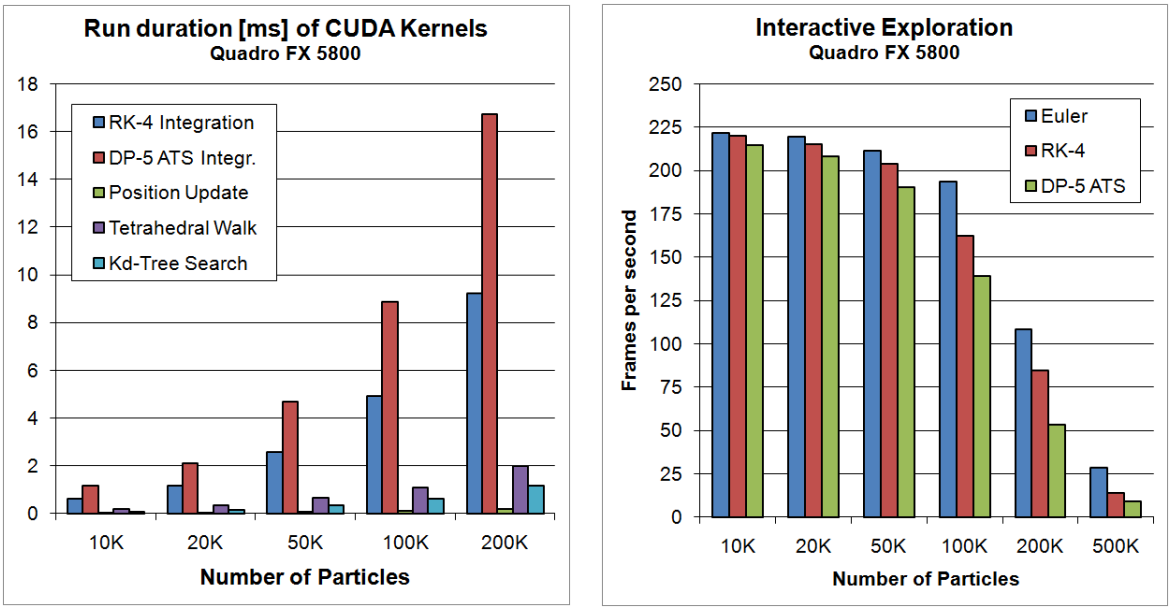


Figure 6.4: The Quadro FX 5800 GPU gives very high frame rates while tracing different populations up to 500K particles through the flow field. The run duration of the CUDA Kernels hereby scales with the number of particles.

The procedure was repeated several times for up to 500K particles while recording the average frame rate in each iteration. The positions of the particles were hereby updated 15 times per second, which yields a smooth animation of the particles' movement. Also, the average run duration of the different computing kernels was recorded using the CUDA Visual Profiler [NVI10d]. The results of the benchmark are presented in Figure 6.4 for the Quadro FX 5800 and Figure 6.5 for the GeForce GT 240. The peak frame rate hereby does not exceed a certain limit, as a specific developer driver is required for CUDA-based calculations, which is optimized for fast calculations rather than high frame rates.

While the flow field integration and the position update are executed many times for each time step, the location of the entire particle population is only performed whenever a new temporal state of the domain is reached. The average run duration of the search kernels hereby depends only on the amount of particles that needs to be located within the new tetrahedral grid and is independent from the chosen integration scheme. As the profiling shows, the GPU-based point-location scheme presented in this thesis, which utilizes a kd-tree structure for the proposed Single-Pass tree traversal method in the global search phase, is capable of updating the cell indices of a huge particle population within in an instant. Even on the middle-class Geforce GT240, the location of a population of 200,000 particles within the tetrahedral grid took only about 8 milliseconds in total. In average, the most computing time is spent on the flow field integration within the tetrahedral grid, especially when using the highly accurate Dopri-5 adaptive step size adjustment procedure

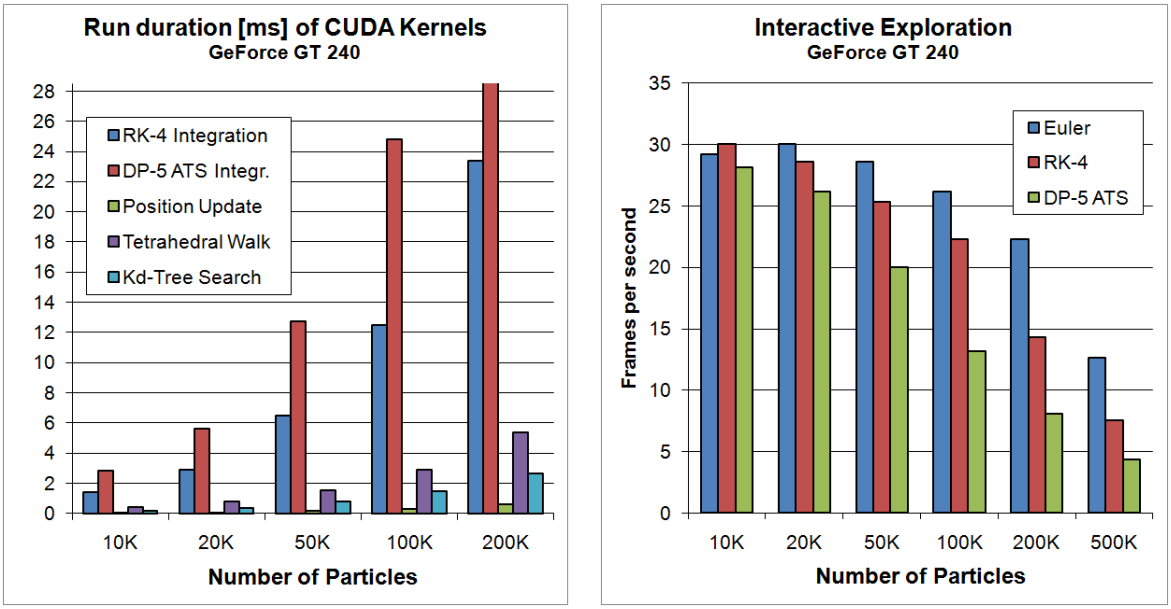


Figure 6.5: Although the GeForce GT240 only has 12 CUDA cores, the benchmark still reaches interactive frame rates for up to 100K particles. The Dopri-5 integration with adaptive time stepping of 200K particles tops-off at 47 msec.

Despite the computations concerning the continuous particle movement, some time is also required to transfer the time step data to the GPU. The relative GPU usage of this data transfer compared to the other computations is shown in Figure 6.6 for the tracing of different particle populations over 400 advection steps and 8 different domain states. A synchronous host-to-device transfer is hereby performed at the beginning of the visualization process on order to copy the data of the first two time steps, while the succeeding time steps are transferred asynchronously. As can be seen, the major workload is due to the flow field integration process which introduces several computationally expensive velocity evaluations in both current tetrahedral grids.

While the described benchmark creates particles only once at the beginning of the advection process, the continuous seeding of particles requires little more effort, as the seeded particles need to be located within both current time steps. While the benchmarked GPUs support the execution of only a single computing kernel at once, the newest generation of Fermi-based NVIDIA GPUs is capable of executing several kernels concurrently using different computing streams [NVI10b]. Therefore, the point-location of newly created particles could be performed simultaneously to the calculations introduced by the particle advection process.

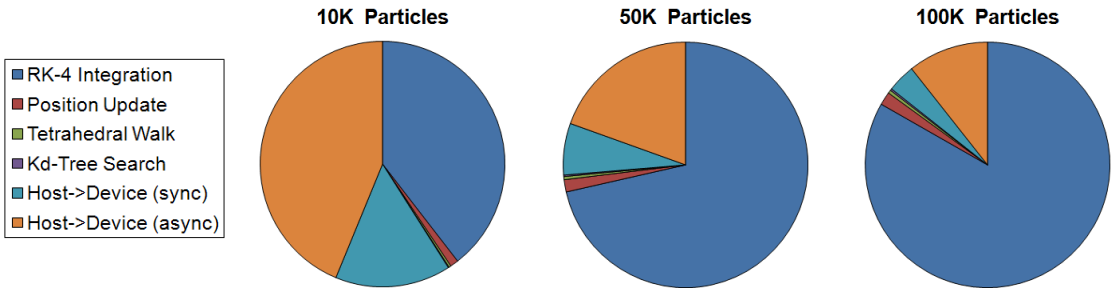


Figure 6.6: With an increasing number of particles, the major workload of the particle advection process results from the flow field integration, while the time needed to transfer the time step data to the GPU remains constant.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Summary and Conclusion

This thesis presented an efficient method for the tracing of massless particles through an unsteady flow field embedded in several tetrahedral grids describing the progression of a time-variant domain, in order to provide an adequate visualization of the original or decimated flow field data resulting from real-world simulations. The major workload of the particle advection process was shifted to the GPU by using the CUDA framework in order to exploit the massive parallel computation capabilities of present programmable graphics hardware.

The nature of the regarded flow field domains requires with the progression of visualization time the repeated location of particle data within consecutive temporal domain states. The efficient two-phase point-location scheme presented in this thesis is entirely performed on the GPU, including the traversal of a kd-tree in the broad phase. As experiments have shown, it is hereby sufficient to traverse the tree only once and additional passes result in only small lesser effort in the narrow phase.

For highest demands on the accuracy of the particle movement, an adaptive step size adjustment procedure was presented, which utilizes the embedded Dopri-5 integration scheme. The accuracy of the integration can be intuitively controlled by an adjustable error tolerance value for the fourth-order accurate solution of the flow field integral.

By utilizing the ViSTA VR Toolkit for the visual depiction of the particles, the time-variant flow can be interactively explored within an immersive Virtual Reality environ-

ment, which also provides an intuitively to handle user interface. Based on the benchmark results of the real-world Engine dataset, up to 500,000 particles may be seeded and advected at interactive frame rates using professional graphics hardware, which should be enough to fill up even large scaled flow field domains.

7.2 Future Work

As a perspective for future work, the visual appearance of the particles may be enhanced in order to achieve a depiction of the flow based on the model of smoke in real-world experiments. This would require additional computational effort, as the particles need to be depth-sorted, while an appropriate smoke-like visual appearance requires also sophisticated calculations for the self-shadowing of the particle cloud.

The major workload of the particle advection process results from the integration of the time-variant flow field, as the integration requires several flow field evaluations, which are calculated by interpolating the velocities defined at the nodes of the grid. For larger particle populations, almost the entire computing time is spent on this task, as profiling of the advection process has shown. The calculations may be enhanced by pre-calculating and storing the per-cell matrices used for the calculation of the natural coordinates which results in additional memory requirements on the graphics device. As the CUDA framework allows distributing thekernel-based computations among several CUDA-enabled devices, a larger improvement of integration performance may be achieved by utilizing additional graphics hardware.

Based on the evaluation of the GPU-driven flow field integration process, it should be possible to generate streak surfaces for the time-variant flow at interactive frame rates, which would yield an additional visual impression of the flow behavior over time but would also require additional considerations on the generation of the surface geometry.

The small amount of graphics memory compared to present host systems allows the examination of only those datasets that fit the limited memory capacity. For larger datasets, a demand-driven data-reduction procedure is required in order to decimate the given dataset to a manageable size, which may also include a region-of-interest specified by the user.

BIBLIOGRAPHY

[Abd98] Aschaf Abdelfattah. *Numerische Simulation von Strömungen in 2- und 4-Ventil-Motoren*. PhD thesis, RWTH Aachen University, 1998.

[Ben90] Jon Louis Bentley. K-d trees for semidynamic point sets. In *SCG '90: Proceedings of the sixth annual symposium on Computational geometry*, pages 187–197, New York, NY, USA, 1990. ACM.

[BFTW09] K. Buerger, F. Ferstl, H. Theisel, and R. Westermann. Interactive streak surface visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, pages 1259–1266, 2009.

[BPSS02] Dirk Bauer, Ronald Peikert, Mie Sato, and Mirjam Sick. A case study in selective visualization of unsteady 3d flow. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 525–528, Washington, DC, USA, 2002. IEEE Computer Society.

[Bun88] Pieter G. Buning. Sources of error in the graphical analysis of cfd results. *J. Sci. Comput.*, 3(2):149–164, 1988.

[But87] J.C. Butcher. *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience New York, NY, USA, 1987.

[CM02] P. Chopra and J. Meyer. Tetfusion: an algorithm for rapid tetrahedral mesh simplification. *IEEE Visualization, 2002. VIS 2002*, pages 133–140, 2002.

[Dev98] Nicolas Devillard. Fast median search: an ansi c implementation. Website, 1998. http://ndevilla.free.fr/median/median.pdf.

[DP80] J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980.

[Ele10] Elemental Technologies Inc. badaboom media converter. Website, 2010. http://www.badaboomit.com.

[Feh70] E. Fehlberg. Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme. *Computing*, 6(1):61–71, 1970.

[GLT+06] Christoph Garth, Robert S. Laramee, Xaviar Tricoche, Jürgen Schneider, and Hans Hagen. Extraction and visualization of swirl and tumble motion from engine simulation data. In *The Topology-Based Methods in Visualization Workshop, 2006*, 2006.

[GTSS04] C. Garth, X. Tricoche, T. Salzbrunn, and G. Scheuermann. Surface techniques for vortex visualization. In *Eurographics - IEEE TCVG Symposium on Visualization*, May 2004.

[KGJ09] H. Krishnan, C. Garth, and K.I. Joy. Time and streak surfaces for flow visualization in large time-varying data sets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1267–1274, 2009.

[Khr10a] Khronos Group. Opencl - introduction and overview. Website, June 2010. http://www.khronos.org/opencl.

[Khr10b] Khronos Group. The opengl graphics library. Website, 2010. http://www.opengl.org.

[Kit10] Kitware Inc. *The VTK User's Guide*. Kitware Inc., 11th edition, March 2010.

[KKKW05] Jens Kruger, Peter Kipfer, Polina Kondratieva, and Rudiger Westermann. A particle system for interactive visualization of 3d flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[KL95] David N. Kenwright and David A. Lane. Optimization of time-dependent particle tracing using tetrahedral decomposition. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 321, Washington, DC, USA, 1995. IEEE Computer Society.

[KL96] David N. Kenwright and David A. Lane. Interactive time-dependent particle tracing using tetrahedral decomposition. *IEEE Trans. Vis. Comput. Graph.*, 2(2):120–129, 1996.

[KM92] David N. Kenwright and G. D. Mallison. A 3-d streamline tracking algorithm using dual stream functions. In *IEEE Visualization*, pages 62–69, 1992.

[KRG03] Peter Kipfer, Frank Reck, and Günther Greiner. Local exact particle tracing on unstructured grids. *Computer Graphics Forum*, 22(4):133–142, 2003.

[Lan93] David A. Lane. Visualization of time-dependent flow fields. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 32–38, Washington, DC, USA, 1993. IEEE Computer Society.

[LGSH06] Robert S. Laramee, Christoph Garth, Jürgen Schneider, and Helwig Hauser. Texture advection on stream surfaces: A novel hybrid visualization applied to cfd simulation results. In *EuroVis*, pages 155–162, 2006.

[LST03] Max Langbein, Gerik Scheuermann, and Xavier Tricoche. An efficient point location method for visualization in large unstructured grids. In *VMV*, pages 27–35, 2003.

[LW77] D.T. Lee and C.K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[Mel09] J.M. Melenk. Numerik von gewöhnlichen Differentialgleichungen. Website, 2009. `http://www.math.tuwien.ac.at/~melenk/`.

[Mic10] Microsoft Corp. Directx software development kit. Website, June 2010. `http://www.microsoft.com/directx`.

[MLP+09] Tony McLouglin, Robert S. Laramee, Ronald Peikert, Frits H. Post, and Min Chen. Over two decades of integration-based geometric flow visualization. *Eurographics 2009, State of the Art Report*, pages 73–92, 2009.

[NJ99] Gregory M. Nielson and Il-Hong Jung. Tools for computing tangent curves for linearly varying vector fields over tetrahedral domains. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):360–372, 1999.

[NVI10a] NVIDIA Corp. *CUDA Programming Guide, Version 3.0*, February 2010.

[NVI10b] NVIDIA Corp. *CUDA Programming Guide, Version 3.1*, May 2010.

[NVI10c] NVIDIA Corp. Physx by nvidia. Website, June 2010. http://www.nvidia.com/physx.

[NVI10d] NVIDIA Corp. The CUDA Toolkit, version 3.0. Website, 2010. http://www.nvidia.com/cuda.

[Pag05] Pagani Automobili S.p.A. Pagani Zonda F - Wind Tunnel. Website, 2005. http://www.paganiautomobili.it.

[Pan08] Rina Panigrahy. An improved algorithm finding nearest neighbor using kd-trees. In *LATIN'08: Proceedings of the 8th Latin American conference on Theoretical informatics*, pages 387–398, Berlin, Heidelberg, 2008. Springer-Verlag.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing, 3rd Edition*. Cambridge University Press, September 2007.

[PVH+03] F.H. Post, B. Vrolijk, H. Hauser, R.S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. In *Computer Graphics Forum*, volume 22, pages 775–792. John Wiley & Sons, 2003.

[Ros09] Randi J. Rost. *OpenGL Shading Language (3rd Edition)*. Addison-Wesley Professional, July 2009.

[Sch08] Marc Schirski. *Interactive Particle Tracing for the Exploration of Flow Fields in Virtual Environments*. PhD thesis, RWTH Aachen University, 2008.

[SL00] Alexander J. Smits and T. T. Lim. *Flow Visualization: Techniques and Examples*. Imperial College Press, 2000.

[SL05] J.P. Shen and M.H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. McGraw-Hill, 2005.

[USM96] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Efficient streamline, streamribbon, and streamtube constructions on unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):100–110, 1996.

[VDBO97] J.W. Van Der Burg and B. Oskam. FASTFLO-automatic CFD system for three-dimensional flow simulations. In *The Conference on Industrial*

Technologies and 3rd Aero Days in Toulouse, France. National Aerospace Laboratory NLR, Amsterdam, The Netherlands, October 1997.

[VFWTS08] W. Von Funck, T. Weinkauf, H. Theisel, and H.P. Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE transactions on visualization and computer graphics*, pages 1396–1403, 2008.

[VGVW99] A. Van Gelder, V. Verma, and J. Wilhelms. Volume decimation of irregular tetrahedral grids. In *Computer Graphics International*, pages 222–230. Citeseer, 1999.

[Vir09] Virtual Reality Group, Center for Computing and Communication, RWTH Aachen. The vista virtual reality toolkit. Website, 2009. `http://www.rz.rwth-aachen.de/ca/c/piz/`.

[WIA09] WIAS, The Weierstrass Institute for Applied Analysis and Stochastics. Tetgen: A quality tetrahedral mesh generator and a 3d delaunay triangulator. Website, 2009. `http://tetgen.berlios.de`.

[Wik09] Wikipedia. Description of regular and unstructured grids. Website, 2009. `http://www.wikipedia.org`.

[WMKE04] Manfred Weiler, Paula N. Mallon, Martin Kraus, and Thomas Ertl. Texture-encoded tetrahedral strips. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, pages 71–78, Washington, DC, USA, 2004. IEEE Computer Society.