UNIVERSITÄT SIEGEN

Computergraphik und Multimediasysteme

Real-Time Particle Systems
and their Application to Flow Visualization

Echtzeit-Partikelsysteme
und deren Anwendung auf Strömungsvisualisierung

vom Fachbereich Elektrotechnik und Informatik
der Universität Siegen

zur Erlangung des akademischen Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)

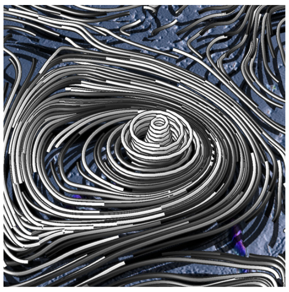
genehmigte Dissertation

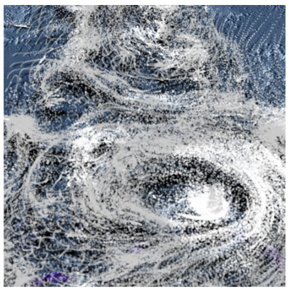
von

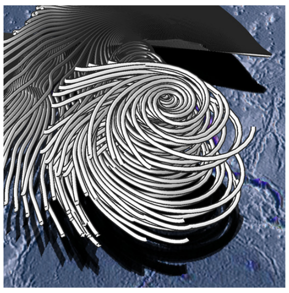
Nicolas Cuntz
Siegen, Mai 2009

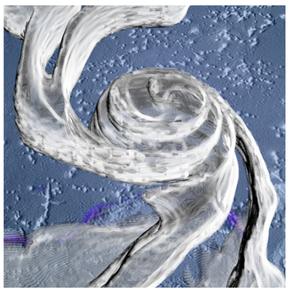
1. Gutachter:	Prof. Dr. Andreas Kolb
2. Gutachter:	Prof. Dr. Daniel Weiskopf

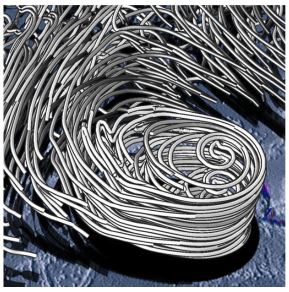
Tag der mündlichen Prüfung: 28. September 2009

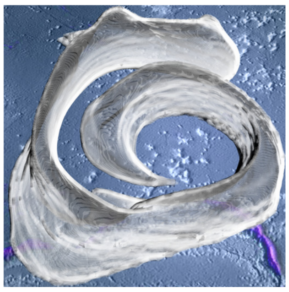

stream lines


flow particles


path lines


time surface


streak lines


streak volume

Visualization of a typhoon flow
(data set: courtesy of MPI for Meteorology)

Acknowledgments

First of all, I would like to thank Prof. Dr. Andreas Kolb for his long-time support and countless advices during my time as a PhD student at the University of Siegen. Being my supervisor, he has shown me the scientific way to interpret and solve problems.

I thank the co-authors of all the papers and articles I worked on during my stay in Siegen: Dr. Robert Strzodka introduced me to particle level sets, and he gave me useful hints to adapt them for graphics hardware. I am very grateful to Prof. Dr. Weiskopf for accepting the co-advisorship of my PhD thesis and for helpful discussions related to the flow visualization aspects of my work.

I would like to acknowledge the German Climate Computing Centre Hamburg for providing an interesting typhoon data set. Special thanks go to Michael Böttinger, who was our primary contact person. His knowledge of climate flow visualization aspects has been of great value.

I also thank my colleagues, especially Marvin Lindner for his friendliness, his encouragement and for extensive discussions about every imaginable C++ language feature.

Finally, the largest thanks belong to my parents, my brothers, my dear girlfriend, and my former schoolmates and fellow students, who all contributed to what I am now in their very own way.

Nicolas, Siegen
May, 2009

Abstract

Both visualization and simulation tasks make high demands on accuracy and interactivity. The former is an evident requisite of any tool focusing on quality, the latter stands for a responsive system in which the user has real-time control. A permanent trade-off between both demands can be observed, since high accuracy is usually time-consuming. The graphics processing unit (GPU), which evolved to a powerful general purpose coprocessor during the last decade, is undoubtedly becoming an ideal instrument for performing visualization and simulation tasks accurately *and* in real-time. One of the main challenges in the context of geometric flow visualization and fluid dynamics is the representation of motion. In both fields, particle systems and grid-based structures constitute basic models describing the objects to which the motion is applied.

This work is dedicated to the development of GPU-based particle techniques and their combination with grids in order to improve the efficiency of fluid simulation and geometric flow visualization techniques. All resulting algorithms are characterized by their parallel nature and by being entirely executable on graphics hardware.

The first aim is to provide an efficient solution to particle coupling in the context of fluids based on smoothed particle hydrodynamics. A parallel processing using graphics hardware is achieved by providing a grid-based mechanism for the efficient processing of particle neighborhoods.

The second aim is to provide a set of algorithms for various geometric flow visualization techniques including flow particles, flow lines and flow surfaces/volumes. The accuracy is explicitly addressed in all cases. Accordingly, a method for generating time-adaptive stream and path lines and a special refinement scheme for streak lines is presented. The flow volumes, including time surfaces, path and streak volumes, are based on a parallel reinterpretation of the particle level set (PLS) method. This reinterpretation is based on a method for grid-particle interchange which is similar to the one proposed for particle coupling. Combining a grid and a particle set takes the advantages from both models: the grid representation is robust w.r.t. deformations and topological changes, and the particles are used to reduce numerical diffusion. For the reinitialization of the level set function, which is the most time-consuming step of the PLS algorithm, a hierarchical method for computing distance transforms is proposed. It turns out that the use of a distance transform is advantageous for realizing a GPU-based PLS framework.

All presented flow visualization techniques are applied to a real-world example in the context of climate research. The evaluation shows that the interactive system facilitates the efficient exploration of complex flow data sets.

Categories and Subject Descriptors (according to ACM CCS):
I.3.1 [Computer Graphics]: Parallel processing
I.3.5 [Computer Graphics]: Curve, surface, solid, and object representations
I.3.3 [Computer Graphics]: Line and curve generation
I.3.7 [Computer Graphics]: Virtual reality

Zusammenfassung

Sowohl Visualisierungen als auch Simulationen setzen hohe Ansprüche an Genauigkeit und Interaktivität. Erstere ist eine offensichtliche Voraussetzung für Werkzeuge, bei denen die Qualität im Vordergrund steht, letztere steht für ein System, das in Echtzeit auf Benutzereingaben reagiert. Beide Aspekte liegen in permanenter Konkurrenz zueinander, denn eine hohe Genauigkeit ist i.A. zeitaufwändig. Moderne Graphikprozessoren (engl. *Graphics Processing Units* bzw. kurz GPU's) haben sich im Laufe des letzten Jahrzehnts zu universal einsatzfähigen Recheneinheiten weiterentwickelt. Die GPU als Coprozessor ist unzweifelhaft ein ideales Werkzeug geworden, um Visualisierungs- und Simulationsaufgaben in hoher Genauigkeit *und* in Echtzeit zu lösen. Eine der größten Herausforderungen im Kontext geometrischer Strömungsvisualisierung und der Simulation von Fluiden ist die Repräsentation von Bewegungen und Verformungen. In beiden Disziplinen bilden Partikelsysteme und gitterbasierte Strukturen grundlegende Modelle zur Beschreibung der veränderlichen Objekte.

Diese Arbeit widmet sich der Weiterentwicklung GPU-basierter Partikelsysteme und deren Kombination mit Gitterstrukturen, um die Effizienz von Techniken in den beiden zuvor genannten Disziplinen zu verbessern. Alle resultierenden Algorithmen sind gekennzeichnet durch Ihren parallelen Aufbau, und dadurch, dass sie vollständig auf Graphik-Hardware ausführbar sind. Die erste Zielsetzung besteht darin, eine effiziente Partikelkopplung im Kontext von Fluiden zu ermöglichen, die auf den SPH-Ansatz (*Smoothed Particle Hydrodynamics*) basieren. Eine parallele Verarbeitung mittels der Graphik-Hardware wird erreicht durch einen Mechanismus zur effizienten gitterbasierten Verarbeitung von Partikelnachbarschaften. Die zweite Zielsetzung besteht darin, einen Satz von Algorithmen für unterschiedliche geometrische Strömungsvisualisierungstechniken bereitzustellen, welche die folgenden Primitive nutzen: Partikel, Linien, Oberflächen und Volumina. Die Genauigkeit wird in allen Fällen explizit thematisiert. Dementsprechend wird eine Methode zur Generierung zeitadaptiver Stream- und Path-Lines und ein Schema zur adaptiven Verfeinerung von Streak-Lines präsentiert. Die dritte Kategorie von Primitiven besteht aus Time-Surfaces, Path- und Streak-Volumes. Diese werden aufgrund einer Neuinterpretation der PLS-Methode (*Particle Level Set*) beschrieben. Diese Neuinterpretation basiert auf einem Schema zum Gitter-Partikel-Austausch, das dem zuvor für die Partikelkopplung vorgeschlagenen ähnelt. Die Kombination beider Strukturen, der Partikelmenge und des Gitters, vereinigt die Vorteile beider Modelle: Die Gitterrepräsentation ist robust gegenüber Deformationen und topologischen Veränderungen, und die Partikel können verwendet werden, um die numerische Diffusion zu verringern. Für die Reinitialisierung der Level-Set-Funktion, welche den aufwändigsten Schritt im PLS-Algorithmus bildet, wird eine hierarchische Methode zur Berechnung eines Distance-Transforms vorgeschlagen. Es stellt sich heraus, dass die Nutzung eines Distance-Transforms vorteilhaft bei der Realisierung eines GPU-basierten PLS-Frameworks.

Alle präsentierten Strömungsvisualisierungstechniken werden auf ein reales Beispiel im Kontext der Klimaforschung angewandt. Die Evaluation zeigt, dass das interaktive System die effiziente Exploration von komplexen Strömungsdatensätzen erleichtert.

Contents

Prologue

Computer graphics defines a *particle system* as a set of points in space, each moving along a path which is determined by an appropriate routine, usually involving the particle's position and velocity. Typically, the particles emerge (*birth*) and disappear (*death*) after specific periods of time. The history of particle systems goes back to the early eighties of the 20th century, when William T. Reeves presented a particle-based effect forming a wall of fire moving over a planet's surface in the movie *Star Trek II* [Ree83]. Since then, particle systems have become a popular tool for graphical effects in countless video games and motion pictures.

Besides these applications concentrating on the animation and an appealing rendering, particle systems have a far larger impact on graphics when thinking of them as an abstract concept: The key idea of tracing discrete particle trajectories reflects the very basic principles of motion, as they originated from classical mechanics. And, from a different point of view, a particle system can be seen as a discrete, punctual representation of any cloudy object, which is moved particle-wise.

Spatial discretizations are omnipresent in computational sciences, especially in graphics. Besides particle systems, *grids* constitute another way of representing spatial entities. In contrast to particles, grids rely on a well-defined arrangement of the discrete locations in space. Grids and particles are commonly used for simulations (e.g. fluid dynamics), visualizations (*voxel grids*) and for many other important tasks.

In fluid dynamics, the distinction between grids and particle systems is reflected in two different paradigms: *Lagrangian* and *Eulerian* solvers for the Navier-Stokes equations. The first type, named after the Italian mathematician Joseph-Louis Lagrange, corresponds to the idea of following individual fluid particles. The second

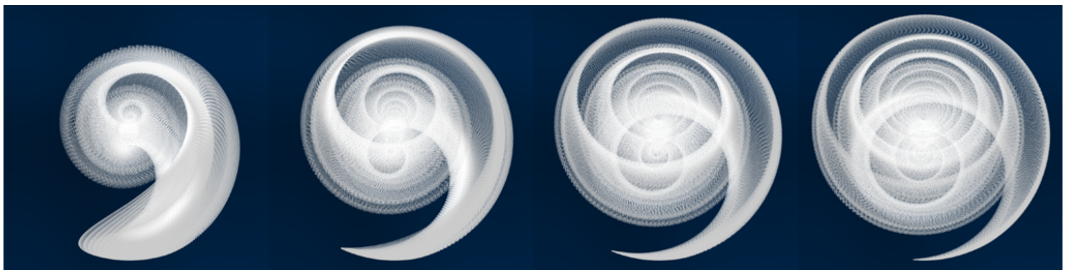


Figure 1: *One million particles in a vortex flow. The initial particle configuration was chosen to form a spherical shape.*

type, named after the Swiss mathematician Leonhard Euler, solves the fluid equations within an inertial reference frame, i.e. a grid covering the volume in which the fluid acts.

In flow visualization, both grids and particles are fundamental in the sense that the majority of systems rely on these two concepts: When thinking of textures as grids, dense representations, e.g. systems using line integral convolution, make heavy use of grids. Particles on the other side form an intuitive way of visualizing a flow, because throwing particles into airflow is very natural. Rendering those particles provides a simple yet powerful visualization of the flow. As we will see later, the spectrum in which grids and particles can be utilized in flow visualization is much wider than that.

An interesting aspect when thinking about grids and particles lies in their strengths and weaknesses when representing volumetric objects in motion: Grid-based approaches, e.g. based on level sets, are largely insusceptible to topological changes like a disruption of two parts of an object. More specifically, no holes can arise in the representation. Thus, a grid cell's neighborhood can easily be accessed by looking for the surrounding cells. A particle set, on the other side, is unstructured per se and may suffer from unspecified regions when particles split into different directions. However, moving particles is very accurate due to high-order integration schemes, while grid advection typically involves a larger numerical error. This comparison is interesting because it is possible to combine both paradigms in order to unify the advantages.

Considering the importance of grids and particles, it is clear that fast algorithms supporting these structures are essential. Due to the separable nature of *per grid cell* or *per particle* processing, both structures are well-suited for the use on parallel architectures. Today, powerful parallel processors can be found on any modern consumer graphics hardware: Modern graphics processing units (GPUs) are designed to provide both a highly optimized traditional graphics pipeline *and* a flexible general purpose toolkit. The high degree of parallelism and flexibility makes the GPU a suitable instrument for solving visualization and simulation tasks.

My Contribution

The scope of this dissertation covers two major topics: the simulation of particle fluids and geometric flow visualization. I have worked on the following thesis:

> *The concept of GPU-based particle systems and their combination with grid structures can be developed to improve the efficiency of fluid simulation and geometric flow visualization techniques.*

I will support this thesis by presenting GPU-based techniques designed for 1. the fast simulation of fluids according to the smoothed particle hydrodynamics method, 2. a real-time deformable surface tracing method reformulating the particle level set method for parallel processing, and an accurate and interactive flow visualization using 3. flow particles, 4. their extension to stream, path and streak lines, and 5. time surfaces, path and streak volumes.

The Objectives

This work is aiming for general objectives which are specified in the following. Together, these thoughts form the motivation for the contributed techniques.

The first objective is to provide interactive techniques in the context of fluids and flow visualization while ensuring high quality results. Consequently, fast algorithms and their adaption to the parallel architecture of modern graphics hardware is one of the main motifs which will lead to a set of accurate real-time simulation and visualization techniques.

The second objective is to improve volumetric representations by combining GPU-based particles and grid structures. This idea appears in two different flavours, one concerning fluids, the other concerning freely deformable objects. The former is characterized by a particle representation supported by a grid, the latter by a grid representation supported by a particle set which is used to improve the accuracy. The ambition is to show that a hybrid processing and the information interchange between both structures can be efficiently realized using graphics hardware.

The third objective is the investigation of GPU-based particle systems as a basic concept for flow visualization. This includes a purely particle-based framework and its extension to more complex flow primitives. One subset of these primitives consists of different types of flow lines which are constructed on a per-particle base. Another subset is the accurate tracing of flow volumes.

The last objective is to bring the presented flow visualization techniques into the context of climate visualization, which is done by applying them to a practical example. The intention is to provide a very basic yet powerful set of techniques for interactively exploring such data.

The Techniques

In the following, the techniques which together form the contribution of this dissertation are introduced briefly. In the schematic overview given in Fig. 0.2, the most important keywords are brought together and classified into the fields *graphics*, *simulation* and *visualization*. This overview also illustrates the connections between the keywords.

Real-time particle fluids – The proposed fluid simulator is based on smoothed particle hydrodynamics (SPH). One of the major challenges when designing a parallel solution to SPH is the search for neighbor particles. This can, in general, be seen as the so-called n-nearest neighbors problem. A solution using a hybrid grid-particle interchange is proposed to solve this problem on the GPU. In order to execute all steps of the SPH algorithm without CPU intervention, the steps must be reorganized in an appropriate way. The presented approach was the first technique introducing GPU-based SPH fluids.

Real-time particle level sets – A GPU-based framework involving a grid-particle interchange is proposed for tracing closed surfaces according to the particle level set (PLS) method. The capability to trace closed surfaces is equivalent to the representation of arbitrarily deformable objects in motion. The challenge posed by the desired accuracy of the tracing is addressed by the PLS approach by minimizing

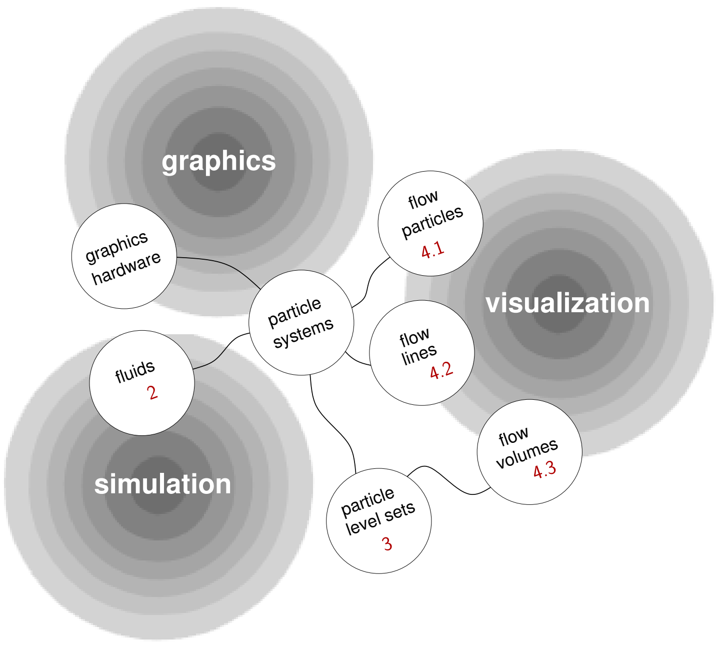


Figure 2: *A diagram showing the context of the thesis. Relations between keywords are visualized by line connectors. The sections corresponding to the key topics are annotated in red.*

the volume loss of the represented object and preserving as many surface details as possible during motion. For the GPU-based framework, the PLS approach is reformulated to take maximal advantage of parallel graphics hardware: All steps, i.e. level set advection, correction, particle reseeding and level set reinitialization are designed to work on the GPU without the need for retrieving any data to the CPU. The level set reinitialization requires special attention as it constitutes a major bottleneck. For this task, a new algorithm for the fast hierarchical computation of grid-based distance transforms is developed. In addition, the use of a distance transform provides an easy and powerful concept for particle reseeding.

Real-time flow visualization – The contribution in the context of flow visualization is divided into three approaches, each related to a specific type of geometric flow primitive. All of these approaches are motivated and evaluated on the basis of an unsteady flow data set coming from the climate research context. The first type of flow primitives forms a purely particle-based flow visualization which is specialized to the requirements of climate flow visualization. Consequently, the proposed method includes shadowing in conjunction with a terrain representation, a very basic non-uniform input data support and an intuitive user interface for interactive exploration. The second type handles the parallel generation of accurate stream, path and streak lines. The corresponding algorithms are designed to generate the line geometries on-the-fly and in real-time, while achieving high accuracy by following an adaptive time-stepping strategy. For streak lines, a refinement scheme ensuring a smooth curve progression is proposed. The major challenge when visualizing real-time flow lines is the synchronization of the geometry construction to unsteady flow data sets. The third visualization approach consists of a specialized PLS method suited for the representation of time surfaces, path volumes and streak volumes. Especially streak volumes require special attention: A model for dye injection must be adapted to the hybrid grid-particle model. The usability of the flow visualization techniques is improved by supporting a virtual reality environment including user tracking devices.

Note that all techniques are designed to run in real-time on graphics hardware. Additionally, all the presented geometric flow visualization techniques explicitly address the necessity of being accurate: The flow particles benefit from high order integration schemes, the flow lines are generated adaptively, and the flow volumes achieve higher accuracy by using corrective particles.

Publications

The following list of publications covers most of the topics I worked on before starting to write my thesis. This material includes most of the techniques contributed in later chapters.

[**KC05**] Dynamic Particle Coupling for GPU-based Fluid Simulation: A fast coupling of particles in the context of fluid solvers based on SPH is achieved by accumulating smoothed quantities in a grid structure using the GPU.

[**CKL*07**] GPU-based Dynamic Flow Visualization for Climate Research Applications: An accurate and fast particle flow framework designed for the visualization of climate phenomena helps climate researchers to interactively explore their flow data sets.

[**CPK09**] Time-Adaptive Lines for the Interactive Visualization of Unsteady Flow Data Sets: A fast and accurate visualization using time-adaptive stream, path and streak lines, which are generated with new fully parallel algorithms, is proposed.

[**CK07**] Fast Hierarchical 3D Distance Transforms on the GPU: A fast hierarchical GPU-based algorithm for computing signed distance transforms improves the scalability between approximation and efficiency.

[**CKSW08**] Particle Level Set Advection for the Interactive Visualization of Unsteady 3D Flow: A fast visualization approach featuring time surfaces, path volumes and streak volumes extends the particle level set method in order to achieve an accurate representation of the volumetric primitive.

Overview

The structure of this document consists of five chapters: The introduction, Chap. 1, discusses the prior work, i.e. the foundation necessary to understand the later parts. The content of this chapter reproduces well-known methods found in the computer graphics literature, thus including a lot of references to related articles, papers and books. The next three chapters contain the actual contribution. Exceptions to this rule, i.e. parts which are based on other author's work within Chap. 2–4, are explicitly referenced and marked appropriately.

The residual structure has been chosen to follow a logical flow from fluid particles over deformable surfaces/objects to geometric flow visualization featuring various geometric approaches including particles, lines, surfaces and volumes. Chap. 2, which discusses a particle fluid approach, provides the technical know-how for hybrid grid-particle approaches. This concept is expanded in Chap. 3 in order to represent deformable surfaces/objects. Chap. 4 then introduces the context of climate flow visualization. Geometric flow visualization techniques based on particles and lines are discussed, then the GPU-based PLS framework from the previous chapter is extended to the task of visualizing flow volumes. Finally, Chap. 5 provides a summarizing overview, and possible improvements and future directions are discussed.

Chapter 1
Introduction

The technology at the leading edge changes so rapidly that you have to keep current after you get out of school. I think probably the most important thing is having good fundamentals.

Gordon Moore

This chapter establishes the foundation necessary to understand the subsequent chapters. It discusses the previous work related to the thesis' main topics, i.e. graphics hardware, particle systems, grids, flow visualization and particle fluids. Readers who are familiar with some or all of these topics might want to skip this chapter and continue in the second chapter.

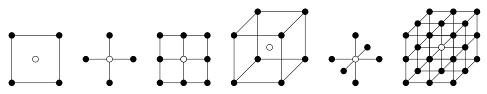
The introduction starts with an explanation of graphics parallelism, as this aspect will be central throughout all chapters. This part furnishes the technical terms necessary for the formulation of algorithms conforming to the parallel concept of graphics hardware. Later parts will, however, contain formal notations as well. Tab. 1.1 provides an overview of the nomenclature and the symbols used later.

1.1 Graphics Hardware

During the last ten years, the development in computer graphics has made major advances, not only on the theoretical side, but also w.r.t. graphics hardware and programming interfaces. Both are influencing each other, as theoretical feasibility motivates the design of new hardware architectures, and, on the other hand, hardware improvements open the door for practically solving more complex problems. Besides, hardware conditions dictate or at least influence the way of thinking about algorithms. More specifically, the specific parallel nature of today's graphics chips implies a careful design of according algorithms and data structures: Researchers in this context must be aware of all concepts and all functionality available in detail.

In the beginning, graphics hardware was very specialized. This was the case for expensive graphics workstations as well as for consumer hardware. The deployed chips supported a highly integrated set of functionality according to a fixed pipeline reflecting the necessity of these days' graphics applications. Consumer boards have evolved to supply the demand for visual effects in computer games, in particular the rendering of 3-dimensional scenes. Since then, the industry has put a lot of effort in improving rendering quality, thus creating one generation of graphics cards after the other.

Table 1.1: *Mathematical symbols and their interpretation. The chosen notation largely conforms with the one used in graphics literature.*

$x,\ \vec{\mathbf{x}},\ \hat{\mathbf{x}},\ \mathbf{x}$	scalar, vector, normalized vector, 2D or 3D point
$(x, y, z)^{\mathrm{T}}$	transposed vector
$\vec{\mathbf{x}}^x, \vec{\mathbf{x}}^y, \vec{\mathbf{x}}^z$	vector component access (similarly for points)
$\vec{\mathbf{x}} \cdot \vec{\mathbf{y}}$	the scalar product of the vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$
$\lVert\vec{\mathbf{x}}\rVert = \sqrt{x_1^2 + \cdots + x_n^2}$	Euclidean norm of $\vec{\mathbf{x}} = (x_1, \ldots, x_n)^{\mathrm{T}}$
$\nabla a = (\frac{\partial a}{\partial x}, \frac{\partial a}{\partial y}, \frac{\partial a}{\partial z})$	the gradient of a scalar field $a(x, y, z)$
$\nabla \cdot \vec{\mathbf{v}} = \frac{\partial \vec{\mathbf{v}}}{\partial x} + \frac{\partial \vec{\mathbf{v}}}{\partial y} + \frac{\partial \vec{\mathbf{v}}}{\partial z}$	the divergence of a vector field $\vec{\mathbf{v}}(x, y, z)$
$\nabla^2 a = \frac{\partial^2 a}{\partial x} + \frac{\partial^2 a}{\partial y} + \frac{\partial^2 a}{\partial z}$	the Laplace of a scalar field $a(x, y, z)$
$f(n) \in \mathcal{O}(g(n))$	the function $f(n)$ is asymptotically bounded by $g(n)$
$\mathcal{M}_k, \mathcal{N}_k$	 filter kernels: $\mathcal{M}_4$ $\mathcal{N}_4$ $\mathcal{N}_8$ $\mathcal{M}_8$ $\mathcal{N}_6$ $\mathcal{N}_{26}$ kernels with central cell: $\mathcal{M}_5, \mathcal{N}_5, \mathcal{N}_9, \mathcal{M}_9, \mathcal{N}_7, \mathcal{N}_{27}$

When contemplating the performance of modern GPUs, in terms of the number of operations per second, one can observe an even faster growth compared to CPUs (see [Moo65] for Moore's famous prediction of the future of integrated electronics). This considerable growth is due to the highly parallel design of the GPU with a high number of computing cores and support for massive threading. This parallelism provides programmers a much higher processing power than the one of CPUs. However, it also constitutes a restriction as not every problem can be formulated to exploit the parallel capabilities of the GPU. Simultaneously to performance advances, each generation of GPUs brought about new features, each widening the previous set of capabilities. During this development, the scope has moved from the ever-growing specialized functionality to a general architecture allowing a wider range of applications. Next to the evolving hardware, powerful programming interfaces, including high-level languages for the use of GPU parallelism, e.g. `Cg` (`C` for graphics – see [MGAK03] for a presentation of `Cg`'s set of features) and `GLSL` (the `OpenGL` shader language), emerged. While these languages still are strongly related to the rendering of 3-dimensional objects, new interfaces like `CUDA` (Compute Unified Device Architecture) become more and more popular for solving problems which are not directly related to graphics. In future, this trend could make the GPU a coprocessor for arbitrary parallel computations which would be an indispensable part of every personal computer.

The algorithms presented in this dissertation are formulated in "shader language", i.e. using the stages of the graphics pipeline. This is convenient, for example in the

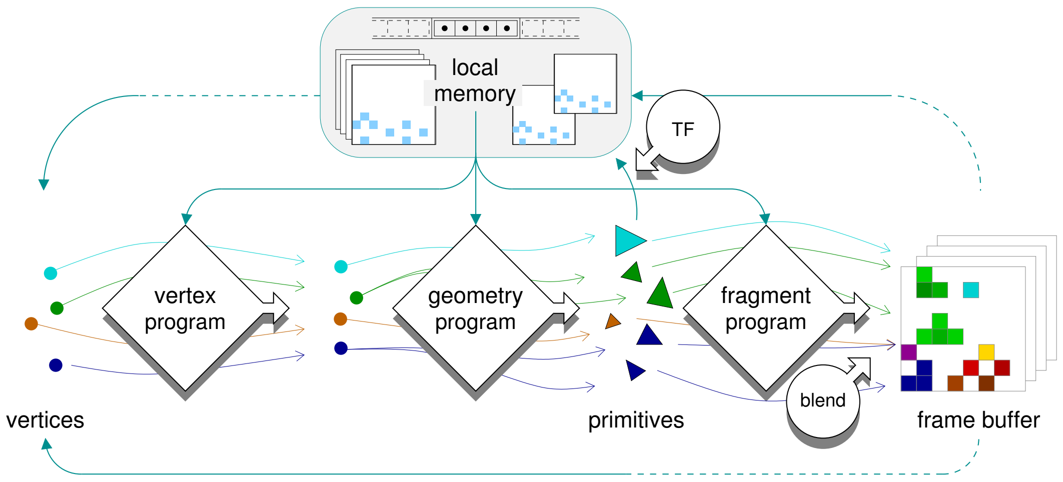


Figure 1.1: *The graphics pipeline. The data flow is symbolized by arrows passing through the three pipeline stages. TF: transform feedback.*

case of the flow line generator, which makes heavy use of the geometry shader: Element insertion and deletion in a geometry program is a feature that has no direct equivalent in `CUDA`. Despite this case where shader APIs seem more powerful, one should keep in mind that in other cases, it surely is reasonable to port the approaches to a general purpose API.

1.1.1 The Modern Graphics Pipeline

Starting from now, the graphics pipeline is presented according to the functionality provided by the *Open Graphics Library* (`OpenGL`), in version 3.0 (plus extensions), which is currently available at the time of writing. The The terminology might slightly differ when switching to another graphics API like `directx`, however, due to the common hardware support, the concepts are equal.

The graphics pipeline is composed of three different stages, realized by the *vertex*, the *geometry* and the *fragment* shader. The arrangement of these stages within the pipeline is sketched in Fig. 1.1. When using the term *shader*, the programmable unit is meant, while the code executed by a shader is called *program* in the following.

Render Passes and Data Access

One render pass is triggered by pushing a set of geometric primitives (i.e. points, lines, triangles, quads or polygons) into the pipeline. These data are processed vertex-wise within the *vertex program*, allowing the programmer to transform the geometry, which is then propagated to the *geometry program*. This second stage is able to reassemble and retransform primitives, while introducing new vertices and removing existing ones. The next stage involves the rasterization of the primitives and a fragment-wise processing within the *fragment program*. Output fragments are written to the frame buffer or to multiple render targets (MRT). As an alternative to fragment processing, the stream of outgoing vertices can be redirected to a so-called transform feedback (TF) buffer. This can be useful when mapping the input vertices to an output set of vertices rather than output fragments. A nice characteristic of the transform feedback is that the output stream remains in the same order as the input.

Besides the vertex input, the API offers several ways of accessing local and shared data. The first is transferred from the CPU application to `OpenGL` through so-called uniforms, which are accessible in one or more pipeline stages. These values are constant for the duration of one render pass. Built-in attributes and varying variables are values which can be propagated element-wise from one stage to the next one. Finally, texture samplers are used to access array data within one or more stages. Textures are characterized by their dimension (1D, 2D or 3D) and the fact that access is highly cache-optimized when reading neighboring texture elements (texels). Additionally, they have the peculiarity of holding a tuple of up to four scalar values in each element, which maps well to RGB images with optional alpha channel.

1.1.2 General Purpose Computation on GPUs

The idea of exploiting commodity graphics hardware in favor of non-graphics computations was boosted by a community of researchers working on complex problems for which a reasonable CPU runtime is hard to achieve. This new field is defined by the challenge of designing parallel algorithms suited to the characteristics of the GPU. In his dissertation, Mark Harris, one of the first promoters, proposed the name GPGPU (General-Purpose Computation on GPUs) [Har03].

Surveys of GPGPU Techniques

According to the popularity of GPGPU, there exist now several extensive overviews of related techniques. The state-of-the-art report by Owens et al. [OLG*05, OLG*07] classifies computing schemes into the following categories, which are explained below: *gather*, *scatter*, *map*, and *reduce*. The GPU Gems book series contains nearly all facets of GPU programming, including GPGPU specific chapters [Har05, Buc05]. The work by Trendal and Stewart [TS00] discusses general GPU paradigms and a new method for the real-time generation of refractive caustics. More details about scientific computation on the GPU can be found in [SDK05]. Finally, the GPGPU community maintains a website collecting various related publications[1].

Computing Schemes

The very basic principle of the GPU's SIMD architecture is the stream operation. Basically, it implies the use of a kernel processing each stream element independently. Due to this condition, it is possible to perform the processing in parallel. As the GPU supports vector arithmetic, it can be thought of as a 4$\times$ SIMD processor reading, processing and writing 4-component vectors. Stream processing comes in different flavours, which are explained in the following paragraph. These schemes are defined and supported by the concepts of the hardware architecture. Following literature, a

[1]http://www.gpgpu.org/

pipeline specific terminology is used when discussing how to realize them (i.e. vertex and fragment processing, texturing, blending, etc.); however it is clear that they can be reformulated for general APIs like `CUDA`.

Gathering describes the grouping of multiple input elements to one single output element. This is a very typical scheme for example when using a filter kernel on a texture. It can be simply realized by texture-fetching the input and writing the result of a specific calculation within a fragment program. In [OLG*05, OLG*07], gathering one-to-one element is called *map*.

Scattering corresponds to the contrary operation, i.e. one single input element is distributed to multiple output elements. When thinking in graphics, this is some kind of output selection. It can be achieved by pushing primitives (e.g. point sprites) through a vertex program, covering an appropriate region of the output buffer. A fragment program can be used to calculate the correct output values, and if necessary to combine the output with the previous state by using blending. `OpenGL` provides additive blending using alpha coefficients and a min-max blending mode for computing the minimum or maximum between the fragment output and the buffer value.

Reduce performs a compaction of an input stream, possibly to one single output element. This can be done very efficiently by step-wise halving the resolution while gathering two input elements into one at unchanged stream location. This approach has a logarithmically bounded runtime, and it can be realized in an efficient way also in 2D or 3D thanks to texture caching of the neighborhood necessary to gather the input. Alternatively, the combination of a geometry program and transform feedback can be used to remove *and* introduce new elements to a stream in a very simple and intuitive way.

Floating Point Textures

Scientific computations, e.g. in computational physics, often require floating point arithmetic, and that in best possible accuracy. Therefore, the GPU provides textures storing one of the following types: 16-bit (half) and 32-bit floating point, and a corresponding instruction set. Today's newest graphics cards even support 64-bit double precision arithmetic.

GPGPU Libraries and Debugging Tools

There exist several libraries easing the use of GPU functionality. This paragraph discusses libraries and debugging tools that build on top of traditional shader programming. Most of the information presented in this paragraph was taken from Owens et al.'s GPGPU survey [OLG*05, OLG*07].

The `Sh` library [MDTP*04] allows to embed GPU programs in a `C++` environment. It eliminates the need for (uniform) parameter binding code by sharing variables with the encapsulating application, and it also provides mechanisms for texture handling. Brook [BFH*04] can be seen as an extension to C adding a stream programming language. It is able to use different API back-ends, i.e. OpenGL or directx, but it also provides an emulation layer which is useful for debugging purposes. The GPU-accelerated system Scout provides a data parallel language for expressing scientific visualization tasks as mathematical expressions. This empowers a scientist to efficiently explore and visualize data sets [MIA*04]. The Glift template library [LKS*06] concentrates on advanced data structure, e.g. stacks, quadtrees and octrees. Components are kept reusable, so that new structures can be generated by combining them. Glift integrates with C++, a Cg or an OpenGL environment.

Debugging GPU code is difficult because the code is uploaded to GPU memory and then runs directly on hardware. Additionally, the parallel processing requires a different handling compared to traditional CPU debugging. Available debuggers are far from being as powerful as CPU solutions, however they still provide helpful utilities: gDEBugger watches the current OpenGL state and reports changes to the user. GLIntercept offers the possibility to edit program code. There also exist tools by the leading operating systems providers Microsoft and Apple, with runtime variable watches and breakpoints. The Image Debugger is capable of returning a printf-style output, which in many cases is the easiest and most powerful way to check results. The Shadesmith Fragment Program Debugger decomposes programs into multiple instances, thus allowing to check the results of single instructions. Unfortunately, Shadesmith supports assembly fragment programs only, which is a strong limitation because most complex programs are written in high-level languages and involve all stages of the pipeline. A promising system is given by the Chromium library [HHN*02] including an abstract debugging layer supporting cluster graphics as well as OpenGL. An extension proposed by Duca et al. [DNB*05] uses relational queries to describe state debugging.

The General Purpose Language CUDA

While the libraries and debugging tools mentioned above form a powerful programming framework for GPGPU applications, they all are based on the idea of mapping general computation tasks to the rendering Pipeline of the traditional GPU. The CUDA language, on the other side, defines a language for general parallel processing without relying on graphics APIs like OpenGL or directx. In principle, the functionality exhibited by CUDA is directly supported by the graphics hardware. Naturally, this requires the hardware or at least the graphics card driver to support CUDA. Currently, this is the case for NVidia hardware only. However, a similar language called OpenCL is about to become a widely supported alternative to CUDA in near future.

Due to the independence of the rendering pipeline, CUDA makes more sense when designing applications without any relation to graphics. Other applications, in contrast, might benefit from the capability implemented within the pipeline. This is not only the case for pure rendering tasks, e.g. geometry processing fits well into the pipeline concept as well. For example, the geometry shader, which is able to insert and delete elements in a vertex stream, has no direct equivalent in CUDA. A similar technique available in CUDA is the more general concept of *stream compaction* [HSO07], however on-the-fly alteration of the stream is difficult to achieve. Thus, the use of a geometry program might be a criterion for using the shader-based approach instead of `CUDA`.

It should be noted that, besides this reasoning, `CUDA` has several advantages: `CUDA` provides synchronization mechanisms and a more powerful memory management. Parallel threads can (in specific constellations called blocks) access a common shared memory which is not available in shader programming. Besides, scattering is explicitly supported in `CUDA` as a generic write operation: The accumulation of scattered data, i.e. blending, can be achieved by atomic operations (e.g. read-add-write aka `atomicAdd`), which is possible since compute capability 1.1 [NVi07].

In summary, a clear advance towards `CUDA` and similar languages is taking place at the time of writing this document. Future hardware support might concentrate on general languages like `CUDA`, while rendering APIs including the `OpenGL` pipeline might be provided as a software library written in that language.

GPGPU Applications

Within a few years, a lot of interesting scientific work emerged in the context of GPGPU, ranging from advanced rendering over physical simulation (e.g. as provided by the *Havok Physics* engine) up to purely hardware specific aspects. Accordingly, Purcell et al. and Carr et al. [PBMH02, CHH02] present methods for mapping ray tracing to graphics hardware. Another example for advanced rendering is volume rendering, which was highly stimulated by the growing GPU functionality. See [RS02] for an extensive discussion of volume rendering techniques with application to medical imaging. Kolb et al. have shown that large particle systems can be simulated on the GPU in real-time including collision detection [KLRS04]. Rumpf and Strzodka present a level set segmentation of the human brain, accelerated using early graphics hardware [RS01]. Physical simulation examples include a 3D coupled map lattice simulation [HCSL02], solving partial differential equations for cloud dynamics [HBISL03], and many more. Research has also been done concerning the abstract layer of GPU programming, e.g. expanding multiple texture channels to higher-bit float representations [Str02], and mapping multiple-instruction multiple-data (MIMD) code to the single-instruction capability (SIMD) of the GPU [Die92, DK93]. All these results show that there is a high need for the computational power of the GPU, and that graphics hardware is much more than just a rendering accelerator.

1.2 Particle Systems

A particle system is one of the most common 3D object representations, next to polygonal models (meshes), non-polygonal models (constructive solid geometries, implicit and parametric surfaces), and voxel grids. Its definition is very easy: A particle system is composed of a set of points, the particles. These move according to a specific rule, and they are *alive*, in the sense that they are born and they die after a certain period of time. Due to their cloudy appearance, particle systems are particularly suited for fuzzy objects which do not possess smooth and well-defined surfaces. This property makes them an ideal tool for representing volumetric effects like fire, explosions, smoke, etc. Those effects can be realized according to simplistic and/or stochastic models, resulting in appealing, but hardly realistic animations (e.g. [Ree83, Sim90]). More sophisticated approaches include collision detection with polygonal objects [KLRS04] or particle-particle collision [KSW04]. Other methods make use of complex physical models for accurately simulating real-life phenomena like fluids [MCG03], crowds or bird-oid objects (*boids*) [Rey87, Ker90]. Particle systems are also well-suited for visualizing flow data sets: Krüger et al. present such a system suited for very large and dense particle sets [KKKW05]. Both fluids and flow visualization are essential subjects to my thesis and will be addressed in detail later.

1.2.1 Particle System Characteristics

Particle systems can be classified into two categories: stateless and state-preserving systems. Stateless systems do not store particle specific information for update during simulation. In particular, motion is calculated according to global parameters, e.g. time and starting positions. This type of particle systems is suited for visual effects, however particles cannot react on their environment independently, and particle-particle interaction is impossible to achieve. State-preserving particle systems are much more powerful in the sense that inter-particle processing is made possible by separate per-particle storage of the position and/or the velocity. Sometimes, it may be useful to store additional attributes like color, size, opacity etc.

Besides this distinction, particle birth and death are major aspects to be configured. Particle birth can be seen as a particle-wise (re)initialization to specific seed positions. Typical implementations arrange the seeds within a geometric shape, the particle emitter. Fig. 1.2 shows a couple of examples. Interactive systems allow the user to move, rotate or scale the emitter shape. Particle death might occur when a particle's lifetime has elapsed. An alternative definition is to reinitialize the oldest particles whenever newly emitted ones exceed the largest possible number of particles in the system.

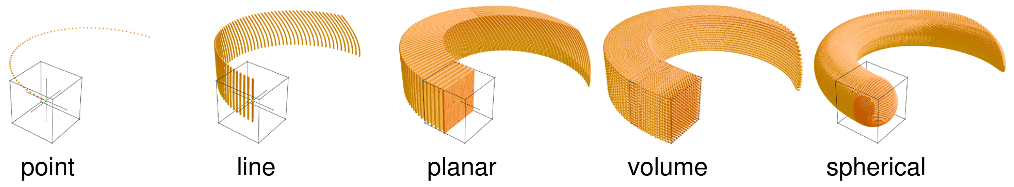


Figure 1.2: *Different particle emitter shapes. The emitted geometry is defined to fit into the bounding box.*

1.2.2 Particle Tracing

Consider the initial value problem given by $\mathbf{x}(0) = \mathbf{x}_0$ and the following differential equation:

$$\mathbf{x}'(t) \;=\; \vec{\mathbf{v}}(t), \tag{1.1}$$

where $\mathbf{x}_0 \in \mathbb{R}^3$ and $\mathbf{x}, \vec{\mathbf{v}} : \mathbb{R} \to \mathbb{R}^3$. Interpreting $\mathbf{x}(t)$ as a particle's position and $\vec{\mathbf{v}}(t)$ as its velocity at time t, Eq. 1.1 just describes the motion of a particle. The solution for $\mathbf{x}$ is defined by the following integral:

$$\mathbf{x}(t) \;=\; \mathbf{x}_0 + \int\limits_0^t \vec{\mathbf{v}}(\tau)\; d\tau \tag{1.2}$$

This is an intuitive analogy to a particle's trajectory.

Note that when referring to the term *unsteady flow* in later parts, a velocity $\vec{\mathbf{v}}$ depending on time *and* space is meant, i.e. the more general form $\vec{\mathbf{v}} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$. This adds a recursive element to the integral form of the particle trajectory:

$$\mathbf{x}(t) \;=\; \mathbf{x}_0 + \int\limits_0^t \vec{\mathbf{v}}(\mathbf{x}(\tau), \tau)\; d\tau \tag{1.3}$$

Simulators solve Eq. 1.2 by numerical integration (also called quadrature), e.g. Euler or a higher order scheme like Runge-Kutta. The following paragraphs discuss the main integration schemes and especially those that are used in later parts of the dissertation. For extensive overviews of numerical integration, please refer to the mainstream literature about numerical mathematics [SB00, Stö95, HH89]. Fig. 1.3 visualizes the accuracy of different schemes by means of two velocity fields. The first consists of a simple oscillator defined by $\vec{\mathbf{v}}_1(t) = (-sin(t), cos(t))^{\mathrm{T}}$. The second is defined by a repetitive B-Spline basic function of order 3 (see [PBP02] for a detailed discussion of B-Splines), whose normalized derivative forms the velocity:

$$\vec{\mathbf{v}}_2(u) \;=\; \frac{\Big(k,\; N_0^{3\prime}(u)\Big)^T}{\left\| \Big(k,\; N_0^{3\prime}(u)\Big)^T \right\|}, \tag{1.4}$$

where $u \in \mathbb{R}$ is chosen to feature the repetitiveness of the curve, i.e. $u = t \bmod 3$, and k is a scaling factor for making the curve narrower. The normalization assures that the overall curve is equally sampled, including the challenging peaks, thus visibly revealing the accuracy of the integration methods.

The Euler Integration Scheme

Probably the easiest solution to Eq. 1.2 is given by the Euler integration, which discretizes the integral into steps according to a time interval $\Delta \in \mathbb{R}$:

$$\mathbf{x}(\tau + \Delta) \;=\; \mathbf{x}(\tau) + \Delta \cdot \vec{\mathbf{v}}\big(\mathbf{x}(\tau),\; \tau\big) \tag{1.5}$$

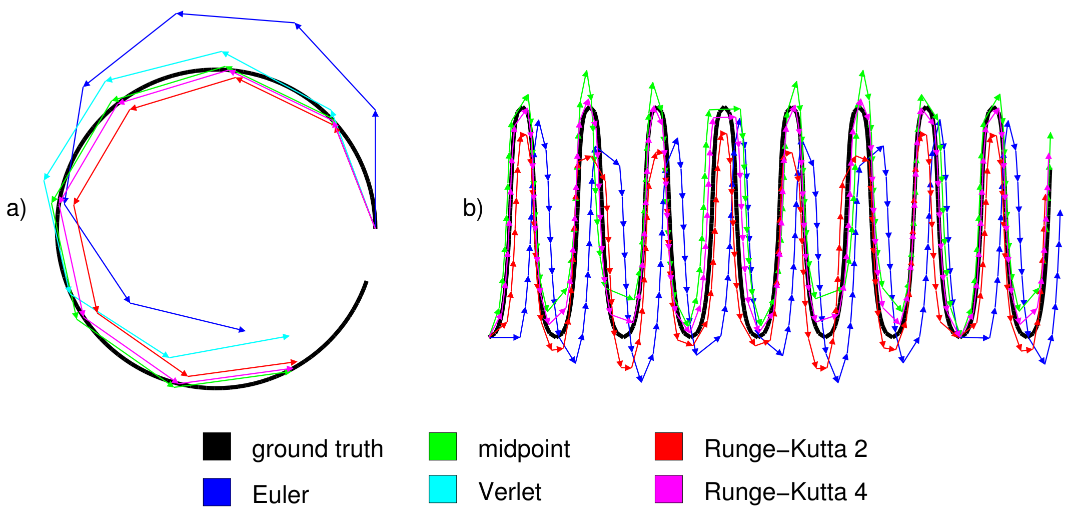


Figure 1.3: *Different numerical integration schemes. One can easily see the correlation of the order (Euler: 1, midpoint, Verlet, RK2: 2, RK4: 4) and the accuracy of the outcome.*

Midpoint and Higher-order Integration

A simple improvement to the Euler integration can be achieved by evaluating $\vec{\mathbf{v}}$ using the *midpoint* at $\Delta/2$:

$$\begin{aligned} \mathbf{x}(\tau + \frac{\Delta}{2}) &= \mathbf{x}(\tau) + \frac{\Delta}{2} \cdot \vec{\mathbf{v}}\big(\mathbf{x}(\tau),\ \tau\big) \\ \mathbf{x}(\tau + \Delta) &= \mathbf{x}(\tau) + \Delta \cdot \vec{\mathbf{v}}\Big(\mathbf{x}(\tau + \frac{\Delta}{2}),\ \tau + \frac{\Delta}{2}\Big) \end{aligned} \tag{1.6}$$

This approach is especially interesting when $\vec{\mathbf{v}}$ depends on time only, because then one can spare the first equation, leading to the following simplification:

$$\mathbf{x}(\tau + \Delta) = \mathbf{x}(\tau) + \Delta \cdot \vec{\mathbf{v}}(\tau + \frac{\Delta}{2})$$

Still, the necessity to forward-evaluate $\vec{\mathbf{v}}$ remains, which may be difficult if $\vec{\mathbf{v}}$ is calculated by numerical integration itself. For this case, the *Leapfrog* algorithm integrates the velocity and the acceleration $\vec{\mathbf{a}} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$ in an alternating way:

$$\begin{aligned} \mathbf{x}(\tau + \Delta) &= \mathbf{x}(\tau) + \Delta \cdot \vec{\mathbf{v}}(\tau + \frac{\Delta}{2}) \\ \vec{\mathbf{v}}(\tau + \frac{3\Delta}{2}) &= \vec{\mathbf{v}}(\tau + \frac{\Delta}{2}) + \Delta \cdot \vec{\mathbf{a}}(\tau + \Delta) \end{aligned}$$

The second equation can be split to provide the velocity at time $\tau + \Delta$. Approximating the initial velocity $\vec{\mathbf{v}}(\frac{\Delta}{2})$ by $\vec{\mathbf{v}}_0 + \frac{\Delta}{2}\vec{\mathbf{a}}_0$ leads to the (velocity) *Verlet* algorithm, named after the French physicist Loup Verlet. This algorithm is well-suited for applications working on forces, i.e. based on acceleration, like fluids.

A generalization of the midpoint method is given by the Runge-Kutta (RK) integration. The RK2 scheme combines two forward steps into an accurate target position:

$$\begin{aligned}
\vec{\mathbf{k}}_1 &= \vec{\mathbf{v}}\Big(\mathbf{x}(\tau),\ \tau\Big) \\
\vec{\mathbf{k}}_2 &= \vec{\mathbf{v}}\Big(\mathbf{x}(\tau) + \Delta \cdot \vec{\mathbf{k}}_1,\ \tau + \Delta\Big) \\
\mathbf{x}(\tau+\Delta) &= \mathbf{x}(\tau) + \Delta \cdot \frac{(\vec{\mathbf{k}}_1 + \vec{\mathbf{k}}_2)}{2}
\end{aligned}$$

The RK4 scheme performs four forward steps, and it is a very commonly used integration method because of its high accuracy:

$$\begin{aligned}
\vec{\mathbf{k}}_1 &= \vec{\mathbf{v}}\Big(\mathbf{x}(\tau),\ \tau\Big) \\
\vec{\mathbf{k}}_2 &= \vec{\mathbf{v}}\Big(\mathbf{x}(\tau) + \frac{\Delta}{2} \cdot \vec{\mathbf{k}}_1,\ \tau + \frac{\Delta}{2}\Big) \\
\vec{\mathbf{k}}_3 &= \vec{\mathbf{v}}\Big(\mathbf{x}(\tau) + \frac{\Delta}{2} \cdot \vec{\mathbf{k}}_2,\ \tau + \frac{\Delta}{2}\Big) \\
\vec{\mathbf{k}}_4 &= \vec{\mathbf{v}}\Big(\mathbf{x}(\tau) + \Delta \cdot \vec{\mathbf{k}}_3,\ \tau + \Delta\Big) \\
\mathbf{x}(\tau+\Delta) &= \mathbf{x}(\tau) + \Delta \cdot \frac{(\vec{\mathbf{k}}_1 + 2\vec{\mathbf{k}}_2 + 2\vec{\mathbf{k}}_3 + \vec{\mathbf{k}}_4)}{6} \qquad (1.7)
\end{aligned}$$

In the following chapters, I will refer to RK4 as an operator $\mathtt{RK}^4\big(\mathbf{x},\ t,\ \Delta\big)$ which calculates a new location according to a given starting point $\mathbf{x}$, an instant of time t and a time step Δ.

Error Analysis

Numerical integration suffers from two kinds of errors: the truncation error and the round-off error. The first occurs when discretizing time using a fixed step size, the latter is due to finite floating-point arithmetic and increases proportionally to the number of integration steps. This paragraph concentrates on the truncation error, as in general, round-off errors are caused by insufficient floating point resolution, which is an implementation specific issue.

The Euler integration produces a large truncation error. It is said to be first order because the accumulating error is linearly bounded by Δ. Proof: The expansion of $\mathbf{x}(\tau+\Delta)$ to its Taylor series (in neighborhood τ)

$$\mathbf{x}(\tau+\Delta) = \mathbf{x}(\tau) + \Delta \cdot \mathbf{x}'(\tau) + \frac{\Delta^2}{2} \cdot \mathbf{x}''(\tau) + \frac{\Delta^3}{6} \cdot \mathbf{x}'''(\tau) + \mathcal{O}(\Delta^4) \qquad (1.8)$$

reveals that, for a single step $\Delta \in (0,1)$, the dominating error given by the difference between Eq. 1.8 and Eq. 1.5 is $(\Delta^2/2) \cdot \mathbf{x}''(\tau) \in \mathcal{O}(\Delta^2)$. Assuming a total number of t/Δ steps, the error of $\mathbf{x}(t)$ sums up to $\mathcal{O}(\Delta)$. $\square$

Similarly, the midpoint method can be proven to be of second order. Proof: Expanding the velocity term in Eq.1.6 yields:

$$\begin{aligned}
\mathbf{x}(\tau+\Delta) &= \mathbf{x}(\tau) + \Delta \cdot \mathbf{x}'(\tau + \frac{\Delta}{2}) \\
&= \mathbf{x}(\tau) + \Delta \cdot \Big(\mathbf{x}'(\tau) + \frac{\Delta}{2}\mathbf{x}''(\tau) + \frac{\Delta^2}{4}\mathbf{x}'''(\tau) + \mathcal{O}(\Delta^3)\Big) \\
&= \mathbf{x}(\tau) + \Delta \cdot \mathbf{x}'(\tau) + \frac{\Delta^2}{2}\mathbf{x}''(\tau) + \frac{\Delta^3}{4}\mathbf{x}'''(\tau) + \mathcal{O}(\Delta^3)
\end{aligned}$$

Here, the error, i.e. the difference to Eq. 1.8, is bounded by $\mathcal{O}(\Delta^3)$. This implies a quadratic error accumulation, thus order 2. □

As the Leapfrog and thus the Verlet algorithm are both based on midpoint evaluation, those are schemes of second order, too. For an error analysis of the Runge-Kutta integration schemes, see [CP92].

Adaptive Time-stepping

Adaptive adjustment of the time step during the numerical integration aims for reducing both the truncation error and the round-off error. Thus, a good adaptive time step is small enough to avoid truncation and large enough to avoid round-off. For the Runge-Kutta 4 method, an easy and common method is step-doubling. Step-doubling determines a numerically sound time step Δ' as follows:

$$\begin{aligned}
\mathbf{x}_1 &= \mathtt{RK}^4\big(\mathbf{x},\ t,\ \Delta\big) \\
\mathbf{x}_2 &= \mathtt{RK}^4\big(\mathtt{RK}^4\big(\mathbf{x},\ t,\ \frac{\Delta}{2}\big),\ t+\frac{\Delta}{2},\ \frac{\Delta}{2}\big) \\
\Delta' &= \Delta \cdot \left(\frac{d}{\|\mathbf{x}_2 - \mathbf{x}_1\|}\right)^{\frac{1}{5}} \qquad (1.9)
\end{aligned}$$

The distance $\|\mathbf{x}_2 - \mathbf{x}_1\|$ is an error estimation which is asymptotically bounded by Δ^5. The target distance d must be chosen to produce smooth results (i.e. no sharp peaks). It is advisable to avoid a denominator near zero. Also, it might be reasonable to clamp Δ' to be within a reasonable interval.

Note that high order integration schemes, in particular Runge-Kutta and adaptive time-stepping according to the step-doubling approach will be essential ingredients for the realization of accurate particle traces in later chapters. Also note that the generation of accurate flow lines (see Sec. 4.2) supports any suitable adaptive time-stepping scheme, including step-doubling.

1.2.3 Hardware-accelerated Particle Systems

Stateless particle systems are straight-forward to realize on programmable graphics hardware: Even early platforms support the storage of a vertex buffer in graphics memory, and on-the-fly transform using a vertex program. Each vertex represents a particle, which is animated by evaluating a formula depending on some input parameters, e.g. time and initial values. Vertices coming out from the vertex shader are

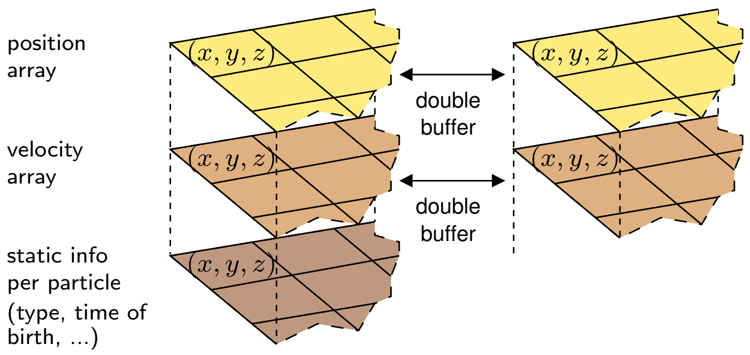


Figure 1.4: *Data storage concept for particle systems. Source: [KLRS04]*

rasterized and rendered as points or screen-aligned sprites (so-called *point sprites*). This overall method fits well into one single render pass through the pipeline.

The first state-preserving particle system engines exploiting consumer graphics hardware were simultaneously presented in 2004 by Kolb et al. [KLRS04] and Kipfer et al. [KSW04]. Both methods are capable of simulating and rendering about one million particles at interactive frame rate. The work by Kolb et al. focuses on particle-object collision using depth maps and a novel parametrization for texture-based normal indexing. Particles are partially sorted on the GPU for blended back-to-front rendering. The engine presented by Kipfer et al. supports particle-scene collision using a precomputed signed 3D distance field or a heightfield, and inter-particle collision by twice sorting particles according to a staggered grid enumeration.

Today's shader APIs provide two mechanisms that can be used for a state-preserving transformation. The first is the render-to-texture capability, which is used in [KLRS04, KSW04]. The second is the newer transform feedback (see Sec. 1.1.1). Both are discussed shortly in the following two paragraphs.

Texture-based Particle Processing

In general, the input and output buffers are better kept distinct when processing the data in parallel on the GPU. This avoids side-effects when reading and writing at the same buffer location, and it guarantees a well-defined read operation (see Sec. 4.4.3 in the `OpenGL` specification [SA08]). Kolb et al. use a double-buffered texture storage in their approach, which is shown in Fig. 1.4. One update pass corresponds to one fragment processing pass. The process is triggered by rendering a quadrilateral covering the complete output texture. After rasterization, the fragment processor then performs the actual particle simulation. The previous state, i.e. the input texture is read by texture-fetching. Thus, particles are identified and accessed through texture coordinates.

Kolb et al. accomplish particle initialization, i.e. birth and death, on the CPU. This is the only step of the simulation process which is not performed on graphics hardware. Storing idle particle indices in a heap structure, which is optimized to always return the smallest element, ensures that the particle data is stored in a compact region of the particle texture. This idea is crucial in the general case where particles are assigned different lifetimes, or if reinitialization is inherently sequential and thus requires a CPU-based routine.

Stream-based Particle Processing

The combination of a geometry program and transform feedback is a powerful alternative on newer hardware. The reinterpretation of texture data as a vertex buffer for rendering becomes obsolete, because the programs work directly on the vertex data. The input stream is processed and modified on-the-fly in the geometry program. Provided that reinitialization is non-sequential in nature (e.g. random reinitialization), element insertion and deletion can be used for an easy implementation of birth and death without any detour involving the CPU.

1.3 Grids and Level Sets

In this section, grids are introduced as a counter-technique to particle systems, aiming for the representation of deformable objects. This includes the notion of advection, which is a direct analog to particle tracing when thinking of the whole particle system as one object. This also includes level sets, which are presented as a method for tracing closed surfaces, then its extension called particle level set (PLS), and the problem of constructing a grid-based distance transform (DT). The latter is strongly motivated by the practical usefulness of DTs in the context of PLS (see Sec. 3.3).

It should be stated that this section is not an overview of all occurrences of grids in literature, not even in graphics, as this would go far beyond the scope of my work. Consequently, I will concentrate on the grid-based techniques that are useful in the following chapters.

Different Grid Types

Grids appear in nearly as many flavours as they have different applications in literature. In our context, a grid is a discretization of space in cells that are arranged at well-defined locations. In principle, the concept of a *mesh* (i.e. a polygonal model) is situated in-between of grids and particles, because it can be seen as a system of connected particles living in a lower dimension, whereas grids are subject to an even stricter arrangement. The following distinction for grids is proposed:

Structured grids are characterized by the fact that cells are arranged in n-dimensional space in the same way as they are indexed according to a n-ary Cartesian product. This class is subdivided into *rectilinear* and *regular* grids. The regular, also called *Cartesian* grid, is more restrictive than the rectilinear one due to its equidistant (also: uniform) cell arrangement in each dimension. Often, the definition of Cartesian grids differs from the one for regular grids in the sense that Cartesian grids consist of cubic cells, while the regular grid only requires congruence of the cells. Curvilinear grids apply an invertible transformation of the cell arrangement to curved (e.g. spherical) coordinates.

Unstructured grids, e.g. tetrahedral grids, may be appropriate when the application domain does not fit into the aforementioned restrictions.

Note that when writing about grids without any predicate in the following chapters, regular/Cartesian grids are meant. The term *non-uniform* grid is used for rectilinear grids for which a non-equidistant arrangement is explicitly assumed.

1.3.1 Field Advection

Advection can be seen as the process of applying motion to a field containing a scalar (or non-scalar) property. This property, which can also be interpreted as a substance, is transported by the flow, i.e. a velocity vector field, defining the applied motion.

Let us assume a 3-dimensional space, in which a scalar field a and a velocity field $\vec{\mathbf{v}}$ are defined, both time-varying, i.e. $a : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}$ and $\vec{\mathbf{v}} : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3$. The following equation, at position $\mathbf{x} \in \mathbb{R}^3$ and time $t \in \mathbb{R}$, expresses the advection of a:

$$\frac{\partial a(\mathbf{x},t)}{\partial t} + \nabla \cdot \big(a(\mathbf{x},t) \cdot \vec{\mathbf{v}}(\mathbf{x},t)\big) \;=\; 0$$

We now consider a divergence-free velocity field $\vec{\mathbf{v}}$, which is given when modeling incompressible fluid flows. Consequently, we have $\nabla \cdot \vec{\mathbf{v}} = \vec{\mathbf{0}}$, and with the product rule $\nabla \cdot (a\vec{\mathbf{v}}) = (\nabla a) \cdot \vec{\mathbf{v}} + a \cdot (\nabla \cdot \vec{\mathbf{v}})$ we obtain:

$$\frac{\partial a(\mathbf{x},t)}{\partial t} + \vec{\mathbf{v}}(\mathbf{x},t) \cdot \nabla a(\mathbf{x},t) \;=\; 0 \tag{1.10}$$

Finding a solution to this equation is a typical requirement for incompressible Navier-Stokes-based fluid solvers. One of the fluid pioneers in the graphics community is Jos Stam, who strongly influenced this research field by presenting a method called *stable fluids* [Sta99]. The main advantage of his approach compared to previous ones is that simulation is unconditionally stable, independently of the time interval between two simulation steps. His stable fluid method is based on a grid-based – yet particle-related, thus "Lagrange" – interpretation of the advection equation, which is known as *semi-Lagrange* advection. This kind of advection schemes primarily evolved in the context of meteorological science (see [SC91] for an overview).

Semi-Lagrange Advection

So-called "Eulerian" advection schemes (i.e. schemes defined in *Eulerian coordinates*), based on Eq. 1.10, describe the evolution w.r.t. to fixed points in space, whereas "Lagrangian" schemes consider the explicit motion of single particles. Eulerian methods embed the advected field in a regular grid, which has the advantage that the computational space is regularly and constantly covered. However, Eulerian methods suffer from problems related to computational stability. In other words, these schemes require a strict bound on the maximum possible time step, which is traditionally expressed using the CFL (Courant-Friedrichs-Lewy) condition [CFL28]. The semi-Lagrange advection surpasses this restriction by interpreting Eulerian advection as a per-point backward particle tracing.

The key idea of the semi-Lagrange advection is very intuitive when thinking of a field location $\vec{\mathbf{x}}$ to be a particle holding a portion of the field to be advected. The key is now to back-trace this particle through the velocity field $\vec{\mathbf{v}}$. Assuming a time step Δ, this yields the following scheme:

$$a(\mathbf{x},\ t+\Delta) \;=\; a\big(\mathbf{p}(\mathbf{x},-\Delta),\ t\big), \tag{1.11}$$

where $\mathbf{p} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$ is the back-trace path of the particle. Approximating this path by a (backward) Euler integration step, we get the following formulation of the semi-Lagrange advection:

$$a(\mathbf{x},\ t+\Delta) \;=\; a\big(\mathbf{x} - \Delta\vec{\mathbf{v}}(\mathbf{x},t),\ t\big)$$

The derivation of Eq. 1.11 solving Eq. 1.10 is based on the method of characteristics. Please refer to the appendix in [Sta99] for the demonstration. Note that the semi-Lagrange advection is not restricted to scalar fields. Stam makes use of it for velocity advection in his stable fluids approach.

Grid Advection

Obviously, grid advection can be interpreted as an operation on texture data. Texture advection maps well to the GPU: Advection can be performed efficiently in parallel, and the GPU also provides functions to directly render the texture, by use of texture mapping in the case of 2D textures, or by means of volume rendering techniques in the 3D case. One of the first approaches for texture advection using graphics hardware was provided by Max [MB95]. Here, the advection is delegated to texture mapping by altering texture coordinates.

It is clear that semi-Lagrange advection can be applied to a texture representation. Generally, this requires an interpolation of the back-traced location, which will typically lie in-between of neighbor texels. A reasonable approach is to compute a trilinear interpolation of the eight surrounding texels. However, one must take care of numerical diffusion due to the approximate nature of interpolation. This effect leads to blurriness, and a loss of previously sharp interfaces. Level set methods are motivated by the task of accurately advecting such interfaces (see [Wei04] for a discussion).

1.3.2 Level Sets

The level set method was introduced by Osher and Sethian [OS88]. It aims for overcoming the problem of numerical diffusion in previous grid-based surface tracing approaches. The key idea is to identify the surface as the zero-set *interface* within a scalar field: the *level set*. A detailed discussion of the technique and its applications can be found in [OF02].

Level sets enjoy a noticeable popularity in a wide range of disciplines. A typical application is the field of incompressible fluid dynamics, including the simulation of water [ELF05] and smoke [FSJ01]. Another example is image segmentation [CV00].

The level set method represents a surface, i.e. a lower dimensional interface I in a metric space D (which could be for example $I \subset D = \mathbb{R}^3$), by the iso-contour of a level set function $\phi : D \to \mathbb{R}$:

$$I(\phi) := \{\mathbf{x} \in D \mid \phi(\mathbf{x}) = 0\}$$

The interface is moved by advecting ϕ within the velocity field according to Eq.1.10.

Typically, ϕ is initialized to be a signed Euclidean distance field. However, after advection, this property is generally lost. Thus, ϕ needs to be reinitialized in order to keep a valid distance field. Note that reinitialization is not done for its own sake. In practice, a frequent reinitialization is numerically useful for two reasons (see also [OF02]): First, areas far from the interface are discarded, which is convenient as this avoids errors from outside to be propagated into the iso-contour and thus into the area of interest. Second, the accumulation of diffusive errors provoking a smearing of the interface is prevented by its relocation and by restoring a sharp representation.

As distance field construction will be one component of the later level set based approach, a short excursus to distance fields and distance *transforms* is provided in the next section. The latter is also motivated by later use, as the distance transform will turn out to be preferable in the GPU-based particle level set method.

1.3.3 Distance Fields and Distance Transforms

Computing a signed or unsigned distance field is a well studied problem [Cui99]. Besides, distance fields have many applications in computer graphics, scientific visualization and related areas. Examples are implicit surface representation and collision detection [KLRS04], skeletonization [ST04] and accelerated volume ray tracing [HSS*05]. Depending on the application, it may be useful to compute distances and closest-point references as well. This structure is called a distance transform (DT).

Consider a closed object Ω defined in 2- or 3-dimensional space. Let $\partial\Omega$ be the object's boundary. The Euclidean distance transform $\mathbf{dt}$ for a location $\mathbf{x}$ is defined as $\mathbf{dt}(\mathbf{x}) = \big(\mathbf{dt}_d(\mathbf{x}),\ \mathbf{dt}_\delta(\mathbf{x})\big)$, with

$$\begin{aligned} \mathbf{dt}_d(\mathbf{x}) &= s_\mathbf{x} \cdot \min_{\mathbf{y}\in\partial\Omega}\{\|\mathbf{x}-\mathbf{y}\|\}, \\ \mathbf{dt}_\delta(\mathbf{x}) &= \arg\min_{\mathbf{y}\in\partial\Omega}\{\|\mathbf{x}-\mathbf{y}\|\}, \end{aligned}$$

where $s_\mathbf{x}$ denotes the sign w.r.t. the boundary $\partial\Omega$ (i.e. 1: exterior, -1: interior), and $\arg\min_\mathbf{y}$ is an operator returning a point $\mathbf{y}$ constituting the minimum. Thus, $\mathbf{dt}(\mathbf{x})$ stores the signed distance and the point $\mathbf{y} \in \partial\Omega$ closest to $\mathbf{x}$. In the following, this closest point to the interface will be called a *reference*.

Computing a 3-dimensional Euclidean DT is a frequent problem [CM99]. Depending on the initial object representation, given as grid or as geometric representation, different approaches have been proposed. Consider a grid, where the object's boundary is identified by a narrow band of interior and exterior cells. This representation is sound, provided that the boundary is closed. Algorithms computing a grid-based

DT can be classified into two major categories: methods based on Voronoi diagrams and methods based on distance propagation.

The Voronoi Diagram Approach

A Voronoi diagram is a space partitioning into cells w.r.t. to a fixed set of seeds (also *sites*). Each cell contains all points closest to one seed. It is clear that the DT can be obtained by setting Voronoi sites onto the object's boundary.

Two-dimensional Voronoi diagrams can easily be constructed using rasterization techniques. For this, cones with a common opening angle are placed over each seed and the resulting scene is rendered from top-view using depth test. In a general Voronoi diagram, the seeds can consist of more geometric objects than just points, e.g. line segments, triangles or curves. Hoff et al. [HKL*99] have shown that the cone approach can be extended to reflect general Voronoi diagrams, and they support 3D diagrams by slice-by-slice construction. Because this approach is based on polygonal rasterization, it can be implemented using graphics hardware.

The Distance Propagation Approach

Propagation methods continuously propagate the distance information to neighboring grid cells, either by spatial sweeping or by contour propagation. As described earlier, a specific initialization $\mathbf{dt}^0$ of the grid is considered where cells close to the boundary $\partial\Omega$ are assigned an initial configuration. All other cells are set to a large positive or negative distance M distinguishing the inner region Ω^- and outer region Ω^+ of the object:

$$\mathbf{dt}^0(\mathbf{x}) = \begin{cases} (0, \mathbf{x}) & \text{if } \mathbf{x} \in \partial\Omega \\ (\pm M, *) & \text{if } \mathbf{x} \in \Omega^\pm \end{cases} \tag{1.12}$$

Alternatively, the first case can be defined to describe precise initial subvoxel references if available. Note that the reference part $\mathbf{dt}_\delta$ in the second case is undefined. This missing information is computed during the iterative propagation. One propagation step can be seen as a filter pass using a kernel $\mathcal{N}_{27}$ including all direct neighbors around grid cell $\mathbf{x}$:

$$\mathbf{dt}_\delta^{i+1}(\mathbf{x}) = \underset{\mathbf{y}\in\mathcal{N}_{27}(\mathbf{x})}{\arg\min}\{\|\mathbf{dt}_\delta^i(\mathbf{y}) - \mathbf{x}\|\} \tag{1.13}$$

The distance component $\mathbf{dt}_d^{i+1}(\mathbf{x})$ can be set to have the same sign as before:

$$\mathbf{dt}_d^{i+1}(\mathbf{x}) = \operatorname{sign}\left(\mathbf{dt}_d^i(\mathbf{x})\right) \cdot \left\|\mathbf{dt}_\delta^{i+1}(\mathbf{x}) - \mathbf{x}\right\|$$

See Fig. 1.5 for a visualization of two sequential propagation steps. It should be noted that distance propagation as defined in Eq. 1.13 can produce wrong results in cases where a reference $\mathbf{dt}_\delta(\mathbf{x})$ initially stored in the boundary configuration never reaches the grid cell $\mathbf{x}$. Such a situation can occur when the reference is overwritten by a local minimum during stepwise propagation somewhere on the path from the initial location to $\mathbf{x}$ (see [CM99] for an illustration of such a case).

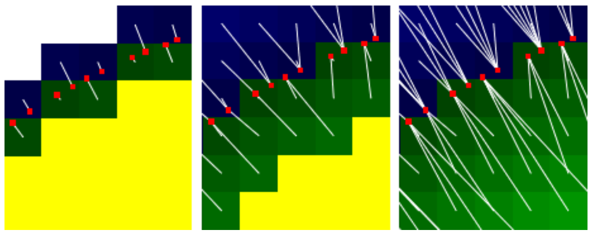


Figure 1.5: *Constructing a distance transform. Two propagation steps (white/yellow: $\pm M$, blue/green: $+$exterior/$-$interior). Note that the grid is initialized with precise subpixel references near the boundary in this example.*

Thus, distance propagation methods generally produce approximate solutions to the DT construction problem.

Rong and Tan [RT06] present a variant of the propagation approach, the jump flooding algorithm (JFA), which is parallelizable on the GPU, and which can be used for the computation of Voronoi Diagrams as well as DTs. The key idea of JFA is to propagate the distance information to more distant cells instead of neighbor cells only. This leads to a faster algorithm, as the grid can be filled within a smaller number of propagation steps. A more detailed description of JFA and a faster alternative is given in Sec. 3.2.1 in the context of level set reinitialization.

1.3.4 Particle Level Sets

The level set approach constitutes a powerful model for describing the tracing of closed surfaces in an arbitrary flow. However, in practice, a grid-based representation of a level set is difficult to handle, because grid advection schemes suffer from numerical diffusion (see Sec. 3.1). A frequent reinitialization of the level set function is advantageous, yet a volume loss and feature smearing still can be observed. The particle level set (PLS) method, first introduced in the work by Enright et al. [EFFM02, EMF02, ELF05], improves the accuracy by interchanging the representation of the surface between the grid and a corrective set of particles. For a more detailed motivation of PLS-based methods, the reader is referred to the introduction in Chap. 3.

The PLS Algorithm

Enright et al. use a fast first order accurate semi-Lagrangian method to advect the level set function ϕ in a grid [ELF05]. The volume loss due to numerical diffusion is prevented by placing corrective particles nearby the interface I. Positive particles are located in the outer region, i.e. $\phi > 0$, and negative particles in the inner region, i.e. $\phi < 0$. Each particle p is defined as a sphere around $\mathbf{x}_p$ with a radius $r_p \in [r_{\min}, r_{\max}]$ touching the interface: thus, $r_p = s_p \cdot \phi(\mathbf{x}_p)$, where s_p is the sign of the particle. The correction step involves the definition of a temporary level set function ϕ_p around each particle:

$$\phi_p(\mathbf{x}) = s_p \cdot (r_p - \|\mathbf{x} - \mathbf{x}_p\|)$$

After level set and particle advection, *escaped* particles, i.e. those that are further away than their radius on the wrong side of the interface, are used for the level set correction. Each escaped particle contributes to the eight surrounding grid cells using intermediate level set functions ϕ^+ and ϕ^- that are initialized to ϕ and updated according to the following formulas:

$$\begin{aligned} \phi^+(\mathbf{x}) &\longleftarrow \max(\phi_p(\mathbf{x}),\ \phi^+(\mathbf{x})), \\ \phi^-(\mathbf{x}) &\longleftarrow \min(\phi_p(\mathbf{x}),\ \phi^-(\mathbf{x})) \end{aligned} \tag{1.14}$$

After processing all escaped particles, a new (corrected) ϕ is constructed according to the following operation and then reinitialized in order to restore a signed distance function.

$$\phi(\mathbf{x}) = \begin{cases} \phi^+(\mathbf{x}) & \text{if} \quad \|\phi^+(\mathbf{x})\| \leq \|\phi^-(\mathbf{x})\| \\ \phi^-(\mathbf{x}) & \text{else} \end{cases} \tag{1.15}$$

The PLS method is known to produce good results even when performing a first order semi-Lagrangian level set advection. The algorithm according to Enright et al. [ELF05] is summarized below. Note that the level set correction is performed twice.

Algorithm 1.1 (particle level set algorithm)

```
1 Definition of the interface location and velocity field
2 Initialization of the level set $\phi$ based on the interface
3 First order semi−Lagrangian level set advection
4 Second order Runge−Kutta particle advection
5 Correction of the level set using the particles according to Eq. 1.14 and 1.15
6 Level set reinitialization
7 Correction of the level set using the particles (same as step 5)
8 Particle reseeding
9 Go to 3
```

Mokberi and Faloutsos provide an open source library implementing the PLS method, as well as a technical report explaining their own interpretation of the original approach [MF06]. Note that level set correction (step 5) is repeated in step 7 of the original PLS algorithm (Alg. 1.1). Mokberi and Faloutsos argue that a second correction step is useful in order to avoid errors introduced by the fast marching method used for level set reinitialization [MF06].

1.4 Flow Visualization

The remainder of this chapter is devoted to flow techniques, more specifically flow visualization and fluid simulation. Flow visualization handles the mapping of possibly 3-dimensional fluent phenomena to a possibly 2-dimensional visual output device. Its ambition is to visually expose interesting characteristics of the flow, and to provide explorative tools with which the user controls the scope of the visualization.

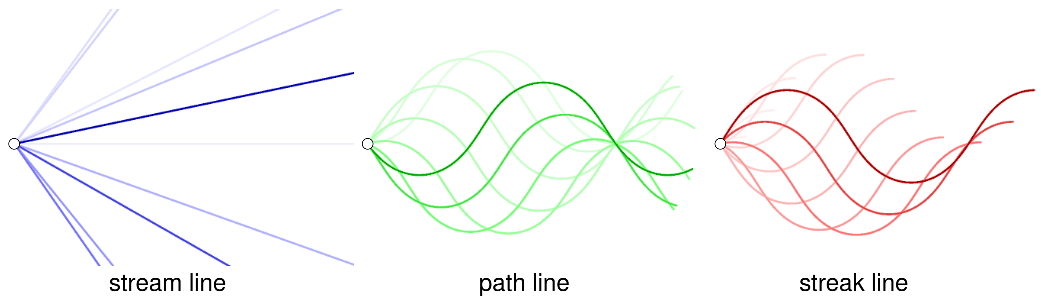


Figure 1.6: *Motion-blurred visualization of stream, path and streak lines at different instants in time. The following velocity field is used in this example:* $\vec{\mathbf{v}}(\mathbf{x}, t) = \big(cos\ (sin\ t), sin\ (sin\ t), 0\big)$.

Flow visualization is a well-studied topic (see [LHD*04] for one of several extensive overviews). State-of-the-art flow visualization can be classified in three main categories: Direct, texture-based and geometric visualization [Lar04]:

Direct visualization typically uses glyphs, e.g. arrows, which cover an overall picture of the flow. Flow information can be coded as a color or shape attribute.

Texture-based visualization offers a dense representation of the flow by advecting planar or volumetric grids. A widely used example is line integral convolution (LIC) [CL93], which applies convolution kernels to noise textures.

Geometric visualization evolves geometric objects like particles, lines, surfaces and volumes by integrating along the flow field. Stream lines consist of the paths formed by particles traced when halting the flow, thus reflecting the flow at one fixed instant of time. A path line shows how the flow is changing in time by tracing particles within an unsteady velocity field. Streak lines differ from the other line types in the fact that the whole line is moved instead of tracing one particle. They are natural in the sense that they frequently occur in the real world. For example, injecting a medium within a fluid (smoke in the wind) yields a streak line. These three flow line types are shown in Fig. 1.6. In principle, volumes can be used instead of lines in order to produce stream, path and streak volumes.

Various GPU-based flow visualization approaches were presented in the past. Most of the literature is related to dense representations, particularly LIC and similar texture-based techniques. Typically, the visual output is designed to reveal as much properties of the flow as possible, e.g. using multi-dimensional transfer functions (MDTF) [PBL*04]. The feature-based visualization by [PVH*02] extracts and visualizes flow features directly. The Lagrangian-Eulerian advection scheme by Jobard et al. [JEH01] produces a dense visualization of time-dependent vector fields which resembles dense particle representations. Weiskopf et al. propose a hybrid particle and texture based technique for unsteady 2D flow data on uniform grids [WSEE05]. Here particle tracing generates a visualization using a LIC together with radial basis functions, thus particles indirectly contribute to the final image. The system, which is interactive, has been applied to 2D climate flow data.

Geometric methods, on the other side, are usually based on explicit particle traces. In the last years, a lot of effort has been made to exploit graphics hardware for real-time visualization. Building on GPU-based particle systems (see Sec. 1.2.3), Krüger et al. present an approach for 3D flow visualization, including the generation of stream lines and ribbons [KKKW05]. The work by Park et al. [PBL*05] discusses dense seeding of stream and path lines in steady and unsteady flows. Their technique supports MDTFs, and it is suited to graphics hardware.

1.5 Particle Fluids

The simulation of fluids (i.e. the field of computational fluid dynamics) is, on the one hand, important for a large set of physical applications like meteorology, oceanography, aerodynamics, and pneumatics – just to state a few ones, and on the other hand a very challenging task which has been intensively investigated for several decades. The interest for fluids within computer graphics community mainly comes from the effects and game industry. However, the general purpose use of graphics hardware opens the door to a strictly physically based simulation of fluids.

Most applications perform their simulation by solving the Navier-Stokes equations for incompressible fluids. In practice, a vast majority of systems solve the Navier-Stokes equations using a grid discretization. While the grid structure itself is beneficial because of its computational simplicity, incompressibility has to be enforced by relaxation techniques, which ensure mass conservation throughout grid cells [FM96]. Other approaches are based on particles. Considering each particle as an entity carrying one portion of the fluid mass, it is clear that mass conservation is given by the fact that the number of particles never changes. The SPH method (Smoothed Particle Hydrodynamics), which is summarized in Sec. 1.5.2, is one example of these particle-based approaches.

Thanks to the growing interest in fluids within the computer graphics community, several approaches focusing on interactivity and supporting graphics hardware have been presented. Krüger and Westermann have shown that effects like explosions, fireballs, smoke and fire can be simulated and rendered in real-time using graphics hardware [KW05]. The basic idea here is the extrusion of 2-dimensional simulation results to 3 dimensions, which significantly reduces complexity. Harris has worked on the general task of solving partial differential equations using the GPU, and he presented a specific method for simulating cloud dynamics in [HBISL03]. An interpretation of SPH for real-time, yet non-GPU-based fluids has been presented by Müller et al. [MCG03].

1.5.1 The Navier-Stokes Equations

The famous Navier-Stokes equations were formulated by Claude Navier and George Stokes in the 19$^{\text{th}}$ century. Since then, these equations have become a standard in the field of computational fluid dynamics, as they have shown to be a powerful model.

The fluid representation is based on a velocity field $\vec{\mathbf{v}}$, the fluid's density ρ and a pressure p. The Navier-Stokes equations for incompressible fluids consists of one equation assuring conservation of mass by imposing a divergence-free velocity field, and one equation formulating conservation of momentum. To simplify matters, a compact notation omitting the field location $\mathbf{x}$ is used for $\vec{\mathbf{v}}$, ρ and p.

$$\nabla \cdot \vec{\mathbf{v}} = 0 \tag{1.16}$$

$$\frac{\partial \vec{\mathbf{v}}}{\partial t} + \underbrace{(\vec{\mathbf{v}} \cdot \nabla \vec{\mathbf{v}})}_{\substack{\text{convective}\\ \text{term}}} = \underbrace{\frac{-\nabla p}{\rho}}_{\substack{\text{pressure}\\ \text{term}}} + \overbrace{\nu \nabla^2 \vec{\mathbf{v}}}^{\substack{\text{viscosity}\\ \text{term}}} + \overbrace{\vec{\mathbf{g}}}^{\substack{\text{external}\\ \text{acceleration}}} \tag{1.17}$$

where t is time, $\vec{\mathbf{g}}$ an external acceleration, e.g. gravitation, and $\nu = \mu/\rho$ is the kinematic viscosity of the fluid, defined by the viscosity constant μ. Note that, assuming Cartesian coordinates, the convective term $\vec{\mathbf{v}} \cdot \nabla \vec{\mathbf{v}}$ is often (e.g. [Sta99]) written as $(\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}}$ using the shorthand operator $(u, v, w)^{\mathrm{T}} \cdot \nabla = u(\partial/\partial x) + v(\partial/\partial y) + w(\partial/\partial z)$.

In literature, Eq. 1.16 is occasionally eased to describe general mass conservation rather than requiring divergence-freeness of $\vec{\mathbf{v}}$ (also compare with Eq. 1.10):

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{\mathbf{v}}) = 0$$

Müller et al. argue, that in Lagrangian approaches, a constant number of particles guarantees this formulation of mass conservation [MCG03]. Furthermore, particles move with the fluid, thus the convective term in Eq. 1.17 can be omitted, resulting in the following simplified version of Navier-Stokes defining a force field $\vec{\mathbf{f}}$:

$$\vec{\mathbf{f}} = -\nabla p + \rho \vec{\mathbf{g}} + \mu \nabla^2 \vec{\mathbf{v}}$$

1.5.2 Smoothed Particle Hydrodynamics

A Lagrangian reformulation of Navier-Stokes called smoothed particle hydrodynamics (SPH) was introduced by Gingold and Monaghan [GM77]. The SPH method models the dynamics of fluids based on particle motions, applying forces according to the Navier-Stokes equations. One of the main advantages, compared to grid-based approaches, is that only the region of interest, i.e. the fluid itself, is represented by particles, and that the volume in which the simulation is defined is principally unbounded. Müller et al. [MCG03] present an optimized software implementation allowing an interactive simulation for a few thousand particles.

Let $\mathcal{P}$ be the set of particles representing the fluid medium. The main concept of SPH is the usage of a *smoothing kernel* $W : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ distributing a quantity A_j into the neighborhood of a particle $j \in \mathcal{P}$ located at $\mathbf{p}_j \in \mathbb{R}^3$. The kernel can be seen as a weight $W(r, h)$ depending on a distance r and the radius of influence h of a particle. The quantity at an arbitrary location $\mathbf{x} \in \mathbb{R}^3$ is determined by the sum of all particle's contributions:

$$\overline{A}(\mathbf{x}) = \sum_{j \in \mathcal{P}} m_j \frac{A_j}{\rho_j} \cdot W(\|\mathbf{x} - \mathbf{p}_j\|, h), \tag{1.18}$$

where m_j is the mass of particle j. Smoothing the density ρ_j according to this equation reduces the right side to a weighted sum over each particle's mass:

$$\overline{\rho}(\mathbf{x}) = \sum_{j\in\mathcal{P}} m_j \cdot W(\|\mathbf{x} - \mathbf{p}_j\|, h) \tag{1.19}$$

This smoothed density $\overline{\rho}$ can be used to define the density $\rho_j = \overline{\rho}(\mathbf{p}_j)$ at particle location $\mathbf{p}_j$ when smoothing additional quantities according to Eq. 1.18. Using a normalized smoothing kernel, i.e. choosing a kernel W with integral equal to 1, asserts that $\overline{\rho}$ sums up to the total fluid mass:

$$\int W(\|\mathbf{x}\|, h)\ d\mathbf{x} = 1 \quad \Rightarrow \quad \int \overline{\rho}(\mathbf{x})\ d\mathbf{x} = \sum_{j\in\mathcal{P}} m_j$$

It is interesting to note that derivatives of field quantities affect the smoothing kernel only. This easiness in handling Laplace and gradient operators makes SPH such a powerful model for solving Navier-Stokes:

$$\begin{aligned} \nabla\overline{A}(\mathbf{x}) &= \sum_{j\in\mathcal{P}} m_j \frac{A_j}{\rho_j} \cdot \nabla W(\|\mathbf{x} - \mathbf{p}_j\|, h), \\ \nabla^2\overline{A}(\mathbf{x}) &= \sum_{j\in\mathcal{P}} m_j \frac{A_j}{\rho_j} \cdot \nabla^2 W(\|\mathbf{x} - \mathbf{p}_j\|, h) \end{aligned}$$

The resulting pressure force $\vec{\mathbf{f}}_\mathrm{p}$ and the viscosity force $\vec{\mathbf{f}}_\mathrm{v}$ for particles i are deduced from the Navier-Stokes equations (more details in Müller et al. [MCG03]) in order to satisfy symmetry requirements:

$$\begin{aligned} \vec{\mathbf{f}}_\mathrm{p}(\mathbf{p}_i) &= -\sum_{j\in\mathcal{P}} m_j \frac{p_i + p_j}{2\rho_j} \cdot \nabla W(\|\mathbf{p}_i - \mathbf{p}_j\|, h), \qquad (1.20) \\ \vec{\mathbf{f}}_\mathrm{v}(\mathbf{p}_i) &= \mu \sum_{j\in\mathcal{P}} m_j \frac{\vec{\mathbf{v}}_j - \vec{\mathbf{v}}_i}{\rho_j} \cdot \nabla^2 W(\|\mathbf{p}_i - \mathbf{p}_j\|, h), \end{aligned}$$

where $\vec{\mathbf{v}}_i$ is the velocity of particle i and $p_j = k(\rho_j - \rho_0)$ is the pressure with gas constant k and rest density ρ_0. In a case where only two particles interact, the symmetrization ensures that both particles exert exactly the same force to each other.

In order to model the surface tension, Müller et al. [MCG03] use a so-called color field $\overline{c}$, which is 1 at particle locations and 0 everywhere else. The gradient of the color field describes the inward normal field of the fluid, whereas the divergence of the normal field is a measure for the curvature κ of the fluid surface:

$$\begin{aligned} \overline{c}(\mathbf{x}) &= \sum_{j\in\mathcal{P}} \frac{m_j}{\rho_j} \cdot W(\|\mathbf{x} - \mathbf{p}_j\|, h), \qquad (1.21) \\ \vec{\mathbf{n}}(\mathbf{p}_i) &= \nabla\overline{c}(\mathbf{p}_i), \quad \kappa = -\frac{\nabla^2\overline{c}}{\|\vec{\mathbf{n}}\|} \end{aligned}$$

Based on the color field $\overline{c}$, the surface traction force $\vec{\mathbf{f}}_{\mathrm{s}}$ is defined using the normal and the curvature:

$$\vec{\mathbf{f}}_{\mathrm{s}}(\mathbf{x}) = \begin{cases} -\sigma \cdot \nabla^2 \overline{c}(\mathbf{x}) \cdot \frac{\vec{\mathbf{n}}(\mathbf{x})}{\|\vec{\mathbf{n}}(\mathbf{x})\|} & \|\vec{\mathbf{n}}(\mathbf{x})\| > \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{1.22}$$

where σ specifies the tension coefficient. A threshold $\epsilon > 0$ is used in order to avoid numerical problems in the case where $\|\vec{\mathbf{n}}(\mathbf{x})\| \approx 0$, which appears for points not at the boundary.

Müller et al. propose to use different polynomial smoothing kernels for the computation of the different forces. In most cases, they use the kernel W_{poly6}:

$$W_{\text{poly6}}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

For pressure, the kernel W_{spiky} which was introduced by Desbrun and Cani in their work about deformable bodies [DC96]:

$$W_{\text{spiky}}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

For viscosity, Müller et al. designed the kernel $W_{\text{viscosity}}$, whose Laplacian is artificially changed in order to increase numerical stability:

$$\begin{aligned} W_{\text{viscosity}}(r,h) &= \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r < h \\ 0 & \text{otherwise} \end{cases} \\ \nabla^2 W_{\text{viscosity}}(r,h) &= \frac{45}{\pi h^6}(h - r) \end{aligned} \tag{1.23}$$

In their main example, Müller et al. are able to simulate a fluid consisting of more than 2,000 particles at 20 frames per second using a 1.8 GHz Pentium CPU. The next chapter aims for reformulating the approach to be suited for the GPU, with the intention to accelerate the simulation by processing the particles in parallel.

Chapter 2

Real-time Particle Fluids

Everything flows, nothing stands still.
Heraclitus of Ephesus, Greek philosopher

This chapter describes an approach for simulating SPH-based fluids in real-time. It is designed to exploit the capabilities of modern graphics hardware: all steps of the proposed algorithm are completely covered by the GPU. One of these steps – solving the so-called n-nearest neighbors problem – is a crucial challenge for any particle engine involving inter-particle effects. It requires each particle to have access to the information of surrounding particles within a specific radius. In Sec. 2.1, an implicit solution to this problem is proposed, which means that the overall neighborhood influence is accessible rather than determining explicit particle identifiers of all neighbors. This solution is introduced as the more general idea of *grid-particle* interchange by scattering 3-dimensional point markers into a texture. This way of processing particle neighborhoods was, to my knowledge, presented in our work [KC05] for the first time. Since then, the idea has been developed by numerous authors, amongst others Harada et al. [HKK07b]. At the end of the chapter (Sec. 2.2), Harada's explicit solution to the n-nearest neighbors problem is discussed, as it constitutes a significant improvement w.r.t. numerical stability of the simulation.

The approach presented in Sec. 2.1 has been published in [KC05].

Two aspects make fluid simulation difficult: For a reasonable degree of realism, a profound physical background is essential. On the other hand, simulators necessitate efficient algorithmic concepts in order to cope with the high computational complexity. As motivated in earlier chapters, particle-based algorithms can be formulated to work on a parallel machine. However, any application involving particle coupling, e.g. SPH-based fluids, the simulation of granular media or simple particle-particle collisions, requires a solution of the n-nearest neighbors problem. This is difficult, as particles are uncoupled by nature, i.e. a spatial arrangement is not implicitly given. Some approaches use a sorting of the particles in order to reflect the spatial relation in the particle stream [KSW04]. However, because of the one-dimensionality of sorting, neighborhoods cannot be reconstructed completely in all cases. The approach by Hegeman et al. accelerate the computation of inter-particle forces by computing a quadtree structure partitioning the volume [HCM06]. The method has to cope with the problem that the tree quickly degrades when particles move during simulation. Thus, a costly optimization of the tree must be done before each simulation step.

The key idea presented in the following section is to transfer the particle neighborhood information into a grid, which is then sampled in a second pass during the simulation. For an efficient realization of this idea on the GPU, a special technique for grid-particle interchange is necessary, which is the major contribution of this chapter. Our work, published in [KC05], constitutes the first brick of one category of GPU-based SPH solvers by introducing texture-based accumulation of particle neighborhoods. This is reflected in the popularity this kind of simulators has gained in the recent past [HKK07b, ZSP08, Har07, Gre08]. However, it should be stated clearly that this first version suffers from numerical problems due to grid discretization, and it is the merit of Harada et al. to have significantly improved the approach in order to get accurate results while increasing efficiency [HKK07b].

2.1 Smoothed Quantity Accumulation

This section presents the original GPU-based grid-particle SPH solver which has been published in [KC05]. First, the general idea behind the method is discussed, then technical aspects of the accumulation of particles into a 3D texture are discussed. Afterwards, an algorithmic overview of the SPH solver is given.

Accumulating Particle Contributions

The smoothing equation, Eq. 1.18, maps a quantity from the particle representation into a field location. The sum can be seen as collecting all particles, which may suggest the gathering scheme when thinking about porting it to the GPU. A naive implementation would just iterate through all particles and compute the sum exactly as described in Eq. 1.18. However, this would lead to a quadratic runtime, as each pair of particles would be involved in the computation of one simulation step. The proposed method goes the inverse way: each particle scatters its contribution (i.e. one summand in Eq. 1.18) to all locations within its influence radius. The scattered information is stored into a 3D texture discretizing the volume of interest. This texture can then be accessed using trilinear interpolation during simulation. An example how this texture looks like is shown in Fig. 2.1. It is clear that this scheme is no more quadratic – it is linear in the number of particles and mostly depends on the influence radius (the smoothing kernel size) and the resolution of the texture.

The accumulation step requires an efficient technique for scattering particle contributions, as described in the next section. Besides, the scattering is carefully designed to carry the proper computation of pressure, viscosity and tension forces. Special attention is necessary for the viscosity force (see Eq. 1.20), as it is particle-centric, i.e. involving $\vec{\mathbf{v}}_i$, rather then relying only on smoothed quantities.

2.1.1 Scattering Particle Contributions

Scattering into a 3D texture is difficult to achieve, because a render pass, i.e. the traditional way to start a GPU computation, always writes the output into a frame

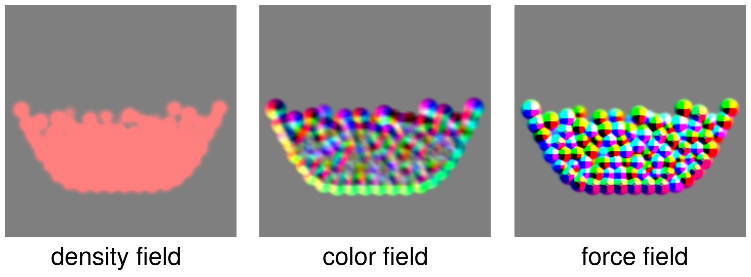


Figure 2.1: *Fields holding specific smoothed quantities. The picture shows one slice of the 3D volume. The contribution of each particle to its local neighborhood is accumulated by scattering blended particle markers. The color (rendered with a bias of* 0.5*) is the result of the mapping of the physical quantity to the RGB color channels (density field: ρ to R, color field: $\vec{\mathbf{n}}$ to RGB, (partly) force field: $\vec{\mathbf{f}}_p + \vec{\mathbf{f}}_v^{sep}$ to RGB).*

buffer, or a transform feedback stream. Writing to a 3D buffer is something highly non-traditional.

3D Scattering

There exist two efficient ways to realize 3D scattering: Flat 3D textures and a geometry shader controlled layered output.

Flat 3D textures constitute a way to represent 3D data as a tiled 2D texture [HBISL03]. According to the desired output slice, a vertex program selects the appropriate region in the 2D texture storing the slice. This is a simple mapping that can be implemented directly on GPU. The rasterizer then produces fragments at the right location. Sampling of this structure involves backward mapping and 2D texture fetching.

Layered output using the `OpenGL` extension `GL_NV_geometry_program4` supports direct selection of a specific 3D texture slice by setting a layer parameter in a geometry program.

Flat 3D textures can also be modeled using a stack of 2D textures (see Fig. 2.2), allowing larger texture sizes and MRT processing. The special arrangement of slices makes it possible to update sequences of four slices within only one MRT fragment program instance. Another convenient aspect is their backward compatibility to older hardware ($\geq$ GeForce 6). Additionally, due to their 2-dimensional representation, depth buffer based techniques, e.g. early-z culling, can be applied without any difficulty: A prepended render pass selects the desired restriction to a set of output fragments by masking the depth buffer. A second pass now processes only the fragments passing through the early-z test, thus avoiding a potentially expensive computation of irrelevant fragments. This early-z test is supported by NVidia hardware, if the fragment program does not alter the depth value.

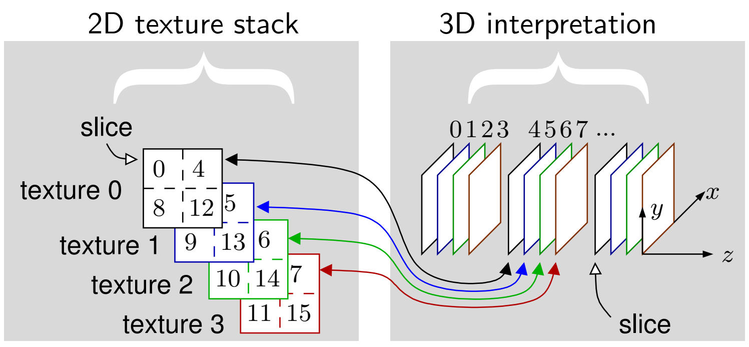


Figure 2.2: *A stack of 2D textures (left) interpreted as a single 3D output texture. This is a powerful and highly backward-compatible structure supporting 3D scattering. Note the spatial relation of slices within the four textures. This makes it easy to update four subsequent slices within a single program instance when using MRT processing.*

Compared to flat 3D textures, the layered output approach has the advantage that a "true" 3D texture is available, i.e. it supports trilinear interpolated fetching by nature.

Particle Markers

Independently of the scattering method chosen, it is obligatory to cut each 3D marker particle into a set of slices. Thus, one marker is constituted of several 2D points with a radius set to match the smoothing kernel size. An efficient way to render these points is to generate so called `OpenGL` *point sprites*, each moved to the appropriate location in the texture representation. Depending on the graphics hardware used, decomposing the point sprite into several single one-texel points can be faster. (The combination of point sprites and 32-bit floating point textures lead to problems on older GeForce cards.) After rasterization, the fragment shader computes one part of the sum in Eq. 1.18 and adds the result to the current location by using additive blending.

2.1.2 Algorithmic Concept

The GPU-based solver involves several computational steps which are listed in the following algorithm. The same specifiers as in Sec. 1.5.2 are used here. A particle i is assigned a position $\mathbf{p}_i$ and a velocity $\vec{\mathbf{v}}_i$.

Algorithm 2.1 (GPU-based SPH solver)

```
for each particle $i$ do
   **scatter** into density field $\overline{\rho}$                          // see Eq. 1.19
for each particle $i$ do
   **scatter** into color field $\nabla\overline{c}$ and $\nabla^2\overline{c}$ using $\overline{\rho}$    // see Eq. 1.21
for each particle $i$ do
```

```
 6     **scatter** into force field $\vec{\mathbf{f}}_\mathrm{p} + \vec{\mathbf{f}}_\mathrm{v}^\mathrm{sep}$ using $\vec{\mathbf{v}}_i, \overline{\rho}$     // Eq. 1.20 and Sec. 2.1.3
 7  for each particle $i$ at position $\mathbf{p}_i$ do {
 8     compute residual forces:
 9     **tension** force $\vec{\mathbf{f}}_\mathrm{s}(\mathbf{p}_i)$ using $\nabla\overline{c}(\mathbf{p}_i)$ and $\nabla^2\overline{c}(\mathbf{p}_i)$     // see Eq. 1.22
10     **viscosity** force $\vec{\mathbf{f}}_\mathrm{v}^\mathrm{part}(\mathbf{p}_i)$ using $\vec{\mathbf{v}}_i$     // see Sec. 2.1.3
11     combine all forces and compute a new velocity $\vec{\mathbf{v}}_i$
12     perform collision detection/reaction with a heightfield object
13  }
14  for each particle $i$ do {
15     move particle using a numerical integration scheme
16     // e.g. Euler or Verlet, see Sec. 1.2.2
17  }
18  go to 1
```

The fields, i.e. their texture representations, are chosen to supply the necessary number of scalar components to hold a quantity smoothed by the kernel itself, its gradient or Laplacian: The density texture contains $\overline{\rho}$, the color texture contains $\nabla\overline{c}$ and $\nabla^2\overline{c}$, and the force texture contains $\vec{\mathbf{f}}_\mathrm{p} + \vec{\mathbf{f}}_\mathrm{v}^\mathrm{sep}$ and the color $\nabla^2\overline{c}_\mathrm{v}$ using the viscosity kernel $W_\mathrm{viscosity}$.

Note that the viscosity force must be split up into two parts, $\vec{\mathbf{f}}_\mathrm{v}^\mathrm{sep}$ and $\vec{\mathbf{f}}_\mathrm{v}^\mathrm{part}$, which are computed in different steps. This is discussed in the next section.

2.1.3 Separating the Viscosity Force

From an abstract point of view, smoothing a quantity $\overline{A}$ according to Eq. 1.18 using the scattering method requires the sum to have *separable summands*. The color field $\overline{c}$ and the pressure force $\vec{\mathbf{f}}_\mathrm{p}$ are separable, since all necessary quantities are known or can be sampled in the density field. The viscosity force $\vec{\mathbf{f}}_\mathrm{v}$, however, is only defined at a particle position, since its velocities $\vec{\mathbf{v}}_i$ is involved.

In order to solve this problem, the viscosity force is partitioned up into a separable part $\vec{\mathbf{f}}_\mathrm{v}^\mathrm{sep}$ which is computed by scattering, and a residual $\vec{\mathbf{f}}_\mathrm{v}^\mathrm{part}(\mathbf{p}_i)$ computed during velocity update:

$$\begin{aligned}\vec{\mathbf{f}}_\mathrm{v}^\mathrm{sep} &= \mu \sum_j m_j \frac{\vec{\mathbf{v}}_j}{\overline{\rho}_j} \nabla^2 W(\|\mathbf{p}_i - \mathbf{p}_j\|, h),\\ \vec{\mathbf{f}}_\mathrm{v}^\mathrm{part} &= -\mu \vec{\mathbf{v}}_i \cdot \nabla^2 \overline{c}_\mathrm{v}(\mathbf{p}_i),\end{aligned}$$

where $\nabla^2\overline{c}_\mathrm{v}$ is a color field Laplacian constructed using the kernel $W_\mathrm{viscosity}$ (see Eq. 1.23) and stored into the fourth component during force field scattering in Alg. 2.1, line 6. This is necessary because the color field Laplacian used for the computation of tension forces is built using the kernel W_poly6, while the viscosity should be computed using the kernel $W_\mathrm{viscosity}$.

2.1.4 Surface Rendering

An interesting and convenient advantage of the accumulation into fields is the direct support for volume rendering approaches: The inner region Ω^fluid of the fluid can be

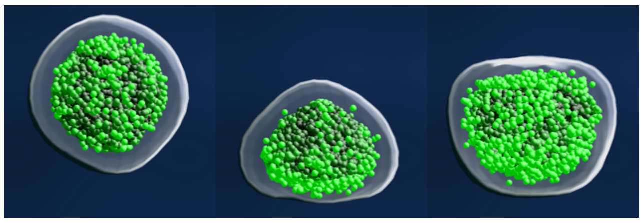


Figure 2.3: *A water ball modeled using strong tension forces and low gravity. This example shows how the color field can be used for rendering a transparent iso-surface with Phong lighting.*

defined as the set of locations with high color value, i.e. larger than a threshold θ defining the fluid's iso-surface:

$$\Omega^{\text{fluid}} = \left\{ \mathbf{x} \in \mathbb{R}^3 \mid \overline{c}(\mathbf{x}) \geq \theta \right\}$$

Moreover, a volume renderer can interpret the color field $\overline{c}$ as normal information. The only problem here is that gradients vanish near particle centers. Thus, the iso-surface threshold θ must be chosen so that no zero-normals are sampled. An example how the output looks like is given in Fig. 2.3.

2.1.5 Boundary Treatment

In the presented system, the bounding object (e.g. a cup or a glass) is modeled using a heightfield given as a height+normal representation, i.e. $H : \mathbb{R}^2 \to \mathbb{R}$ and $\hat{\mathbf{n}} : \mathbb{R}^2 \to \mathbb{R}^3$. The collision between a fluid particle and the boundary is detected by comparing the height and the current particle position $\mathbf{p}_i$. When a particle breaks through the heightfield, a collision reaction is performed by projecting the particle's velocity $\vec{\mathbf{v}}_i$ to a new velocity $\vec{\mathbf{v}}_i'$ touching the object:

$$\vec{\mathbf{v}}_i' = \begin{cases} \vec{\mathbf{v}}_i - \left(\hat{\mathbf{n}}(\mathbf{p}_i^{xy}) \cdot \vec{\mathbf{v}}_i\right) \cdot \hat{\mathbf{n}}(\mathbf{p}_i^{xy}) & \text{if } \mathbf{p}_i^z \leq H(\mathbf{p}_i^{xy}) + \epsilon \\ \vec{\mathbf{v}}_i & \text{otherwise} \end{cases}$$

The offset ϵ must be chosen to avoid situations where particles traverse the object's boundary. The texture can be stored as a 2-dimensional texture with four components. Sampling should be done using linear interpolated texture fetch.

2.1.6 Evaluation

The simulator has been tested on a PC system comprising an NVidia GeForce 9600 GT graphics card with 512 MB of memory. In our implementation, 32-bit floating point textures are used for the particle information and for the fields. The latter depend on blending and linear interpolation when writing and reading the field information.

The runtime of the SPH solver has been evaluated with the example shown in Fig. 2.4. The results are plotted in Fig. 2.5. In the example, the emitter configuration is chosen to produce realistic results: a starting volume of size r^3, which is

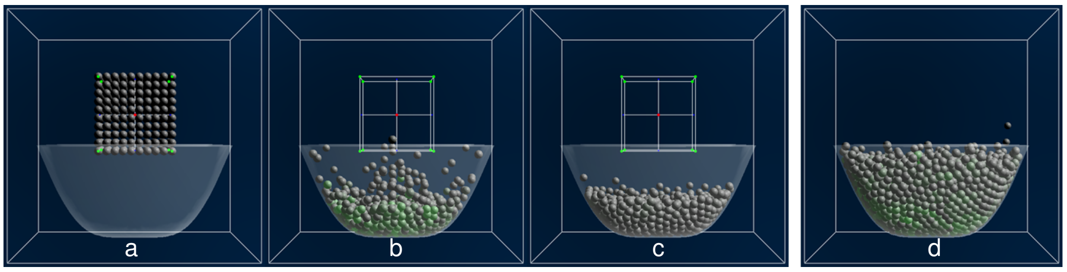


Figure 2.4: *Water falling into a cup. Part a to c: A fluid volume consisting of 1000 particles. Part d: The initial fluid volume has been emitted four times, resulting in 4000 particles.*

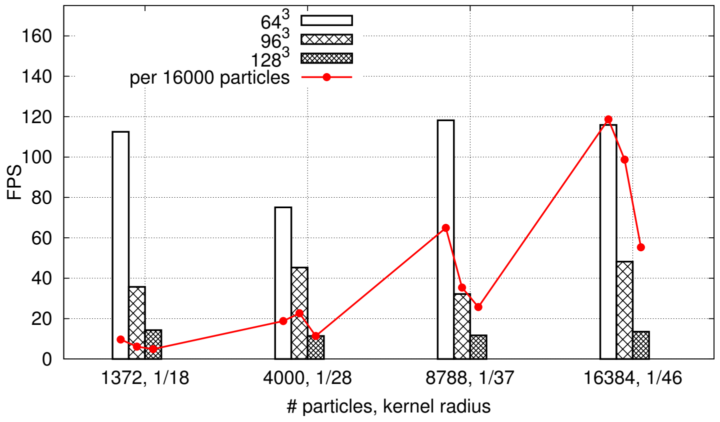


Figure 2.5: *Smoothed quantity accumulation: the performance for a fluid of varying granularity. The starting volume is emitted within the same bounding box (compare with Fig. 2.4). Thus, the kernel radius is chosen to match the initial rest state in this box.*

instantiated four times during the test, falls into a cup. The cup walls are defined by a heightfield used for collision detection and reaction, as described in Sec. 2.1.5. The number of particles n has been increased without altering the emitter configuration. This implies a higher particle density and thus a smaller smoothing kernel radius $h = r/\sqrt[3]{n}$. In the following, the term *granularity* is used to describe how dense the particles represent the medium (high granularity means a small kernel radius). An interesting aspect that can be observed in the diagram is that higher granularity does not necessarily imply a lower performance. On the contrary: The red line, which shows the frame rate scaled to a constant number of 16,000 particles, reveals that the per-particle performance is increasing. This is due to the fact that the size of point markers rendered into the fields is the main bottleneck of the approach. Thus, smaller particle radii may result in better performance despite the higher number of particles. On the other hand, note that the resolution of the grid must be high enough to capture smoothed quantities during the simulation. The example shown in Fig. 2.4, i.e. using a particle radius of 1/18 of one field dimension, works well with a 64^3 grid. Changing the granularity to 1/28 causes instabilities at this resolution, thus an increase to 96^3 is necessary. Accordingly, higher granularities require even higher resolutions. This high demand on grid resolution together with the numerical problems resulting from the grid discretization is the main problem of the approach.

2.2 Particle Index Accumulation

The approach proposed in Sec. 2.1 suffers from numerical problems due to the discretization of smoothed quantities. In order to avoid this problem, Harada et al. have ameliorated the approach by accumulating particle indices instead of physical attributes in the grid [HKK07b]. This provides a significant improvement in both efficiency and accuracy while being based on a similar idea, i.e. the idea of coupling particles by use of a grid structure. Consequently, Harada et al.'s approach (which we call *particle index accumulation*) is presented in the following. Our implementation of the approach is evaluated as well.

2.2.1 An Explicit Solution to the n-nearest Neighbors Problem

Provided that the fluid to be simulated is incompressible, the number of particles within a fixed region never exceeds a constant limit. Accordingly, Harada et al. argue that in a grid cell of size h^3, where h is the SPH kernel radius, at most four particles can be arranged. Consequently, a grid covering the volume of interest with an equidistant cell-spacing of h is sufficient for storing the complete particle information, supposed that one cell can hold four particles. The complete neighborhood of one particle can then be sampled by checking all surrounding grid cells, which corresponds to a $3 \times 3 \times 3$ sampling kernel. This furnishes all other particles within the influence radius, thus all particles contributing to the sum in Eq. 1.18. This concept is illustrated in Fig. 2.6.

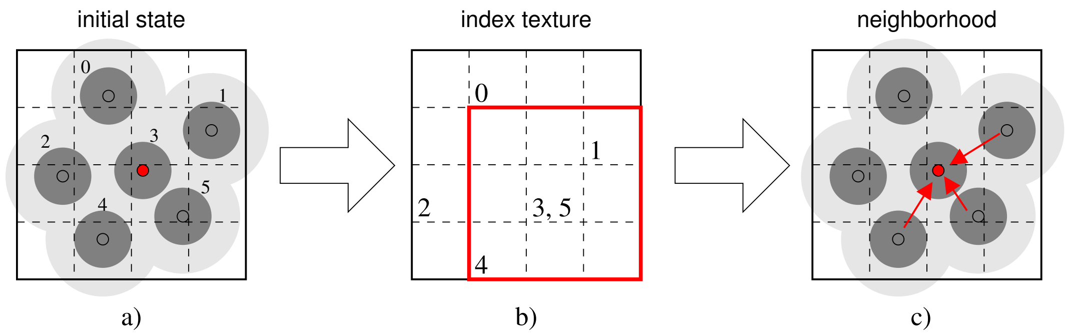


Figure 2.6: *The index accumulation approach (in 2D). a) Particles are enclosed in grid cells. b) The particle indices are stored into an index texture. The red box corresponds to the sample kernel around particle 3. c) neighbors have been found around particle 3 within the sampling kernel.*

The remaining challenge now consists in designing a GPU routine that scatters particle indices into a texture with four components. The original method by

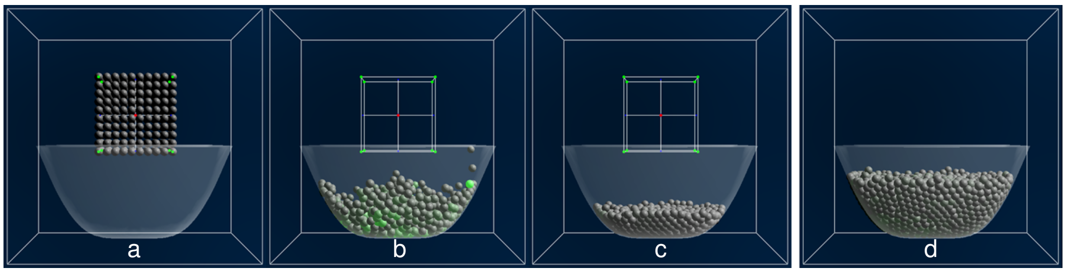


Figure 2.7: *Water falling into a cup. A similar scenario like in Fig. 2.4, this time using the index accumulation approach. Part a to c: A fluid volume consisting of 1000 particles. Part d: 4000 particles. After having poured four fluid instances, the fluid medium rapidly stabilizes towards a rest state.*

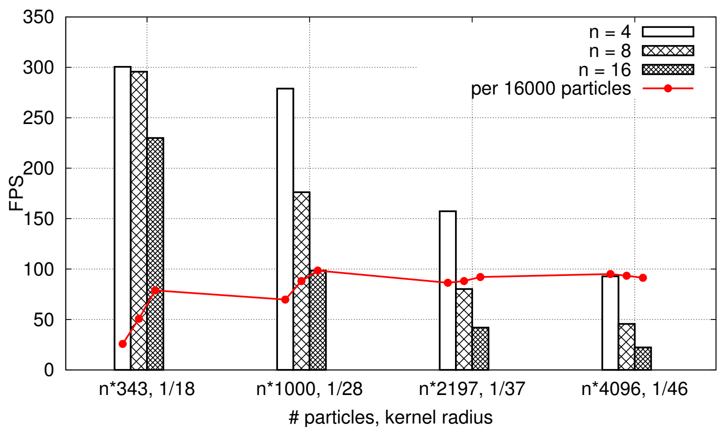


Figure 2.8: *Index accumulation: the performance for a fluid of varying granularity (compare with Fig. 2.7).*

Harada et al. use depth test in combination with a stencil test in order to write each particle index into the appropriate texel component. In summary, scattering is apportioned to four render passes accumulating indices in increasing index order. Each render pass steps from one component to the other by restricting rendering using an `OpenGL` color mask.

2.2.2 Evaluation

Our implementation of the index accumulation approach has been evaluated using the same example as for the smoothed quantity accumulation presented in Sec. 2.1. The result can be seen in Fig. 2.7. It is obvious that being independent of a grid discretization of the physical attributes yields a significantly better numerical stability. In particular, one has an exact minimum bound for the required grid size, as one grid cell dimension is set to be equal to the particle radius. This grid size is much smaller than the grid size necessary to capture forces in Sec. 2.1. Visually, two effects prove the superiority of index accumulation compared to smoothed quantity accumulation: The particles are calmer and reach a more compact rest state (compare Fig. 2.7 and Fig. 2.4), and the fluid's surface is smoother. Especially when choosing a high granularity, the index accumulation is clearly in advantage, because small particle radii are difficult to capture when accumulating smoothed quantities.

The runtime behaviour has been evaluated using the example shown in Fig. 2.7, which is identical to the one evaluated in the previous section. Accordingly, the frame rate has been determined for different granularities (see Fig. 2.8). Analyzing the diagram reveals that the performance is more or less linear w.r.t. the number of particles: The red line shows the performance, scaled to a constant number of particles. Its nearly constant behaviour shows that the grid size is not an essential bottle neck. The initial rise can be explained by the effect that for frame rates larger than 200 FPS, other factors like the framework application may influence the performance.

This chapter has proposed a GPU-based fluid approach combining a particle set and a grid structure. This kind of hybrid grid-particle processing has been used to solve the problem of accumulating particle neighborhoods. We will see in the next chapter, that a similar idea can be used to realize a GPU-based surface tracing based on the particle level set method.

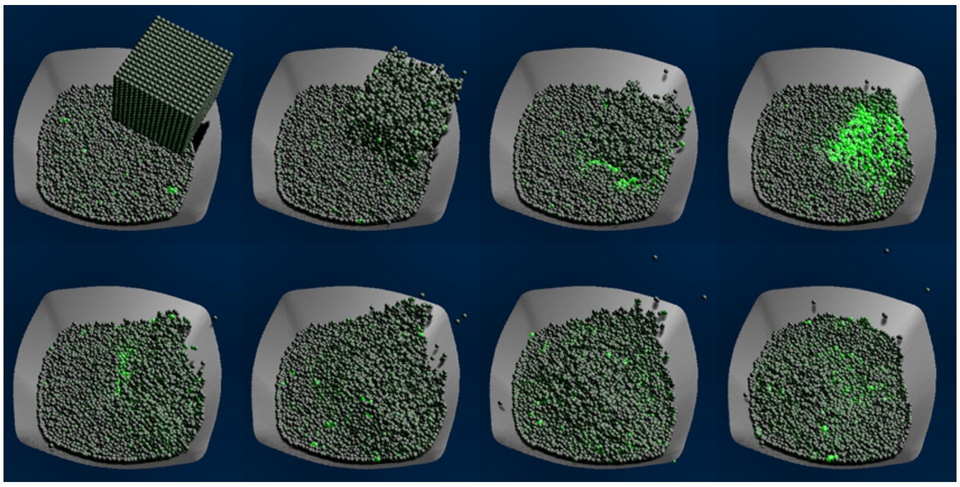


Figure 2.9: *A cup-of-water simulation using the hybrid grid-particle technique for coupling particles. The improvement proposed by Harada et al. [HKK07b], storing indices instead of forces in the grid, is used here.*

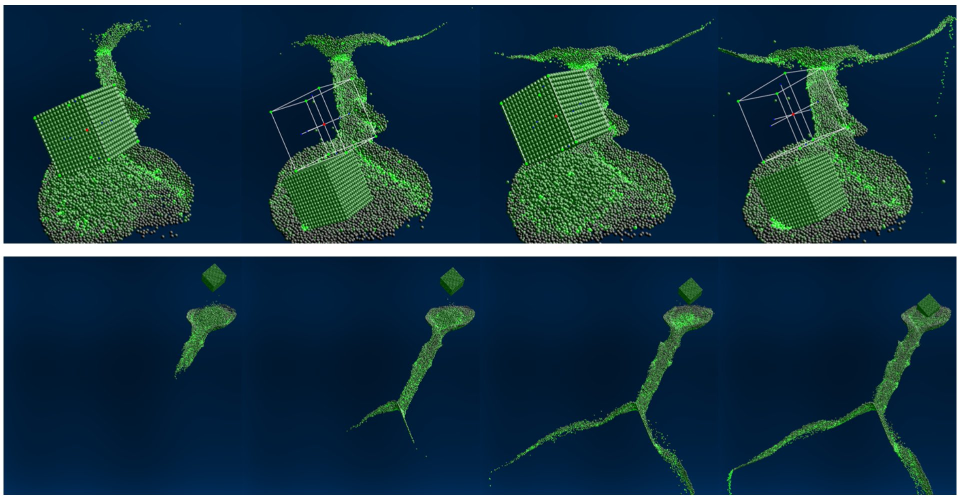


Figure 2.10: *A fluid in a branching canal using the same implementation as in Fig. 2.9.*

Chapter 3

Real-time Particle Level Sets

(...) *most of the world is of infinitely great roughness and complexity.*

Benoît Mandelbrot about shapes in nature

The focus of this chapter lies on a parallel surface tracing method suited for the GPU, and thus capable of real-time. This includes a modification of the PLS algorithm in order to make best benefit of the capabilities of graphics hardware. In order to accomplish this, the idea presented in Chap. 2 is evolved to an innovative way of exchanging information between the corrective particles and the level set grid representation. As the PLS method is a central aspect, it is first motivated in Sec. 3.1 by a discussion of the general problem of representing a free surface using a grid structure or a particle set. The level set reinitialization pass, which is one of the computationally most expensive tasks within the PLS algorithm, is then discussed in Sec. 3.2. Afterwards, the real-time PLS reformulation is presented and evaluated in Sec. 3.3. In the end of the chapter, Sec. 3.4 provides some supplementary examples.

The surface tracing method has been published in [CKSW08], the reinitialization strategy, i.e. the hierarchical DT algorithm, has been published in [CK07].

In practice, the level set approach suffers from a numerical diffusion due to the advection on a grid of finite resolution. Despite frequent reinitialization of the level set function, inaccuracies typically result in a volume loss after a certain number of advections. Fig. 3.1 demonstrates how this effect becomes noticeable even for a very basic example where an object representing a notched sphere is moved in a rotational flow. The first row shows the result of a semi-Lagrange advection (see Sec. 1.3.1) of an initially sound level set representation. While the interface is still recognizable after the last advection step, one can yet observe a smearing and a considerable volume loss during advection. The second row also uses semi-Lagrange advection, however involving a reinitialization pass after each step. This reduces the volume loss a bit, and it ensures that a proper level set representation is kept. However, the problem is still evident. The third row shows the results of the PLS method, which aims for annihilating the volume loss by embedding particles into the grid-based interface representation. One can observe that the initial shape is well preserved throughout the advection.

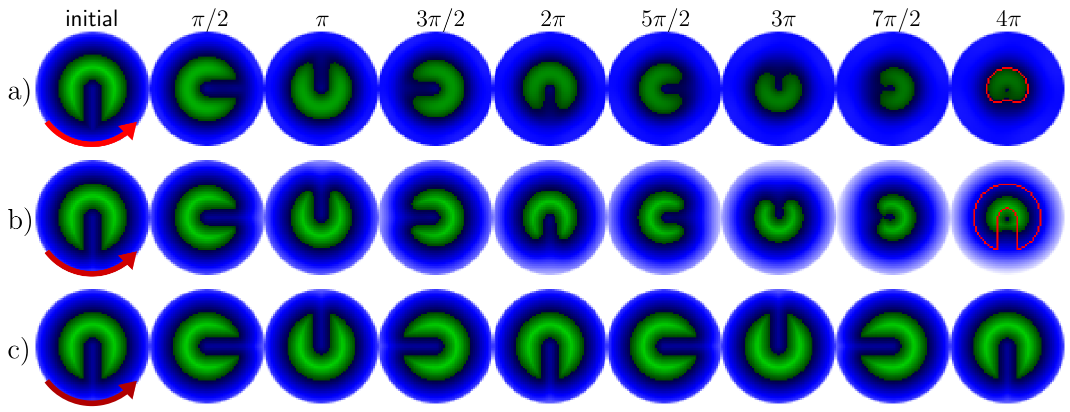


Figure 3.1: *Comparison of a) level set advection without reinitialization, b) with initialization and c) using the particle level set method. One slice of a 64^3 grid is shown. The PLS advection uses 262,144 particles. Negative values are green, positive values are blue. In order to compare the volume loss, the contour of the next row is overlaid in red in the last sub-image.*

3.1 Grid Advection vs. Particle Motion

The PLS method (see Sec. 1.3.4), which was introduced by Enright et al. [EFFM02, EMF02, ELF05], is highly motivated by the advantages of both purely grid-based and purely particle-based approaches: Grid-based semi-Lagrange advection is unconditionally stable, i.e. no artifacts occur, even when the flow causes topological changes, and independently of the time interval between two steps. Particle-based methods on the other side benefit from easy and high order integration schemes (see Sec. 1.2.2) allowing a very accurate tracing. Particles are ideal for capturing detailed features and transporting them within arbitrary flows. The idea of PLS is to use a dual grid- *and* particle-based representation of the interface while interchanging information from one representation to the other. This leads to a combination of both advantages.

As a matter of fact, PLS can also be motivated the other way around, by analyzing the problems related to purely grid-based and purely particle-based approaches: The first suffers from numerical diffusion, as we have seen earlier. The second, particle-based representations, suffer from the fact that the particle set tends to produce sparse or overpopulated areas in some cases. This problem manifests itself when the interface forms thin structures for which the particle density is too low, or when bunches of particles are accumulated in specific regions. In summary, PLS aims to prevent diffusion while preserving an even particle representation of the surface.

It is clear that PLS can highly benefit from being translated to a graphics hardware accelerated algorithm. The PLS algorithm is full of computationally expensive steps: Advection of the grid, numerical integration of particle velocities, level set correction, particle reseeding and particularly the level set reinitialization are all tasks that cannot be executed in real-time in a sequential CPU-based way for reasonably large data structures. However, each of these tasks require a careful design in order to be efficiently ported to the GPU. One of the most challenging of these tasks, the level set reinitialization, is discussed in the next section.

3.2 Fast Hierarchical Distance Transforms

The PLS method proposed in this chapter is driven by the challenge of performing all algorithm steps using the GPU. One of the most difficult steps is the reinitialization of the level set function, which can be interpreted as the task of constructing a grid-based distance field w.r.t. to the interface. Distance field construction is a well-studied problem (see Sec. 1.3.3). However, known algorithms, even though they may be suited for the GPU, are not fast enough to reinitialize larger grids ($> 128^3$) in real-time. In this section, an hierarchical propagation-based algorithm called FHA (fast hierarchical algorithm) is presented. It gives an alternative to the jump flooding algorithm (JFA) by Rong and Tan [RT06], being asymptotically faster while producing satisfactory approximate results. The algorithm is largely independent of the PLS context it will be later used for, thus it is presented as a separate component in this section. The FHA has been published in [CK07].

The fast hierarchical algorithm computes a distance transform on 3D grid input models, and it uses a propagation technique, both on a single hierarchy level and between the levels. Using the hierarchical approach, the effort to compute the DT is significantly reduced. It is well suited for applications that mainly rely on exact distance values close to the boundary. The technique is purely GPU-based, i.e. all hierarchical operations are performed on the GPU. A direct comparison with the JFA shows that the fast hierarchical algorithm is faster and provides better scaling in speed and precision, while JFA should be preferred in applications that require a more precise DT.

3.2.1 Prior Work

There exist previous methods for constructing DTs using the GPU. As discussed in Sec. 1.3.3, a DT can be derived from a Voronoi diagram directly. Consequently, GPU-based algorithm capable of computing a Voronoi diagram, e.g. the polygon rasterization approach by Hoff et al. [HKL*99], can be used for DTs. Improvements of this approach have been presented: Sigg et al. [SPG03] propose the construction of a signed 3-dimensional DT by scan-converting polyhedra. Sud et al. [SGGM06] present another GPU-based slice-wise construction of a 3D distance field including a specialized culling technique. However, these techniques are typically designed for mesh-defined objects, while we are interested to construct a DT starting from a grid-based initialization.

Strzodka and Telea [ST04] present a GPU-based DT framework for 2D grids using an arc length parametrization for the boundary. However, it generally does not carry over to the 3D case. In the special context of PLS, reinitialization (which can be seen as a distance field construction) is typically performed using the fast marching approach [Set96]. Unfortunately, no feasible way of translating this algorithm to the GPU is known. The jump flooding algorithm, which is described below, constitutes a suitable and efficient general algorithm for the construction of DTs. With the fast hierarchical algorithm presented later in this section, the intention is to provide an even faster, yet approximate alternative to JFA.

The Jump Flooding Algorithm

The JFA, introduced by Rong and Tan in 2006 [RT06], updates the whole grid using distance propagation steps (see Sec. 1.3.3) while varying the kernel $\mathcal{N}$ in order to propagate references across longer distances. In the k-th step, $\mathcal{N}$ maps a location $\mathbf{x} = (x, y, z)$ to a $3 \times 3 \times 3$ subgrid defined as:

$$\mathcal{N}(\mathbf{x}) = \Big\{(x + i_1, y + i_2, z + i_3) \mid i_1, i_2, i_3 \in \{-k, 0, k\}\Big\} \tag{3.1}$$

For a grid with resolution n^3, the JFA can be summarized as follows:

Algorithm 3.1 (jump flooding algorithm)

```
1 initialize the grid according to Eq. 1.12
2 for $k \in \{n/2,\ n/4,\ \ldots,\ 1\}$ do
3   reference/distance propagation for all cells using Eq. 3.1
```

The JFA makes $\log(n)$ loops over the whole grid. Rong and Tan prove various properties of the JFA [RT06], and they present extensions to the algorithm in order to improve the accuracy. The two JFA variants are JFA+j, meaning that j additional propagation steps with step size $2^{j-1}, \ldots, 1$ are performed, and JFA^2, where JFA is applied twice.

3.2.2 Algorithmic Overview

FHA uses a hierarchy of levels, each having half the previous resolution (compare with Fig. 3.2). Let us assume level 0 to be the grid of desired resolution, and being initialized with the starting configuration of **dt** according to Eq. 1.12. The algorithm now performs two iterations: a loop consisting of down-sampling steps called *push-downs*, and one consisting of up-sampling steps called *pull-ups*. During the push-down iteration, the grid resolution is reduced by factor 2 until level N is reached. A pull-up propagates the information back to a higher level of twice the resolution. Due to the staggered arrangement of the levels, a grid cell resulting from a push-down from level k to $k+1$ lies in-between eight neighbors of finer resolution (four in the 2D case shown in Fig. 3.2). Similarly, a pull-up combines eight surrounding neighbors of a coarser grid cell. Consequently, both push-down and pull-up can be realized using a $2 \times 2 \times 2$ filter kernel. A formal description of the push-down and pull-up filters is provided in the next two paragraphs.

During the push-down iteration, the DT gets coarser and more imprecise. A full reduction to a single cell is not meaningful, thus the number of levels is limited to $N < \log(n)$, where n^3 is the resolution of level 0. On the coarsest level N, a complete DT is computed by using a propagation kernel until all cells are reached, or by using JFA. Due to the low resolution, this is cheap compared to the rest of the algorithm.

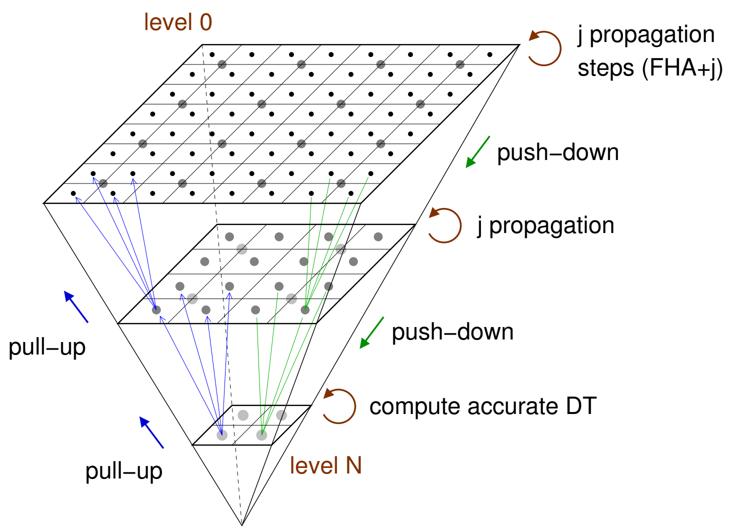


Figure 3.2: *The fast hierarchical algorithm uses temporary, staggered levels of lower resolution in order to reduce computational costs. The push-down and pull-up operations are illustrated in a 2D grid case.*

During the pull-up iteration, the DT regains detail, as the information of both the own level and a coarser level are combined. The fast hierarchical algorithm can be summarized as follows:

Algorithm 3.2 (fast hierarchical DT algorithm)

```
initialize the grid according to Eq. 1.12
for level $k = 0, \dots, N-1$ (push−down) do
  for all cells on the coarse level $k+1$ do
    propagate DT from level $k$ to level $k+1$ using a $2 \times 2 \times 2$ kernel
compute DT on coarsest level by repeating propagation steps or by using JFA
for level $k = N-1, \dots, 0$ (pull−up) do
  for all cells on the fine level $k$ do
    combine DT of level $k+1$ with level $k$ using a $2 \times 2 \times 2$ kernel
```

FHA usually incorporates j additional steps on each level during pull-up, yielding FHA+j, either using standard $3 \times 3 \times 3$ propagation steps (see Eq. 1.13) or JFA steps for $k = 2^{j-1}, \dots, 2, 1$. Clearly, FHA is an approximate algorithm. The left side of Fig. 3.3 shows a situation where an error occurs.

Push-down

The push-down step propagates the DT information from a level k to the coarser level $k+1$, i.e. from ${}^{k}\mathbf{dt}$ to ${}^{k+1}\mathbf{dt}$. However, the starting level contains only the boundary of the object for which a distance transform is constructed. Thus, figuratively, the push-down step transports the narrow band of information to a lower resolution. More formally, this is done by supersampling a neighborhood $\mathcal{M}_8$ of $2 \times 2 \times 2$

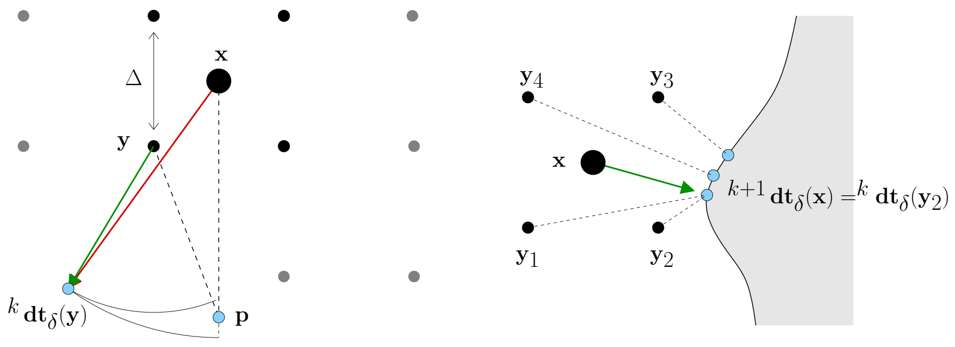

Δ
$\mathbf{x}$
$\mathbf{y}$
${}^k\,\mathbf{dt}_\delta(\mathbf{y})$
$\mathbf{p}$
$\mathbf{y}_4$
$\mathbf{y}_3$
$\mathbf{x}$
${}^{k+1}\,\mathbf{dt}_\delta(\mathbf{x}) = {}^k\,\mathbf{dt}_\delta(\mathbf{y}_2)$
$\mathbf{y}_1$
$\mathbf{y}_2$

Figure 3.3: *Left: A situation where an error is produced, because* $\mathbf{dt}_\delta(\mathbf{y})$ *is closer to* $\mathbf{y}$ *than* $\mathbf{p}$*, while* $\mathbf{p}$ *is closer to* $\mathbf{x}$ *than* $\mathbf{dt}_\delta(\mathbf{y})$*. Right: Push-down of neighbors to a sample in the coarse grid* ${}^{k+1}\,\mathbf{dt}$*. The closest reference point w.r.t. level* k *triggers the push-down to level* $k+1$*.*

surrounding cells in the higher resolution. The operation is very similar to traditional distance propagation. It is based on a comparison of reference points sampled within $\mathcal{M}_8$. For $\mathbf{x}$ in level $k+1$, we set

$$ {}^{k+1}\,\mathbf{dt}_d(\mathbf{x}) \;=\; \operatorname{sign}\left({}^k\,\mathbf{dt}_d(\mathbf{x})\right) \cdot \min_{\mathbf{y}\in\mathcal{M}_8(\mathbf{x})} \left\{ \left\| {}^k\,\mathbf{dt}_\delta(\mathbf{y}) - \mathbf{x} \right\| \right\} \qquad (3.2) $$

where the sign is sampled by choosing the nearest neighbor, and the reference component ${}^{k+1}\,\mathbf{dt}_\delta(\mathbf{x})$ is updated using the selected $\mathbf{y}$. Fig. 3.3, right side, shows an example of one single push-down operation. Note that cells initialized with $\pm\infty$ are properly handled in the next level, i.e. the reference is set for $\mathbf{x}$ if at least one $\mathbf{y} \in \mathcal{M}_8(\mathbf{x})$ has a valid reference.

Pull-up

For now, we assume that the correct distance transform ${}^N\,\mathbf{dt}$ for the coarse level is given. This is clearly the case for the coarsest level, because for this level, we calculate ${}^N\,\mathbf{dt}$ explicitly (see Alg. 3.2). The pull-up pass propagates the DT information from a level $k+1$ to the finer level k. Formally, the pull-up equation is the same as the one for push-down (Eq. 3.2) when substituting k for $k+1$ and vice versa. The reference point for a cell $\mathbf{x}$ in level k is determined by sampling and comparing eight surrounding samples in level $k+1$. Note that the distance information near to the object boundary coming from push-down is maintained during this step, i.e. only cells which contain $\pm\infty$ as distance are updated. This can be done because each level is stored separately.

As mentioned before, the result after a pull-up pass is only a rough approximation of ${}^k\,\mathbf{dt}$. To correct the error, two strategies are followed: First, FHA+j performs j additional propagation steps on each level. Second, the cells surrounding the reference point $\mathbf{p} = {}^k\,\mathbf{dt}_\delta(\mathbf{x})$ resulting from the pull-up step in the same level are used for a refinement, yielding ${}^k\,\mathbf{dt}'$:

$$ {}^k\,\mathbf{dt}'_d(\mathbf{x}) \;=\; \operatorname{sign}\left({}^k\,\mathbf{dt}_d(\mathbf{x})\right) \cdot \min_{\mathbf{y}\in\mathcal{M}_8(\mathbf{p})} \{ \left\| {}^k\,\mathbf{dt}_\delta(\mathbf{y}) - \mathbf{x} \right\| \}, $$

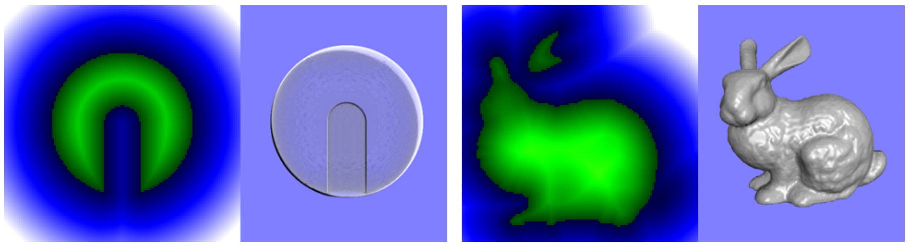


Figure 3.4: *Two examples visualized by rendering a single slice and by volume-rendering the iso-surface $\{\mathbf{x} \mid \mathbf{dt}_d(\mathbf{x}) = 0\}$. Left: a simple geometric object representing a notched sphere (transparent surface). Right: the Stanford bunny. The DTs were computed using FHA+2 on a 128^3 grid, performing push-down until 8^3 resolution. Colors: negative distances are green, positive distances blue. Runtime: ca. 28 FPS for both.*

where the reference component ${}^{k}\,\mathbf{dt}'_{\delta}(\mathbf{x})$ is updated using the selected $\mathbf{y}$.

Ideally, one would like a pull-up that generates ${}^{k}\,\mathbf{dt}$ from ${}^{k+1}\,\mathbf{dt}$ without any error, using a minimal number of propagation steps j on each level. Unfortunately, the optimal j is hard to determine, even though a constant error bound for ${}^{k}\,\mathbf{dt}$ can be given for the general situation.

3.2.3 Implementation

All steps of FHA can be implemented using fragment programs writing values to a floating point 3D texture. As in principle, no scattering is necessary, there is no disadvantage in fetching the DT information from a 4-component 3D texture. However, the intention is to use the DT in the context of PLS, where grid-particle interchange is involved. Consequently, we have chosen to represent the 3D grids as flat 3D textures (see Sec. 2.1.1). These support arbitrary 3D scattering even on older hardware.

3.2.4 Evaluation

FHA has been tested using implicit and predefined geometries stored as texture data. The graphics hardware used for the tests is a GeForce 8800 GTS. Fig. 3.4 shows the computed distance field for two examples.

Since FHA is proposed as a direct alternative to JFA, both algorithms have been evaluated and compared directly w.r.t. to runtime and accuracy. As intended, FHA is significantly faster than JFA, due to the reduction of the resolution within the level hierarchy. This performance advantage can be seen in Fig. 3.5. Note that jump flooding can never be faster than JFA+0, because JFA+0 performs the minimum of additional steps possible. Thus, the higher runtime is a true advantage of FHA – however, it is also clear that FHA produces more errors than JFA. This is discussed below. A runtime analysis of the fragment programs involved shows that the push-down and the pull-up are texture-fetch-bounded, while the distance propagation is computation-bounded.

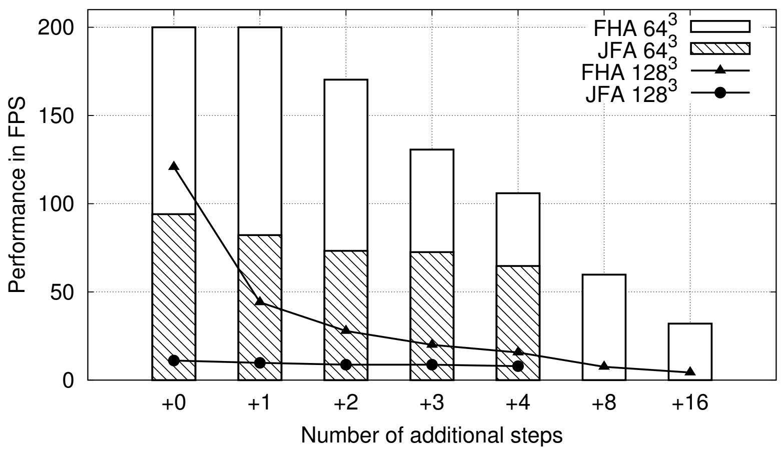


Figure 3.5: *The runtime in frames per second for JFA and FHA w.r.t. the number of additional propagation steps (object: notched sphere).*

In order to get an impression of the accuracy, the following quality measures are used: the relative number of wrong cells e_{cell}, the average distance error e_{avg} w.r.t the number of wrong cells, and the maximal distance error e_{max}:

$$e_{\text{cell}} = \frac{\#\text{ of wrong cells}}{\#\text{ cells}}, \quad e_{\text{avg}} = \frac{\sum \|\mathbf{dt}_d(\mathbf{x}) - \mathbf{dt}_d^*(\mathbf{x})\|}{\#\text{ wrong cells}},$$

$$e_{\text{max}} = \max\{\|\mathbf{dt}_d(\mathbf{x}) - \mathbf{dt}_d^*(\mathbf{x})\|\},$$

where $\mathbf{dt}^*$ is the correct DT, and the sum and the max symbol take all grid cells $\mathbf{x}$ into account. A *brute force* algorithm checking all pairs of cells, i.e. a quadratic approach, has been implemented in order to compute the exact DT. The quality tests have been performed on a 32^3, a 64^3 and a 128^3 grid using the notched sphere example shown in Fig. 3.4. The results are listed in Tab. 3.1.

When taking a large number of seeds (i.e. initialized cells) as starting situation, the number of wrong pixels (e_{cell}) as well as the maximum error is significantly larger for FHA compared to JFA. The quality of the error, however, is ambiguous. Taking the average error e_{avg} into account, it appears that it is, except for FHA+0, in the range of $2-4\%$ of a cell's size. When taking a sparse geometry as input featuring a few hundreds of seeds, then the faster FHA+2 provides better results than JFA(+0). For this test, a simple tetrahedron mesh which is subdivided repeatedly was used.

Finally, replacing the j additional propagations for FHA using a $3 \times 3 \times 3$ local neighborhood with the last j JFA steps yields a slight improvement in accuracy, i.e. e_{cell} is reduced up to 1% for the 64^3 grid.

In summary, FHA is faster (123 FPS for FHA vs. 11 FPS for JFA on a 128^3 grid) but less accurate than JFA for a large number of seeds. For a small number of seeds, the accuracy of FHA+2 is comparable to JFA. Depending on the application, the resulting errors can be acceptable and the performance advantage of FHA is possibly

Table 3.1: *Error evaluation comparing FHA and JFA. The relative number e_{cell} is given as a percentage. Note that for FHA+j, the number of additional steps j must be in $\{0, \ldots, \; log(n) - 1\}$ for a grid of resolution n.*

sphere, 64^3, 32457 seeds							
FHA	+0	+1	+2	+3	+4	+8	+16
e_{avg}	0.083	0.047	0.04	0.035	0.028	0.021	0.022
e_{max}	1.41	0.375	0.266	0.211	0.156	0.125	0.109
e_{cell} %	35.312	7.607	5.019	3.686	2.618	1.314	0.529
JFA	+0	+1	+2	+3	+4	–	–
e_{avg}	0.032	0.022	0.022	0.022	0.023	–	–
e_{max}	0.181	0.059	0.059	0.059	0.059	–	–
e_{cell} %	0.309	0.102	0.097	0.097	0.119	–	–

sphere, 32^3, 32457 seeds							
FHA	+0	+1	+2	+3	+4	+8	+16
e_{avg}	0.094	0.048	0.028	0.027	0.026	0.033	0.047
e_{max}	0.807	0.381	0.168	0.137	0.137	0.115	0.115
e_{cell} %	29.569	4.852	2.637	1.910	1.511	0.369	0.201
JFA	+0	+1	+2	+3	–	–	–
e_{avg}	0.039	0.033	0.037	0.037	–	–	–
e_{max}	0.125	0.041	0.045	0.045	–	–	–
e_{cell} %	0.128	0.012	0.024	0.024	–	–	–

subdivided tetrahedron, 64^3				
	number of seeds			
FHA+2	4	10	34	130
e_{avg}	–	–	–	–
e_{max}	0	0	0	0
e_{cell} %	0	0	0	0
	number of seeds			
JFA(+0)	4	10	34	130
e_{avg}	–	–	0.262	0.107
e_{max}	0	0	0.375	0.191
e_{cell} %	0	0	0	0.01

more important in order to achieve an overall system with interactive frame rates. An example is given by the level set method. In particular, the hybrid particle level set technique can benefit from the reference part of the DT for particle reseeding without requiring full accuracy.

3.3 GPU-based Particle Level Sets

In the previous section, a solution for computing distance transform has been proposed, which is suited for a fast level set reinitialization in the PLS algorithm (Alg. 1.1). This section focuses on translating the remaining steps to the GPU. This includes important aspects like the level set correction and particle reseeding. In addition, it should be stated that all steps are designed to work together as efficiently as possible. Consequently, the contribution of this section is a geared system containing all ingredients to implement an efficient GPU-based PLS framework. The new method achieves both, higher performance *and* superior quality in terms of volume preservation, compared to the public CPU-based reference implementation by Mokberi and Faloutsos [MF06]. The method has been published in the context of flow volumes [CKSW08].

3.3.1 Prior Work

While, to my knowledge, no GPU-based PLS method has been presented before the one presented here, particle systems and grid-based numerical schemes on their own have been efficiently mapped to GPUs. For GPU-based particle systems, see Sec. 1.2.3. A first GPU-based implementation of the level set equation was presented by Rumpf and Strzodka [RS01]. It uses intensity and gradient based forces to drive a segmentation of 2D images. Lefohn et al. additionally incorporated a curvature term and an adaptive memory model for the narrow band technique and thus could run efficient segmentation in 3D [LKHW04]. In these applications, the numerical diffusion of the level set advection is not a problem. On the contrary, a smooth boundary of the segmented 3D region is desirable and the curvature term is used exactly for this purpose. An iterative solution to the level set equation was presented by Griesser et al. [GRNG05].

3.3.2 Data Structures

The fast hierarchical algorithm (see Sec. 3.2), which is used for level set reinitialization, produces a DT. On the one hand, the use of a DT is motivated by the propagation-based reinitialization itself, on the other hand, the references stored in the DT can be used for a simple yet efficient particle reseeding (which will be discussed in Sec. 3.3.4 and Sec. 3.3.5) with subvoxel accurate radii. This advantage compensates the higher memory consumption for storing the reference data.

The DT $\mathbf{dt}(\mathbf{x}) = (\mathbf{dt}_d(\mathbf{x}), \mathbf{dt}_\delta(\mathbf{x}))$ is stored in a grid. For each cell $\mathbf{x}$, the signed distance $\mathbf{dt}_d(\mathbf{x}) = \phi(\mathbf{x})$ is equal to its level set value, and $\mathbf{dt}_\delta(\mathbf{x})$ is a reference to the nearest interface location. Note that in some stages of the algorithm, e.g. after the level set advection, the grid may not represent a precise or complete distance transform. Internally, $\mathbf{dt}$ is represented as a 4-component flat 3D floating point texture. Frame buffer objects are used for render-to-texture functionality. Double-buffering separates the input from the output. The velocity field defining the flow for level set advection can be given in any form that allows arbitrary sampling, e.g. in

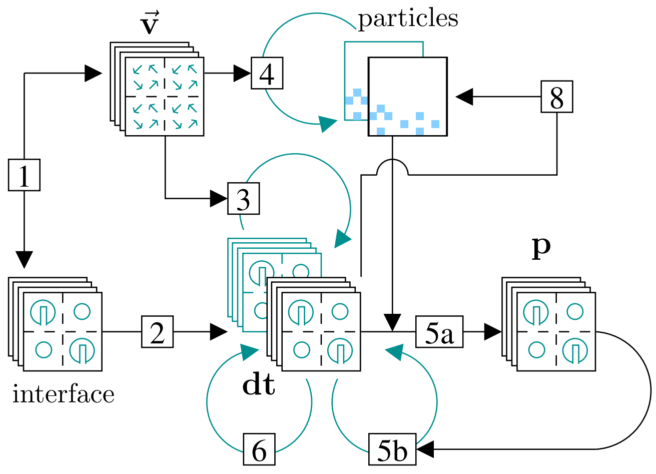


Figure 3.6: *The data flow in the GPU-based PLS framework. The use of double buffering is marked by cyclic, blue arrows. The numbers indicate the according steps in Alg. 1.1. (Step 7 is omitted.)*

a 3D texture. The particle system is held in a 2D floating point texture (preferably in 32-bit format), storing the position $\mathbf{x}_p$ and the radius r_p of each particle p.

The overall structure of the original PLS algorithm (Alg. 1.1) is maintained in the GPU-based framework. A diagram of the data flow using the GPU-based structures and containing the steps of the PLS algorithm is given in Fig. 3.6. In- and outgoing edges represent data input and output, respectively. The error correction (step 5) involves two passes and is split into two separate entities 5a and 5b. The repeated particle correction (step 7) is omitted in order to improve performance with negligible loss of overall accuracy.

3.3.3 Level Set Reinitialization

This section briefly discusses how FHA is used to reinitialize the level set (Alg. 1.1, steps 2 and 6). Before reinitialization, **dt** contains only the interface, which is given by means of the grid-based classification (interior/exterior) or by a gray-level function (e.g. the advected $\mathbf{dt}_d$, defining the interface on a subvoxel level). In particular, the reference component $\mathbf{dt}_\delta$ contains no valuable information.

Now, the interface needs to be identified, i.e. references are assigned to grid cells in direct proximity of the interface. Without this information, it would be impossible to propagate the interface. Those cells are identified by searching adjacent cells $\mathbf{x}$ and $\mathbf{y}$ with a different sign in the distance: $\text{sign}(\mathbf{dt}_d(\mathbf{x})) \neq \text{sign}(\mathbf{dt}_d(\mathbf{y}))$. This check is performed for each cell $\mathbf{x}$ in a fragment program, where $\mathbf{y}$ is a cell in a 6-neighborhood of $\mathbf{x}$ containing all adjacent non-diagonal neighbors. If one cell $\mathbf{y}$ is found, a new reference $\mathbf{dt}_\delta(\mathbf{x})$ is computed by moving along the local gradient $\hat{\mathbf{n}}$ at position $\mathbf{x}$:

$$\mathbf{dt}_\delta(\mathbf{x}) \longleftarrow \mathbf{x} - \mathbf{dt}_d(\mathbf{x}) \cdot \hat{\mathbf{n}}$$

The gradient is computed by central differences in the 6-neighborhood of $\mathbf{x}$ within the *old* field $\mathbf{dt}_d$. The *new* level set value $\mathbf{dt}_d(\mathbf{x})$ is set to the distance between $\mathbf{x}$ and $\mathbf{dt}_\delta(\mathbf{x})$, multiplied by the sign previously stored at position $\mathbf{x}$. The subvoxel references in $\mathbf{dt}_\delta(\mathbf{x})$ can now be used for the reinitialization of $\mathbf{dt}(\mathbf{x})$ using FHA.

3.3.4 Particle Reseeding

For an effective PLS correction, all particles should be located near the interface. This, however, is difficult to achieve in parallel on the GPU, since one cannot easily iterate over the interface proximity and place particles accordingly. Moving the particles to the interface is not only necessary during the initialization in the beginning, but also occasionally during the execution of the algorithm, because more and more particles will drift away from the interface with time evolving. Frequent particle reseeding has the negative side effect that inaccuracies in the grid are constantly transferred into the new set of particles, thus annihilating the advantage gained by the particle correction. According to Enright et al. [EMF02], a reasonable trade-off is to reseed the particles every 20 time steps on average, e.g. to reposition 5 percent of all particles in each iteration.

Due to the level set representation as a DT, the nearest interface location is known at any volume position. First, the initial location $\mathbf{x}_p$ for each particle p is chosen randomly within the volume of interest, i.e. $[0,1]^3$. Afterward, the particle is pushed towards the interface by following the reference and adding a random offset:

$$\mathbf{x}_p \;\longleftarrow\; \mathbf{dt}_\delta(\mathbf{x}_p) + \epsilon \cdot \hat{\mathbf{v}}_{\mathrm{rand}},$$

where $\hat{\mathbf{v}}_{\mathrm{rand}}$ is a normalized random vector pointing in an arbitrary direction and ϵ is a predefined small constant scalar value. This way, inner and outer particles surrounding the interface are produced. In our implementation, conforming with [ELF05], ϵ is chosen such as to place particles around the interface within a band that is a few grid cells wide (e.g. three cells). The random vector can be fetched from a texture with precomputed random values.

Pushing the particles towards the interface according to the above scheme can lead to sparsely populated regions, especially in areas with high curvature. In the experiments, this effect could always be sufficiently reduced by using a higher number of particles, as the surface represented by the level set is bounded by the number of discrete grid cells.

3.3.5 Setting a Particle's Radius

During reseeding, the radius r_p of each particle p must be determined. Two methods for radius sampling have been tested. The first one samples the distance stored in $\mathbf{dt}_d(\mathbf{x}_p)$ using trilinear interpolation. The other approach looks for the nearest reference in a neighborhood $\mathcal{M}_8$ consisting of 8 grid cells around $\mathbf{x}_p$:

$$r_p \;\longleftarrow\; s_p \cdot \min_{\mathbf{x}' \in \mathcal{M}_8(\mathbf{x}_p)} \{ \| \mathbf{dt}_\delta(\mathbf{x}') - \mathbf{x}_p \| \}, \tag{3.3}$$

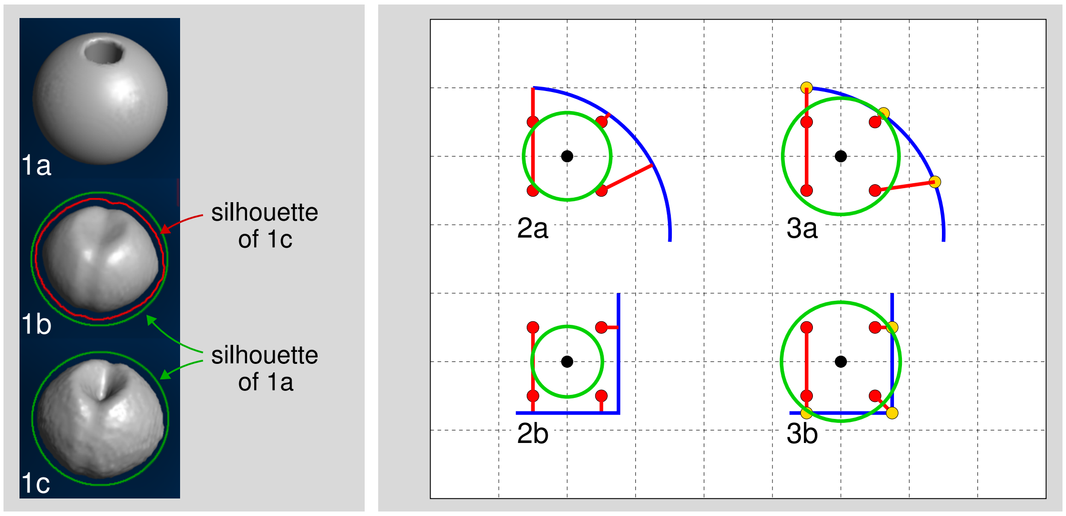


Figure 3.7: *Comparing particle radii using trilinear distance interpolation and neighborhood-sampling the nearest reference point. Left: A notched sphere (1a) is rotated* 8 *times, with reseeding* 5 *percent of the particles in each step using interpolated radii (1b) and nearest reference (1c). Middle and right: Two (2D) cases where interpolation (2a and 2b) leads to a larger error than nearest reference (3a and 3b).*

The sign s_p of the particle is determined by the sign of $\mathbf{dt}_d$ for the nearest grid cell. Note that there is no obvious way to obtain the sign on a subvoxel level as it is the case for the radius.

Fig. 3.7 (left) shows different results for both methods after eight rotations, each involving 100 advection and reseeding steps. One can observe a much higher volume loss when interpolating the distance, while the nearest reference (Eq. 3.3) leads to a slightly more fluctuating surface. Note that during the test, the complete particle set is reseeded 40 times. This configuration has been chosen purposely in order to show an extreme situation revealing the difference between the reseeding strategies. It should also be noted that with no particles at all, the volume completely disappears after six rotations due to numerical diffusion.

In Fig. 3.7 (middle and right), the grid cells (red points) contain the length of the normals (red lines) to the interface (blue). In Fig. 3.7 (2a, 2b), the distance at the particle position (black point) is computed with a bilinear interpolation of the cell distances. In case of a convex interface the resulting distance (green circle) is too small, because distances corresponding to very different normals are averaged. In Fig. 3.7 (3a, 3b), the nearest of the reference points is taken (yellow points). Here, another error is committed, as the selected reference point is not exactly the closest point on the interface to the particle (black point), but for highly convex interface parts, this is more accurate than the interpolation. For (almost) flat interface parts the interpolation is better because all normals point in (almost) the same direction.

In [ELF05], the radii of the particles are reset after level set reinitialization, while Mokberi and Faloutsos [MF06] omit this step. The reason for this is similar to the explanation why particle reseeding after each frame should be avoided: as particle

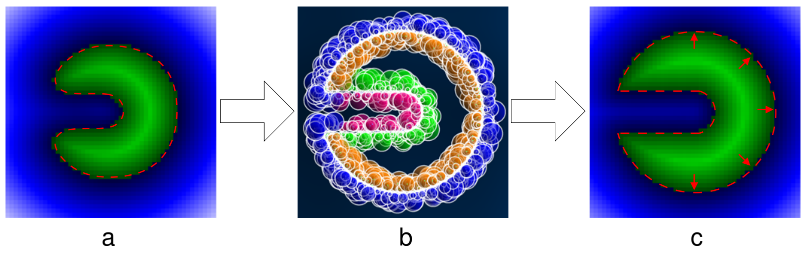


Figure 3.8: *Visualization of a level set correction step. a) The level set is clearly affected by numerical diffusion after advection. b) Inner region: green and orange (corrective) particles, outer region: blue and (corrective) magenta particles. c) The corrected level set.*

radii have to be reset with information stored in the level set, inaccuracies of the level set are propagated into the particle model. Thus, it is reasonable to wait until the next particle reseeding before the radii are updated. This approach is followed in the GPU-based framework, too.

3.3.6 Level Set Advection and Particle Tracing

Level set advection (Alg. 1.1, step 3) is easily ported to the GPU due to its parallel nature. New advected level set values $\mathbf{dt}_d$ are computed according to a semi-Lagrangian scheme (see Sec. 1.3.1). For particle tracing (Alg. 1.1, step 4), the same time step is used in a second order accurate Runge-Kutta integration (see Sec. 1.2.2).

As particle tracing is an operation on all particles, this is a good opportunity for choosing the subset of particles that will correct the level set. Those particles are marked by changing the sign of the first component of the position $\mathbf{x}_p$. As the volume of interest is bounded by $[0, 1]^3$, it is then easy to distinguish the position and the marker bit. In contrast to the original PLS approach (Sec. 1.3.4), the level set correction involves more particles: a particle contributes if its radius is greater than its distance to the interface. Following this rule, fewer particles are necessary in order to produce satisfactory results in the tested examples.

3.3.7 Level Set Correction

The level set correction (Alg. 1.1, step 5) combines the particle representation of the interface with the grid-based level set (see Fig. 3.8 for an illustration). More precisely, the particle information is transferred into the grid by locally updating the grid. In the following, we use a technique which is very similar to the scattering approach proposed in Chap. 2. Basically, the particles are rendering as point markers and blended into the grid. First, we need a way to select particles marked as *escaped*, and, based on this selection, to construct the intermediate level set functions ϕ^+, ϕ^-. In order to reduce bandwidth requirements, this is done differently than in the original formulation, postponing the combination with ϕ until the final update.

The key idea when constructing ϕ^+ and ϕ^- is to render a marker geometry at each particle position in order to trigger a computation for the 8 grid cells around the particle. In order to process all particles, the particle texture containing $\mathbf{x}_p$ and r_p is reinterpreted as a vertex buffer object. Then, all particles are sent through the graphics pipeline and a vertex program detects whether a particle p has escaped by checking the marker bit (see Sec. 3.3.6). If the particle has not escaped, it does not contribute to the level set correction and it is discarded by moving it outside of the volume.

After having evaluated different types of marker geometries, i.e. point sprites, quads, and individual points, the conclusion was drawn that rendering four individual points into the two subsequent slices of the grid is the most effective variant in this situation. The intermediate level sets ϕ^+ and ϕ^- given in Eq. 1.14 are stored in a two-component grid $\phi^\pm = (\phi^+, -\phi^-)$. The accumulation of particle contributions in $\phi^\pm$ uses min-max blending. Initially, $\phi^\pm$ is set to $(-\infty, -\infty)$ for all grid cells. Storing $-\phi^-$ instead of ϕ^- allows a single pass update of $\phi^\pm$ using maximum blending only. A second pass rasterizes the complete level set, computing the corrected level set $\phi^\pm$ according to Eq. 1.15, including the postponed combination with the previous state of ϕ. The new level set value is stored into the distance component of $\mathbf{dt}$.

3.3.8 Evaluation

In the following, the GPU-based PLS framework is evaluated by presenting some examples as well as performance and volume preservation results. The hardware used for testing is a PC with an NVidia GeForce 8800 GTS graphics chip. For evaluation, an object called *Zalesak's sphere* was taken (Fig. 3.13, page 63), which is a 3D version of the disk used in [ELF05]. The evaluation involves grids up to 128^3 and max. 4 million particles due to limited GPU memory.

The GPU method was compared with the PLS library [MF06] using exactly the same object, i.e. Zalesak's sphere (see Fig. 3.13, page 63), and equal parameters for both implementations. The test consists of 100 advection steps in a vortex flow field to a total of 360°. Fig. 3.9 shows the resulting frame rate and relative volume loss as function of the number of particles. Following prior work (e.g. Enright et al. [ELF05]), the resulting volume loss is measured by counting the number of interior grid cells of the object.

Tab. 3.2 lists the time consumption for all steps of the GPU-based algorithm separately for Zalesak's sphere. One can see that, depending on the resolution, the level set reinitialization and correction are the most time-consuming steps of the application. Due to the large size of the kernel used in the propagation method, the level set reinitialization is texture-fetch-bound.

3.4 Examples

With the GPU-based PLS framework proposed in the previous section, it is possible to accurately trace the surface of a freely deformable object in real-time. Some academic examples on page 63 demonstrate this usage by applying a predefined flow to different initial objects. Obviously, the approach is not restricted to analytical

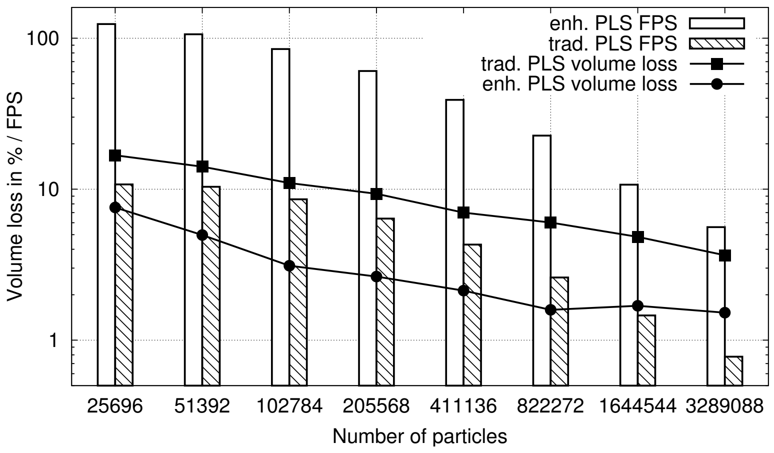


Figure 3.9: *Comparison of traditional PLS (trad.) and the GPU-based method (enh.) for Zalesak's sphere (same logarithmic scale for both). 64^3 grid, no particle reseeding.*

Table 3.2: *Runtime of the steps involved in the PLS algorithm. Object: Zalesak's sphere (see Fig. 3.13, page 63), 262,144 particles. In each frame,* 5 *percent of the particles are reinitialized.*

step	grid / time (ms)	grid / time (ms)
framework	64^3 / 1.3	128^3 / 1.3
LS advection	64^3 / 0.15	128^3 / 1.15
particle tracing	64^3 / 0.05	128^3 / 0.25
LS reinitialization	64^3 / 5.83	128^3 / 40.65
LS correction	64^3 / 10.8	128^3 / 12.88
particle reseeding	64^3 / 0.65	128^3 / 0.475

flow functions, in fact, the next chapter introduces PLS volumes as a geometric visualization technique for visualizing complex flow data sets.

The following two applications of the GPU-based PLS framework show that user interaction with the PLS object is possible, and that the DT representation of the level set can be used to realize an easy collision detection and reaction.

3.4.1 Object Modeling

Interactive advection of a PLS-based object can be used as a 3D modeling tool (see Fig. 3.10). The idea is to apply a locally bounded flow in the region illustrated by the semi-transparent sphere. The user can move and resize the sphere in order to change the region of interest. When pushing a button, the volume is advected according to the so-called deformation painting approach by Funck et al. [vFTS06]. This defines a local velocity field, which is volume-preserving and which allows one to smoothly shift a part of the volume without any discontinuities. The accuracy of

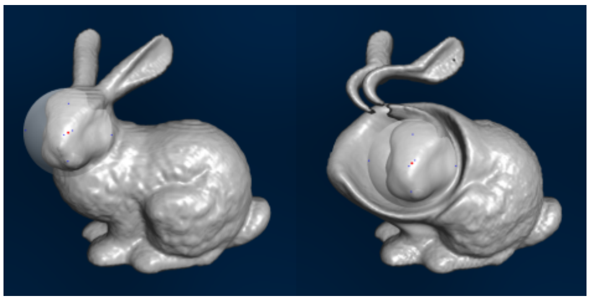


Figure 3.10: *The Stanford bunny is subject to a local deformation controlled by the user. This deformation painting resembles clay modeling.*

the PLS representation ensures that no volume is lost during modeling. Thanks to this property, the modeling tool is comparable to the use of real clay.

3.4.2 Collision with Deformable Objects

The use of a DT as level set structure in the PLS framework is already justified by its use in the PLS algorithm. However, the DT can be convenient beyond that: With the aid of the shortest distance information, a simple yet powerful collision detection can be realized by checking whether a collider is situated in the inner part of the PLS object, i.e. $\mathbf{dt}_d < 0$. When a collision is detected, the reference point stored in the second component $\mathbf{dt}_\delta$ of the DT can be used to realize a collision response, e.g. by pushing the collider onto the reference point.

This idea has been applied to two test cases: the first comprises a combination of a PLS object (the Stanford bunny) and a set of particles moving in the direction of a user controlled gravitational point (see Fig. 3.11). The movable gravitation point, which is not visible in the image, can be moved and altered in a similar way to the one in Fig. 3.10.

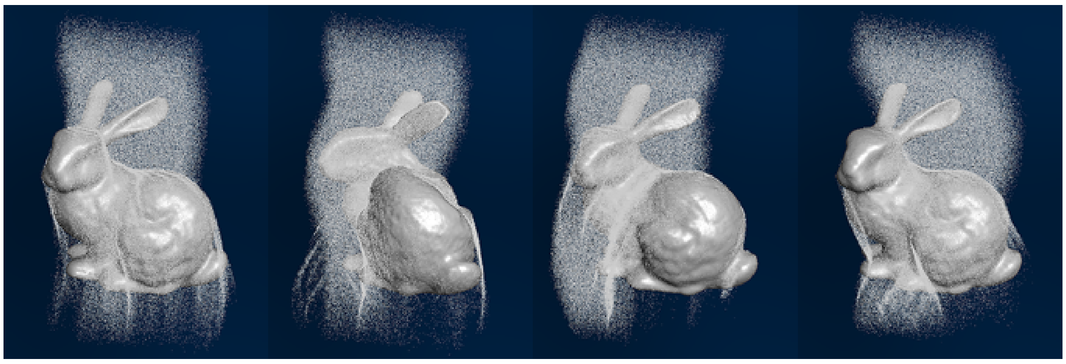


Figure 3.11: *The Stanford bunny in a deforming vortex while being sprinkled with "snowflakes". This is an example showing the combination of the PLS framework and flow particles. 262,144 flow particles, PLS:* 96^3 *grid, 524,288 corrective particles. Frame rate: 15.74 FPS (GeForce 9600 GT).*

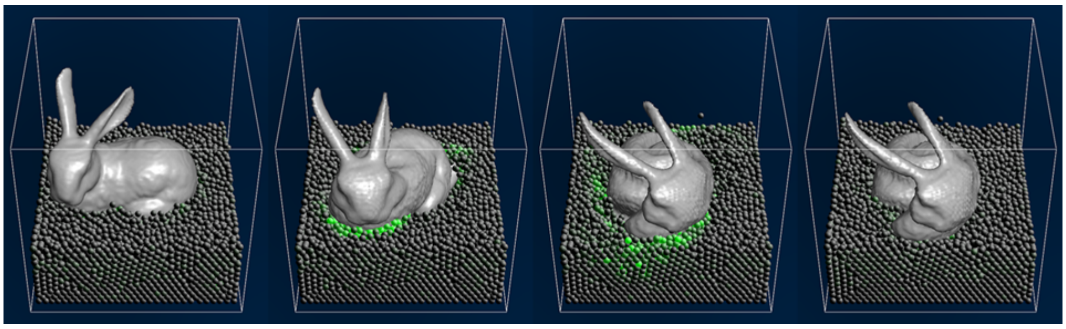


Figure 3.12: *The Stanford bunny, warped by a deforming vortex, stirs up water. This example shows how the PLS can be combined with the fluid solver. 12,288 fluid particles, PLS:* 96^3 *grid, 524,288 corrective particles. Frame rate: 15.83 FPS (GeForce 9600 GT). Green particles are those to which an acceleration is applied.*

The second test case lets the PLS object collide with a fluid modeled with the approach of Chap. 2. A deforming vortex rotates the bunny, and during the advection, the bunny's body brushes the fluid particles aside (see Fig. 3.12). Using the DT as collision criterion, the fluid can be easily pushed out of the body by using a similar rule to the one proposed for heightfields in Sec. 2.1.5.

In this chapter, a modified and enhanced GPU-based PLS method for deformable surfaces/objects has been presented. The method shows that deformable surface tracing can be performed efficiently and accurately on the GPU. In the next chapter, geometric flow visualization techniques will be presented, including a model for flow volumes, which is based on the GPU-based PLS framework.

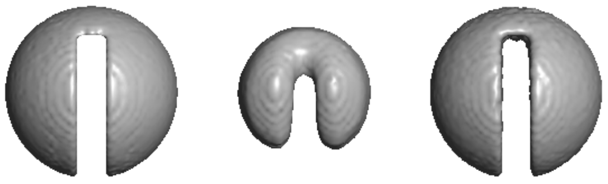


Figure 3.13: *360° rotation of Zalesak's sphere (100 advections): initial, LS, and PLS. 524,288 particles, 128^3 grid, 14.74 FPS.*

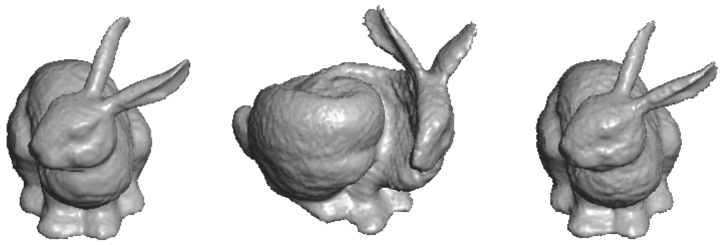


Figure 3.14: *The "Stanford bunny" in a deforming vortex flow: initial, after 40 advections, after backward advection. 131,072 particles, 96^3 grid, 40.47 FPS.*

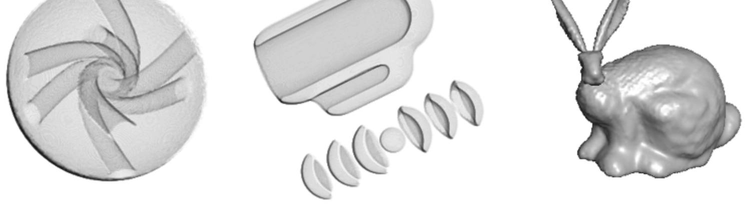


Figure 3.15: *Some examples of PLS surfaces. From left to right: A sphere with 4 crossing tubular holes (semi-transparent rendering), a more complex constructed geometry, the bunny in a locally contracting flow.*

Chapter 4

Real-time Flow Visualization

Once again, his fingers attack the keyboard of the smart-stupid, and in half a trice, there appear on the screen an enormous number of lines, swinging about on the screen

Douglas R. Hofstadter: Gödel, Escher, Bach

This chapter discusses the challenges of flow visualization in the context of climate research. This motivates a set of techniques related to the GPU-based particle techniques presented in the previous chapters. The overall goal in this chapter is to provide the user of the visualization with interactive *and* accurate methods for exploring and visually analyzing the structure of complex flow data sets. The techniques involve a wide range of geometric flow visualization primitives, specifically a purely point-based representation (Sec. 4.1), flow lines (Sec. 4.2) and flow surfaces/volumes (Sec. 4.3). The flow line generator is designed to support different line types and the use of an adaptive time step. The flow volume approach is connected to the PLS framework presented in Chap. 3 in the sense that it extends the applicability to specific volumetric flow primitives. Finally, in Sec. 4.4, the use of virtual reality is proposed as an improvement of the user interface.

The climate visualization system has been introduced in [CKL*07], the time-adaptive flow line generator has been published in [CPK09], and the flow volumes PLS method is part of [CKSW08].

The problem of designing a practical flow visualization system reveals, besides data processing or feature extraction tasks, two major challenges: interactivity and accuracy. Interactivity stands for explorative capabilities and real-time control. Accuracy is a prerequisite for every professional visualization in order to provide a reliable base for analysis of a data set. Next to these fundamental requirements, a flow visualization must cope with very different target applications. Typical application areas include climate research, where general circulation models are used to describe atmospheric and oceanic phenomena, as well as the aerospace, automotive, and chemical industries. Depending on the application, the visualization must be designed to reveal specific characteristics of the flow. For example, some cases might be best visualized by a dense representation, others may benefit from a specific type of geometric primitive. Another example could provide the user with precise values or glyphs representing relevant attributes. This wide range of variations makes flow visualization a research topic with lots of facets.

In the following, we will concentrate on one specific domain: the visualization of climate phenomena. More precisely, three building blocks of geometric techniques will be discussed: a purely point-based representation, flow lines and flow surfaces/volumes. The concrete goal is to maximize efficiency and accuracy by providing algorithms designed to run on parallel graphics hardware. All of the mentioned geometric flow primitives can be processed in parallel. High accuracy is explicitly addressed by the use of high order integration schemes, adaptive time-stepping and PLS surface representations.

The GPU with its fast rendering *and* general purpose capabilities turns out to be an ideal tool for visualization systems which are computationally very expensive. In general, the algorithms presented in this chapter scale well to larger data sets. All three geometric techniques support 4-dimensional, unsteady flow data sets which are not necessarily restricted to the size of the GPU memory. While the evaluation has been performed on different data sets and analytically defined flows, the main example represents a tropical storm, i.e. a typhoon which is introduced in the next section and will be referred as the *typhoon data set* in later sections.

4.1 Particle-based Climate Flow Visualization

The content of this section has been published in cooperation with the German Computing Centre Hamburg [CKL*07]. The task of visualizing complex climate flow data sets is solved by proposing a fully GPU-based, interactive visualization system employing points as geometric primitives.

It is the purpose of climate models to simulate the most important processes of the Earth system, including the circulation of the atmosphere and the ocean. The results, which typically are large unsteady – i.e. time dependent – data sets, have to be analyzed if a person wants to interpret them or to deduce any properties or qualities. One powerful and important possibility is to visualize the flow. Most commercial and free software packages available for climate data visualization have limited support for the interactive exploration of large flow data sets. In fact, some of them are designed to generate precalculated animations only. On the other hand, recent developments in visualization techniques exploiting programmable features of current GPUs have proven to be very powerful. This is especially true for particle-based techniques, as we have seen in the preceding chapters (see for example Sec. 1.2). However, there is still a functional gap between these particle-based flow visualization techniques utilizing the GPU's processing power and the requirements in the context of geophysical flow, which is the focus in the following.

A complete system for the interactive visualization of unsteady climate flow data is presented. The system comprises a proper data workflow from the simulation back-end to the visual output. Beyond that, the system offers a sound support for the interactive exploration customized to climate research. The applicability is demonstrated on a typhoon data set which is introduced in the next part.

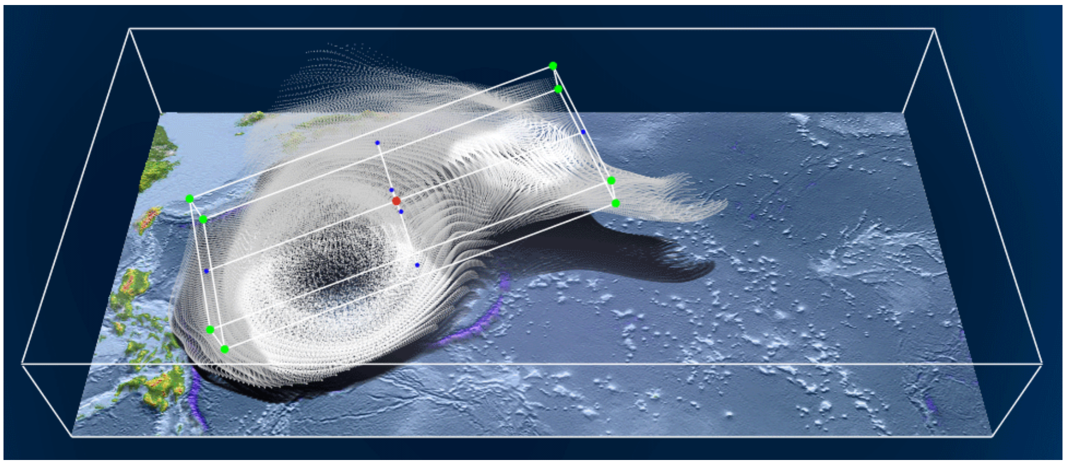


Figure 4.1: *A particle-based flow visualization of the typhoon (data set: courtesy of MPI for Meteorology). The quadrilateral box surrounding the red point can be moved, scaled and rotated in order to control particle emission. This helps the user to explore specific regions of the flow volume. Frame rate: 80 FPS.*

4.1.1 The Typhoon Data Set

For the simulation of long term changes of the Earth's climate, coupled 3-dimensional models of the most important compartments of the earth system are used: atmosphere and ocean, including models for sea ice and land cover. These so-called general circulation models simulate the dynamics of fluids. The partial differential equations which describe these processes can be solved only approximately for discrete computational grids. The resulting multivariate and unsteady data consist of numerous scalar and vector quantities, which are stored for all grid cells in regular time intervals. Due to limitations in storage capacity, it is only feasible to use storage intervals (e.g. 6 hours) which are much larger than the computational time step (e.g. 20 minutes) used for the simulation.

The chosen data is a subset of the results of a complex simulation based on the global atmospheric general circulation model ECHAM5 [RBB*03] in so-called T213 resolution, which corresponds to a horizontal grid spacing of 0.5625 degrees or approximately 60 km near the equator. For the visualization, the originally 31 vertical hybrid model levels have been interpolated onto 39 rectilinear height levels. A time period and region is selected where a strong tropical storm, a typhoon, occurs in the simulation. The resulting subset of the original data is $106 \times 53 \times 39$ grid cells and 32 time records large.

Internally, the data set is stored as a stack of 3D textures containing the 32 time records. As a result of the simulation process, the slices have a non-uniform spacing in the vertical direction. This means that the distance between two slices varies along the upper direction. A solution for this specific problem is given in Sec. 4.1.5.

4.1.2 Requirements and Motivation

The visualization of flow data in the context of climate makes high demands on the techniques and system specifics. The overview given by Hibbard et al. [HBSB02] argues that visualization has a high potential to facilitate data exploration, and a set of desired functions is formulated. Many of these requirements are related to interfaces and usability. The following list of aspects concerning the data structures and the algorithms used for the visualization has been deduced. These aspects also form the motivation for the GPU-based system. The respective solutions proposed are thus stated for each of the requirements.

Data Sets: In general, the output of the simulation consists of large unsteady flow data. The proposed technique is designed to asynchronously transfer the data, so that data sets exceeding the size of GPU memory are supported. An evaluation of this functionality will be given later in Sec. 4.2 for the flow lines approach.

Grids: Ideally, the visualization system should read the data in the format given by the simulator without any conversion. For example, the input grid could be curvilinear in one direction and non-uniformly rectilinear in another one. However, for real-time applications, fast data access is decisive. In the proposed system, a special case of non-uniformity is explicitly addressed. Curvilinear grids are supported by the approach in principle, i.e. in cases where the curvilinearity can be described by a mapping which can be evaluated analytically.

Data Formats: Clearly, a different data storage might be wise for the visualization. This affects the grid type, the units, the resolution and the file format. The GPU-based system does not introduce any further constraint on the data management. Thus, the integration of existing tools for data handling suffices to guarantee a proper data handling.

Flow Primitives: Many systems use stationary arrows for the visualization of vector quantities, i.e. wind or ocean currents. This is convenient and reasonable, because the arrows can be replaced with glyphs coding additional information like temperature, humidity etc. However, this kind of glyphs fails to visualize the transport within unsteady flow fields, which is the focus application of this chapter. Particle trajectories close this gap by visualizing the transport explicitly. First, a purely particle-based approach is introduced, later in this chapter the method is extended to more complex primitives, i.e. lines and surfaces/volumes.

Geographical Mapping and Visual Effects: A proper geographical mapping, in combination with visual effects like shadows and lighting effects significantly improves the perception of the flow primitives and thus of the overall visualization. In this framework, a heightfield represents the geographical area, and the renderer is capable of lighting and shadowing.

Interactivity: Interactive visualization is especially important when a highly dynamic scenario, for example a storm event, is investigated. This presumes real-time rendering, but it also makes demands to the usability: for example, the viewer may want to isolate specific regions of interest during visualization. This task is solved by placing a particle emitter into the volume, for which the viewer can vary the size, location and proportions.

Accuracy: In order to assess the quality of a climate model, simulations of the past and the current climate are made and compared with observations. This is only possible if some level of accuracy is guaranteed. In the case of particle tracing, this implies the use of high order integration schemes like RK4.

Computational Resources: Many systems, especially real-time tools, require high-end hardware in order to cope with the large amount of data. Performing interactive flow analysis on standard PCs highly decreases the costs for the hardware and helps to reduce the time for development and evaluation of climate models.

4.1.3 Prior Work

The development of programmable graphics hardware induced a large number of very powerful techniques in the visualization context. Examples are real-time volume graphics [EHK*06] and illustrative visualization [VKG05].

Flow visualization is a well-studied topic (see Sec. 1.4, and [LHD*04] for one of several extensive overviews). Most of the literature is related to dense representations, particularly line integral convolution [CL93, SK98] and similar texture-based techniques. Often, the visual output is designed to reveal as many characteristics of the flow as possible, e.g. using multi-dimensional transfer functions (MDTFs) [PBL*04]. Accordingly, feature-based visualization [PVH*03] extracts and visualizes flow features. Geometric methods, on the other side, are typically based on particle traces that involve a numerical integration of the flow field. GPU-based particle systems are known to be very efficient (see Sec. 1.2.3). Krüger et al. extend the idea to a powerful 3D flow visualization [KKKW05], without explicitly focusing on climate phenomena.

Although the interactive visualization of the climate data is more and more becoming essential for the analysis and communication of the results, only few suitable visualization systems are available, and in most cases they offer only a limited subset of the desired functionality listed in Sec. 4.1.2. The `vis5d`(+) project offers a free `OpenGL`-based volumetric visualization program for scientific data sets in three or more dimensions, i.e. supporting unsteady flow data [HS90]. Geometric visualization using e.g. arrows is possible in `vis5d`, however, after the initial particle placement the tracing is done on the CPU, and the positions are stored for all time steps before the results can be visualized. It is not possible to move the emitter around in order to explore the flow field interactively. Other visualization systems like AVS/Express lack the desired real-time performance, especially for a larger number of particles.

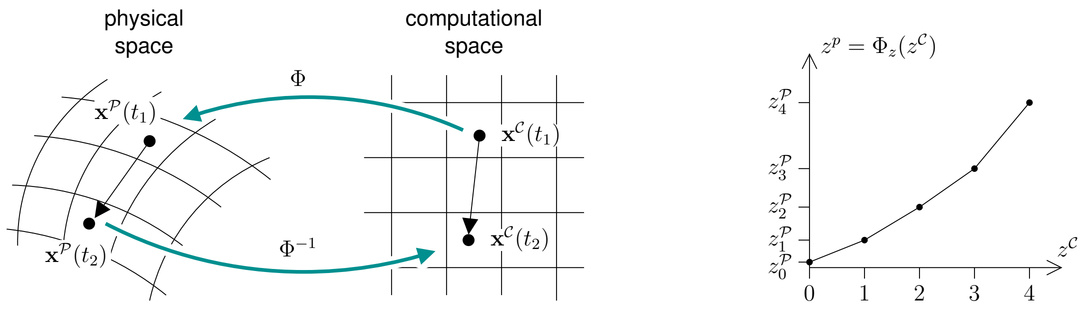

physical space
computational space
Φ
Φ^{-1}
$\mathbf{x}^{\mathcal{P}}(t_1)$
$\mathbf{x}^{\mathcal{P}}(t_2)$
$\mathbf{x}^{\mathcal{C}}(t_1)$
$\mathbf{x}^{\mathcal{C}}(t_2)$
$z^{\mathcal{P}} = \Phi_z(z^{\mathcal{C}})$
$z_4^{\mathcal{P}}$
$z_3^{\mathcal{P}}$
$z_2^{\mathcal{P}}$
$z_1^{\mathcal{P}}$
$z_0^{\mathcal{P}}$
$z^{\mathcal{C}}$
0
1
2
3
4

Figure 4.2: *Mapping physical space to computational space. Left: a particle is traced from time t_1 to t_2. Right: the ascending altitudes with increasing spacing in the case of atmospheric simulations.*

4.1.4 The Data Flow Until Visualization

The data coming from the simulation stages are transmitted in `NetCDF` (network Common Data Form), which is a suitable format for the storage, exchange and further analysis of scientific data. At this stage, the data include the velocity in horizontal direction in $[m/s]$ and in vertical direction in $[Pa/s]$ (pressure variation per second), as well as additional quantities like temperature, relative humidity and surface pressure. In a preprocessing step, the vertical component of the 3D velocity vectors are converted to $[m/s]$, resulting in a consistent flow vector representation. These data are then exported to a binary float format using the *climate data operators*[1]. The data are now ready to be used in the GPU-based particle visualization.

4.1.5 Non-uniform Grids

In computational science, one distinguishes between the physical space $\mathcal{P}$ and the computational space $\mathcal{C}$ (sometimes also called *p-space* and *c-space*). In our case, the physical space is a specific region on Earth, and the computational space corresponds to the array in which all vector and scalar quantities are stored. Note that the quantities are valid in $\mathcal{P}$ only. A location $\mathbf{x}^{\mathcal{C}}$ in $\mathcal{C}$ can be mapped to a location $\mathbf{x}^{\mathcal{P}}$ in $\mathcal{P}$ via a bijective mapping $\Phi : \mathcal{C} \to \mathcal{P}$ (see Fig. 4.2, left side). For local phenomena, the grids are defined in longitude-latitude coordinates and altitude. The longitude-latitude coordinates are subject to a curvilinear transformation, which can be approximated by an analytical function Φ_{xy}. The altitude mapping Φ_z is defined using a list of fixed and ascending height values $z_1^{\mathcal{P}}, \ldots, z_{\max}^{\mathcal{P}}$. This way, a higher sampling in lower altitude (i.e. lower atmosphere in the physical space) can be achieved (see Fig. 4.2, right side).

The particles traced during visualization live in the physical space. Thus, it is necessary to transform a particle position into computational space when sampling any particle attribute, especially the particle velocity. Here, we make the assumption that the attribute can be obtained using trilinear interpolation on the computational grid. This is convenient as trilinear interpolation is hardware-supported when using

[1]http://www.mpimet.mpg.de/˜cdo (MPI for Meteorology, Hamburg)

a 3D texture. In the typhoon example, Φ_{xy} is linear in both directions. However, this is not obligatory: Provided that Φ_{xy}^{-1} is known analytically, one could describe curvilinearity by interconnecting the calculation directly in the code where the computational grid is sampled. The non-uniformity in z-direction is handled by searching two appropriate slices $z_i^{\mathcal{P}}, z_{i+1}^{\mathcal{P}}$ before linear interpolation. For a particle's altitude $z^{\mathcal{P}}$, the index i must be chosen so that $z^{\mathcal{P}} \in [z_i^{\mathcal{P}}, z_{i+1}^{\mathcal{P}})$. It will turn out that this can be done efficiently by performing a binary search.

4.1.6 Asynchronous Data Transfer

The transfer from main memory to GPU memory is limited by the bandwidth of the graphics port, e.g. AGP (Accelerated Graphics Port) or PCIe (PCI Express). A naive call of `OpenGL`'s upload function causes the application to wait until all data has been written before the control flow continues. This can be a considerable problem in the case of flow visualization if the flow data are too large to fit in the graphics memory completely. Copying larger chunks of memory results in undesirable delays or, even worse, an interruption of the animation. Using `OpenGL`'s *pixel buffer object* extension, it is possible to transfer data asynchronously, i.e. a buffer located in CPU memory is copied in parallel while control flow continues. As we will see later in the context of flow lines (Sec. 4.2.8), this yields a clear improvement if high transfer rates are involved.

4.1.7 Height Field Rendering

A heightfield is used to represent the Earth's surface. This is a well-known technique for rendering terrain geometries. The heightfield is represented by a gray value texture covering the area of interest and storing the height of the terrain in each element. A regular triangle mesh with as many vertices and same arrangement as the gray value texture, forms the heightfield geometry. Each vertex is moved along the z-displacement defined by the texture. An additional texture, with potentially higher resolution, is used to visualize detailed color information.

4.1.8 Particle Tracing

The particle tracing works according to the texture-based technique for state-preserving particle systems introduced in Sec. 1.2.3. The particle positions are stored in a floating point texture. The fragment program used for particle motion is capable of computing an RK2 or RK4 velocity integration (see Sec. 1.2.2). The transformation from physical to computational space is performed according to the strategy proposed in Sec. 4.1.5. Accordingly, each time a velocity is sampled, a binary search is performed in order to handle the non-uniformity in z-direction. For this purpose, a loop accesses a 1-dimensional texture where the height values $z_1^{\mathcal{P}}, \ldots, z_{\max}^{\mathcal{P}}$ are stored.

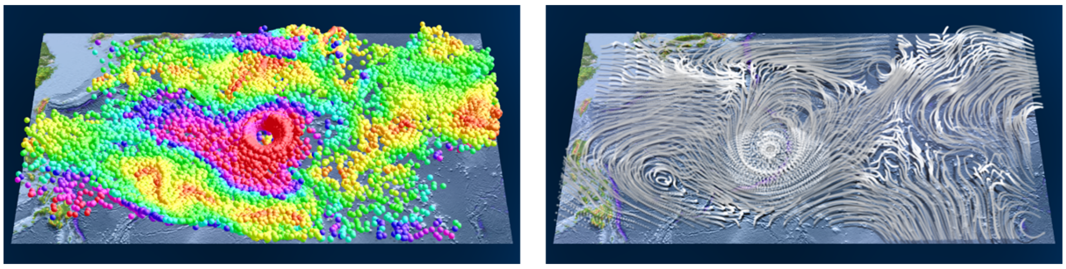


Figure 4.3: *Comparison of two different particle rendering techniques with the typhoon example (data set: courtesy of MPI for Meteorology). Left: Solid spheres color-coding the velocity magnitude (142 FPS). Right: Pseudo streak lines generated by frequently emitting semi-transparent particles at the same locations (132 FPS).*

4.1.9 Rendering Particles and Shadows

The particles are rendered either as solid spheres or as semi-transparent sprites (compare with Fig. 4.3). The spheres are rendered using a binary alpha component in the point sprite stenciling the sphere in conjunction with an alpha test. The actual sphere shape is generated in a fragment program using a Phong model. The semi-transparent sprites are rendered using alpha blending. Currently, this is restricted to commutative blending operators, in order to prevent a particle depth sorting. However, there is no conceptual challenge to extend this to non-commutative blending operators (see [KLRS04, KSW04]). Note that when frequently emitting particles at the same location, this rendering mode tends to produce smooth streak lines. We call them *pseudo streak lines*, as the line segments are not interconnected. A method for generating true streak lines is presented in Sec. 4.2. When using the sphere mode, it is possible to color-code additional scalar quantities stored in the 3D flow texture. However, this can be applied only in case of the spheres, since blending would alter the color values.

Along with the particle rendering, shadows facilitate the perception of the spatial location and orientation of displayed 3D objects (see Fig. 4.1). In our case, particles moving in the flow can be associated more easily with locations on the underground. This is especially the case where the flow is not restricted to stay in a plane, thus forcing the user to choose different points of view. Shadows are rendered orthogonally onto the textured heightfield. This is done by mapping the z component of the particle positions in a vertex program while re-rendering all particles. The advantage of orthogonal mapping in combination with heightfields is that even for elevated regions, the shadows remain correct.

4.1.10 Evaluation

The evaluation of the described visualization framework was performed using an implementation running on a GeForce 9600 GT graphics card (512 MB memory). Tab. 4.1 provides a set of runtime measurements for different configurations. It is apparent that the visualization is capable of real-time processing even for a large

Table 4.1: *Runtime for the example shown in Fig. 4.1.*

particles	height field	shadows	binary search	FPS (RK2/RK4) spheres	FPS (RK2/RK4) pseudo streaks
262,144	✓	✓	✓	80.58/57.92	80.44/58.41
262,144	✓	–	✓	113.88/74.36	113.82/73.79
262,144	–	–	✓	131.04/80.27	129.96/79.82
524,288	✓	✓	✓	43.03/30.35	42.97/29.87
1,048,576	✓	✓	✓	22.76/15.53	23.34/15.44
1,048,576	✓	✓	–	23.26/17.13	23.57/16.63

number of particles (more than one million). The rendering mode (spheres or pseudo streak lines) exerts no noticeable influence on the frame rate. Even when zooming into the scene, the higher fill rate is no significant challenge for the hardware. Rendering the particle set twice in favor of particle shadows, however, makes a difference. Still, the computational costs for the velocity integration outweigh rendering, which can be seen when comparing the RK2 and the RK4 integration: The frame rate for the fourth order integration is about 0.7 times the second-order integration. Note that visually, it is very hard to perceive any difference between the RK2 and RK4 integration. For example, Fig. 4.1 has been compiled in RK2 mode, still yielding very plausible results. The binary search, which is used for the piecewise linear approximation of the non-uniform slice arrangement, turns out to be very efficient. Only a marginal increase of the frame rate can be achieved by deactivating this routine.

The positive feedback given by a domain expert after having tested the system shows that the presented application is a valuable improvement compared to previous solutions for climate research. The user highly appreciates the possibility to interactively navigate through the data and to explore the inherent information by selectively probing the vector field using the particle emitter.

4.2 Time-Adaptive Flow Lines

This section expands the set of flow primitives in the visualization framework to time-adaptive stream, path and streak lines. The following concepts have been published in [CPK09].

Geometric flow visualization has a long tradition and comes in very different flavors. Among these, stream, path and streak lines are known to be very useful for both 2D and 3D flows. Despite their importance in practice, appropriate algorithms suited for contemporary hardware are rare. In particular, the adaptive construction of the different line types is not sufficiently studied.

The following work provides a profound representation and discussion of stream, path and streak lines. Two algorithms are proposed for efficiently and accurately generating these lines using modern graphics hardware. Each includes a scheme for adaptive time-stepping. The adaptivity for stream and path lines is achieved through a new processing idea we call *selective transform feedback*. The adaptivity for streak lines combines adaptive time-stepping and a geometric refinement of the curve itself.

The method is applied to the typhoon example introduced in Sec. 4.1.1. The storage as a set of 3D textures requires special attention. Both algorithms explicitly support this storage, as well as the use of precomputed adaptivity information.

4.2.1 Flow Lines and their Challenges

In the following, we concentrate on stream, path and streak lines. A brief introduction to these flow line types is given in Sec. 1.4, a more formal one will be provided in Sec. 4.2.3.

Typically, the flow line equations are solved by applying a high order integration like Runge-Kutta 2 or 4. This kind of numerical computation is based on discrete time steps. Simple strategies use a constant time step (see Fig. 4.5a & b), more sophisticated ones choose a time step that adapts to local flow characteristics (Fig. 4.5c). The objective of adaptive time-stepping is a reduction and adjustment of the truncation error of the integration. Ideally, this results in a finer sampling of intricate regions (Fig. 4.5: ①) and a coarser sampling in problem-free regions (Fig. 4.5: ②). Flow lines visualized by connecting linear segments eminently benefit from time-adaptivity, as sharp bends can be avoided.

Adaptive time-stepping can be applied directly to stream and path lines. Unfortunately, streak lines suffer from another problem: Discrete sampling is emphasized during evolution. This means that injected points will diverge according to flow divergence, resulting in sharp bends and overpopulated regions like in Fig. 4.6a.

This work discusses the visualization of adaptive flow lines using up-to-date graphics hardware. The presented approach includes the following contributions: A clear discussion of stream, path and streak lines is provided. Then, two algorithms are proposed for efficiently generating lines using geometry programs. In this context, a new processing idea called *selective transform feedback* is presented. The approach includes direct support for time-adaptive stream and path lines. Streak lines require a different treatment in order to avoid the divergence problems shown in Fig. 4.6a. Accordingly, an appropriate refinement scheme for adding and removing points dur-

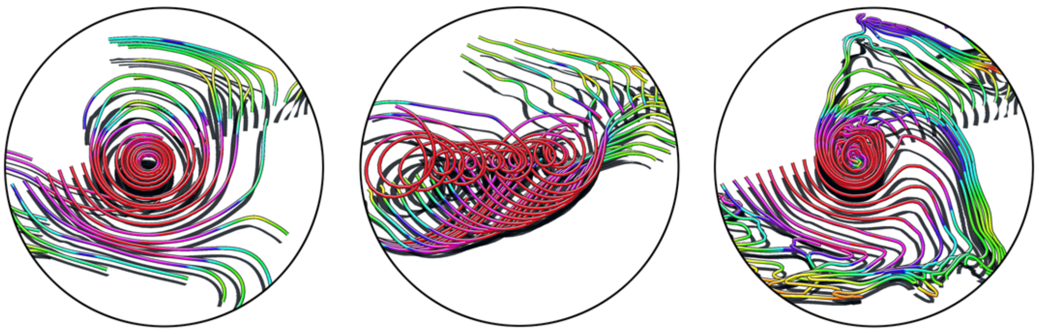


Figure 4.4: *Time-adaptive stream, path and streak lines (from left to right) in the unsteady typhoon flow (data set: courtesy of MPI for Meteorology). All images are taken at the same time. The velocity magnitude is mapped to "rainbow" colors (red: high velocity).*

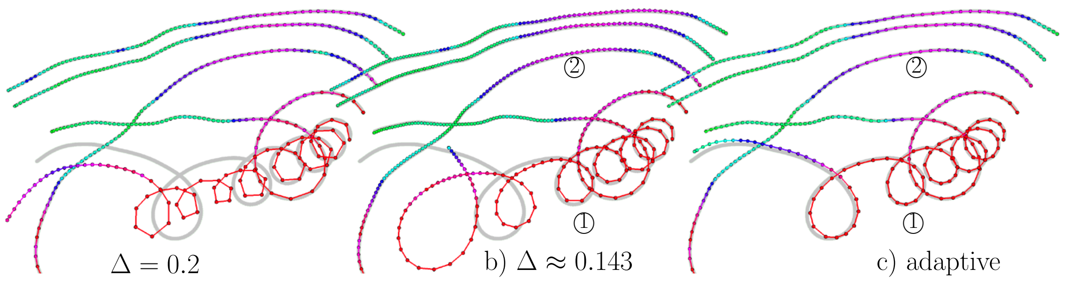


Figure 4.5: *Five path lines without adaptive time-stepping: time step* $\Delta = 0.2$*, 66.22 FPS, 385 segments (a),* $\Delta \approx 0.143$*, 63.96 FPS, 540 segments (b), and with adaptive time-stepping: 66.23 FPS, 355 segments (c). The gray ground-truth curve has been generated using time step* $\Delta = 0.02$*.*

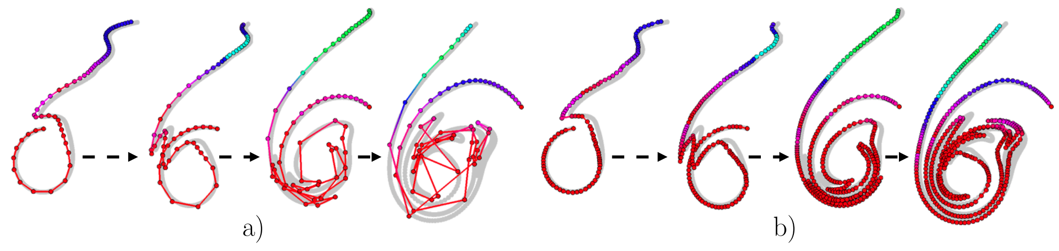


Figure 4.6: *Stop-motion of a streak line without (a) and with refinement (b). Frequency of the injection:* $\Delta = 0.2$*, both > 400 FPS. The gray ground-truth curve has been generated using time step* $\Delta = 0.02$*.*

ing streak line evolution is proposed, which helps to maintain a balanced curve progression (Fig. 4.6b). This scheme can be regarded as *geometric* adaptivity. The asynchronous flow data access required by the parallel time-adaptive particle tracing is explicitly addressed in the algorithms. Consequently, the approach supports texture-based unsteady flow data sets. After all, a qualitative evaluation of both a complex data set and analytical examples is provided. The approach is also applied to very large flow data sets which do not fit entirely into GPU memory. The evaluation shows that asynchronous texture transfer results in interactive frame rates even for those large data sets.

4.2.2 Prior Work

The work by Park et al. [PBL*05] discusses dense seeding of stream and path lines in steady and unsteady flows. Their technique supports MDTFs, and it is suited to graphics hardware. However, no streak lines are generated, and the algorithm requires one render pass per segment. Finally, adaptive time-stepping is not handled in this work. Liu et al. present a strategy for 2D stream line placement [LMG06] featuring loop detection and adaptive distance control. The results show clear images where flow patterns can be easily recognized by the viewer. The GPU-based flow visualization method by Krüger et al. [KKKW05] includes the generation of stream lines and ribbons. No adaptive time-stepping is provided, however a visual feedback of the truncation error marks difficult areas.

The rendering of flow lines using graphics hardware can be done very efficiently. Zöckler et al. present a technique for real-time illumination that uses texture mapping capabilities of the GPU [ZSH96]. Stoll et al. propose a hybrid CPU-GPU algorithm for a stylized line rendering, that combines piecewise-quadrilateral approximations with exact tube geometries [SGS05].

The Runge-Kutta method is a widely used high order integration scheme that is known to produce numerically stable traces. For further details about particle tracing, i.e. integration schemes and adaptive time-stepping according to the step-doubling approach, see Sec. 1.2.2. An extensive survey on Runge-Kutta methods is given by Cartwright et al. [CP92], also including a discussion of adaptive time-stepping. Advanced state-of-the-art techniques using the theory of automatic control for the design of adaptive numerical time-stepping are surveyed by Söderlind [Söd02]. An analysis of the truncation error (due to discrete time-stepping) and the interpolation error (due to discrete flow data) for particle tracing methods in general is provided by Teitzel et al. [TGE97]. Note that a different interpretation of adaptivity in the context of flow lines is used by Sanna et al. [SMA00]. They create a dense texture-based representation by adaptively *seeding* streak lines into the problem domain.

The graphics pipeline, as introduced in Sec. 1.1.1, includes the geometry shader and a processing paradigm called transform feedback, which will be of special interest in the following. Remember that the geometry shader transforms each input primitive (i.e. a point, line or triangle) by adding or removing vertices. The output can be a variable number of reassembled primitives. This number, however, is limited by the hardware implementation. Geometry programs apply well to algorithms which dynamically alter or create geometrical entities, like subdivision surfaces or marching cubes. The output primitives are then propagated to the rasterization stage, or they can be redirected to a specific stream using `OpenGL`'s *transform feedback* extension. Note that in principle, a geometry program corresponds to a general case of stream compaction, which is a well-studied problem in the context of parallel algorithms. It should also be noted that geometry programs constitute an advantage in comparison to the functionality provided by general APIs like `CUDA`. At present, `CUDA`-based stream compaction exists (see e.g. [HSO07]), however there exist no direct support for on-the-fly insertion of stream elements in `CUDA`. In the following, we will make heavy use of both the geometry shader and transform feedback.

4.2.3 Stream, Path and Streak Lines

Assume an unsteady flow given by a 3-dimensional velocity field $\vec{\mathbf{v}} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$ which maps a spatial location $\mathbf{x}$ and an instant in time t to a velocity vector $\vec{\mathbf{v}}(\mathbf{x}, t)$. The following formal definition of stream, path and streak lines is proposed.

The **stream line**, starting at $\mathbf{x}_0$, is defined as the trace of a particle moved in the

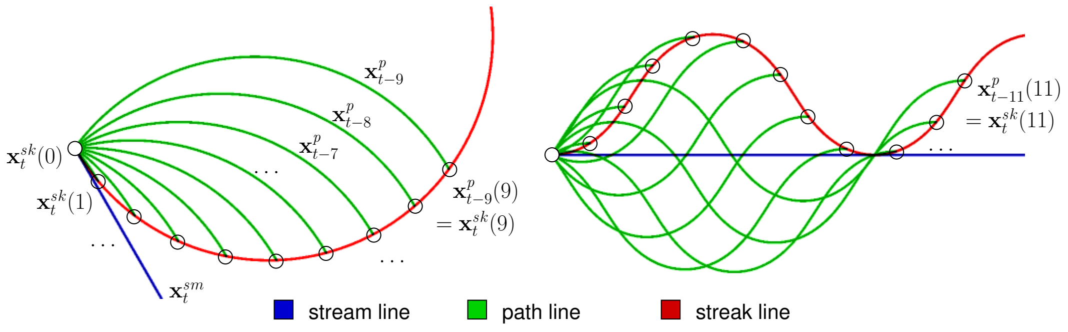


Figure 4.7: *Schematic comparison of stream, path and streak lines. Note that a streak line consists of the ending points of path lines. Left:* $\vec{\mathbf{v}}(\mathbf{x},t) = (cos\, t, -sin\, t, 0)$. *Right:* $\vec{\mathbf{v}}(\mathbf{x},t) = (cos\ (sin\ t), -sin\ (sin\ t), 0)$.

flow at a fixed time t:

$$\mathbf{x}^{sm}_t(s) \;=\; \mathbf{x}_0 + \int_0^s \vec{\mathbf{v}}\big(\mathbf{x}^{sm}_t(\sigma),\ t\big)\ d\sigma \tag{4.1}$$

The **path line**, starting at $\mathbf{x}_0$ and t, is defined as the trace of a particle moved in the flow which is changing in time:

$$\mathbf{x}^{p}_t(s) \;=\; \mathbf{x}_0 + \int_0^s \vec{\mathbf{v}}\big(\mathbf{x}^{p}_t(\sigma),\ t+\sigma\big)\ d\sigma \tag{4.2}$$

The **streak line** in $\mathbf{x}_0$ and t is defined as the line built by individual traces of particles which were injected at successive points in time. These individual traces can be described by using the path line representation:

$$\begin{aligned}\mathbf{x}^{sk}_t(s) \;&=\; \mathbf{x}^{p}_{t-s}(s)\\ &=\; \mathbf{x}_0 + \int_0^s \vec{\mathbf{v}}\big(\mathbf{x}^{p}_{t-s}(\sigma),\ t-s+\sigma\big)\ d\sigma \end{aligned}\tag{4.3}$$

Fig. 4.7 shows this relation between a streak and a path line in two different velocity fields. According to [WTS*07], the term *generalized streak line* applies to a streak line that is built using a starting point $\mathbf{x}_0$ that changes in time.

Both stream and path lines (Eq. 4.1 and 4.2) can be generated in one pass. Consider the Runge-Kutta operator $\mathtt{RK}^4\big(\mathbf{x},\ t,\ \Delta\big)$ which calculates a new location according to a given starting point $\mathbf{x}$, an instant in time t and a time step Δ (see Eq. 1.7). Then the following recursion provides an approximation of $\mathbf{x}^{sm}_t$:

$$\begin{aligned}\mathbf{x}^{sm}_t(0) &= \mathbf{x}_0\\ \mathbf{x}^{sm}_t(s+\Delta) &= \mathbf{x}^{sm}_t(s) + \int_s^{s+\Delta} \vec{\mathbf{v}}\big(\mathbf{x}^{sm}_t(\sigma),\ t\big)\ d\sigma\\ &\approx \mathbf{x}^{sm}_t(s) + \mathtt{RK}^4\big(\mathbf{x}^{sm}_t(s),\ t,\ \Delta\big)\end{aligned}\tag{4.4}$$

(For $\mathbf{x}_t^p$, modify the time parameter according to Eq. 4.2.) Unfortunately, streak lines cannot be constructed this way due to the dependence of the integrand on the integral bound s in Eq. 4.3. However, it is possible to deduce a location $\mathbf{x}_{t+\Delta}^{sk}(s+\Delta)$ from the previous line location $\mathbf{x}_t^{sk}(s)$:

$$\begin{aligned}
\mathbf{x}_t^{sk}(0) &= \mathbf{x}_0 \\
\mathbf{x}_{t+\Delta}^{sk}(s+\Delta) &= \mathbf{x}_{(t+\Delta)-(s+\Delta)}^{p}(s+\Delta) = \mathbf{x}_{t-s}^{p}(s+\Delta) \\
&= \mathbf{x}_{t-s}^{p}(s) + \int_{s}^{s+\Delta} \vec{\mathbf{v}}\big(\mathbf{x}_{t-s}^{p}(\sigma),\ t-s+\sigma\big)\ d\sigma \\
&= \mathbf{x}_{t}^{sk}(s) + \int_{s}^{s+\Delta} \vec{\mathbf{v}}\big(\mathbf{x}_{t-s}^{p}(\sigma),\ t-s+\sigma\big)\ d\sigma \\
&\approx \mathbf{x}_{t}^{sk}(s) + \mathtt{RK}^4\big(\mathbf{x}_{t}^{sk}(s),\ t,\ \Delta\big)
\end{aligned} \tag{4.5}$$

Using this scheme, however, adaptivity is hard to achieve as all line points, including those added in earlier steps, are moved during integration.

4.2.4 Line Construction

This section discusses the construction of stream, path and streak lines. The according algorithms are presented in GPU specific terms, because the applicability to parallel graphics hardware is one major aspect of the approach. The line generator relies on the geometry shader (see Sec. 1.1.1). This programmable unit is able to process a stream by adding and removing elements. The stream is variable in size, thus it is ideal for representing adaptive line data.

Stream and Path Lines

From a theoretical point of view, Eq. 4.4 can be used to generate stream and path lines in one pass. The input to this pass consists of a set of line seeds which are processed in parallel.

Unfortunately, the number of elements that can be generated in a geometry program is limited to a constant number. Thus, in order to get a line of arbitrary length, we have to break the creation into several passes. This idea is sketched in Fig. 4.8. The selective transform feedback used in the diagram is explained below.

Selective transform feedback (briefly *selective TF*) selects a subset of the geometry program's output elements to be written into a second independent stream. In the following, selective TF is explained in respect of line creation. Breaking the geometry pass implies a splitting of, first, the line data used for rendering and, second, the seed information (i.e. the last point generated for each line) used as input for consecutive subpasses. Note that the seed information is not located at the end of the overall line data, because parts of different lines are generated and stored in an interlocked way. The geometry shader does not directly support the output into two differently structured streams. However, it is possible to generate a transform feedback stream

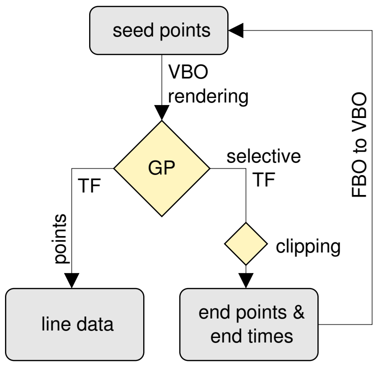


Figure 4.8: *Generating a stream (or path) line. GP: geometry program, TF: transform feedback, VBO/FBO: vertex/fragment buffer object.*

while simultaneously rendering primitives into the current frame buffer. The ID of the current input seed (given by `gl_PrimitveID`) determines the line that is generated in the current geometry program instance. Using this ID, it is possible to write the last point on a per-line basis into the frame buffer. The frame buffer content is then interpreted as new seed input for the next subpass. Time stamps are stored together with the seed location in order to ensure the processing to be correctly continued in the next step. Superfluous points rendered when pushing elements into the TF are discarded by moving them outside the clipping area.

Alg. 4.1 lists the complete algorithm in pseudo code. In each frame, the whole line is generated (in contrast to the generation of streak lines). The iteration stops when no more elements are generated in the geometry program, i.e. if the desired line length has been reached in the previous pass. Querying the number of generated elements is a subfeature of `OpenGL`'s transform feedback.

Algorithm 4.1 (stream/path line algorithm)

```
for each frame {
  emit seeds $\mathbf{p}_1, \dots, \mathbf{p}_n$                  // chosen by user interaction
  while query $\neq 0$ do {
    query $= 0$
    process each seed $\mathbf{p} \in \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$:
    geometry program + TF:
      $s = \mathbf{p}.s$                              // using time stamp $\mathbf{p}.s$
      while s < line_length {
        emit line point $\mathbf{p.x} = \mathrm{RK}^4(\mathbf{p.x},\ t,\ \Delta)$   // with adaptive $\Delta$
        $s$ += $\Delta$
        query++
      }
      emit new seeds $\mathbf{p}_1, \dots, \mathbf{p}_n$              // using selective TF
  }
}
```

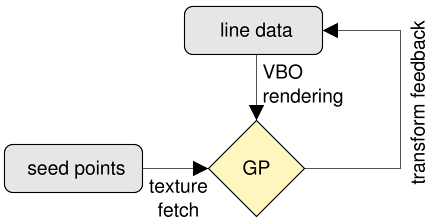


Figure 4.9: *Generating a streak line. GP: geometry program, VBO: vertex buffer object.*

For path lines, replace the time parameter of the $\mathtt{RK}^4$ operator by $t+s$. Note that the number of iterations of the inner *while* loop is bounded by a constant limit specific to the hardware. The adaptive time step Δ is determined using the step-doubling approach (see Sec. 1.2.2) or it is read from the flow data set. See Sec. 4.2.5 for a discussion of data set specific aspects, including texture synchronization.

Streak Lines

Streak lines are generated according to Eq. 4.5. The line data of the previous frame form the input of the next frame. As the line data are used for processing, a time stamp is added to *all* points so that the length of the line can be controlled. The streak line approach is sketched in Fig. 4.9.

The main idea of the algorithm is to add new points (fetched from a seed texture) at the beginning of each line and to remove points at the end according to the desired line length. Thus, the geometry processor must distinguish between starting, inner and end points of the line. Our implementation uses special markers in the line data, which are also motivated by the rendering process. The exact structure of the line data is discussed in Sec. 4.2.7. The streak line algorithm scans the stream for markers and writes the new line data directly through transform feedback:

Algorithm 4.2 (streak line algorithm)

```
initialize line data
for each frame do {
  process each line data segment $(\mathbf{p}_1, \mathbf{p}_2)$:
  geometry program + TF:
    if  $\mathbf{p}_2$ is a starting point ($\mathbf{p}_1$ is marker):
      emit new seed by texture fetching
    if  $\mathbf{p}_1$ is an end point ($\mathbf{p}_2$ is marker) {
      if  $t - \mathbf{p}_1.s > \mathtt{line\_length}$:   remove $\mathbf{p}_1$
      emit $\mathbf{p}_1 = \mathtt{RK}^4(\mathbf{p}_1,\ t,\ \Delta)$                // possibly adaptive
    }
    if  $\mathbf{p}_1$ and $\mathbf{p}_2$ are inner points {
      emit $\mathbf{p}_1 = \mathtt{RK}^4(\mathbf{p}_1,\ t,\ \Delta)$                // possibly adaptive
      perform refinement
    }
}
```

Alg. 4.2 has to synchronize at a fixed time step Δ in order to ensure the correct line output (see Eq. 4.5). In other words, when computing a point $\mathbf{x}_{t+\Delta}^{sk}(s+\Delta)$, the time step Δ is not only a time step for numerical integration but also the time step controlling the injection of new seeds. Consequently, adaptive time-stepping cannot be applied directly to the streak line approach. The proposed interpretation of adaptive streak lines involves two strategies:

1. Time-adaptivity: The time step Δ can be divided into several adaptive substeps in order to achieve a more accurate result in tracing individual points.
2. Geometric adaptivity: In order to avoid sharp bends and oversampling (see Fig. 4.6a), a refinement scheme is applied during the algorithm in line 13.

The geometric refinement step compares the distance between two adjacent points $\mathbf{p}_1, \mathbf{p}_2$ as processed in Alg. 4.2. If the distance is larger than a user specified threshold, then a new point is added in-between by linear interpolation of $\mathbf{p}_1$ and $\mathbf{p}_2$. If the distance is smaller than half the threshold, the point $\mathbf{p}_1$ is removed. This simple scheme has shown to be very efficient and essential for generating reasonably smooth streak lines.

4.2.5 Flow Data Set Storage and Synchronization

The flow data set is partitioned into several time records. Each of them is stored in a 3D texture holding the velocity data. In addition, an adaptive time step is stored at each location of each time record. This time step is computed according to the step-doubling approach (see Sec. 1.2.2) for every texture location within a separate preprocessing.

Binding several textures in a shader program can have a large impact on the performance: Conditional texture selection (using `if`) is significantly more expensive than fetching one single texture. This has been verified on a GeForce 8 graphics card: In an experiment, a single conditional fetch among 4 textures is about 3 times faster than among 16. `OpenGL`'s *texture array* extension supports a presumably faster solution, however it does not support arrays of 3D textures. Thus, it is essential to bind only a few 3D textures in each render pass. This restriction particularly affects the generation of path lines, as a path line may cover several time records.

The restriction to a small set of flow textures yields a synchronization problem: Only a limited time interval can be accessed by a path line generated in one render pass. It turns out that this restriction can be fulfilled by just adding an additional stop criterion in line 8 of Alg. 4.1 which ensures the right time interval. When `query` signals that no more vertices have been output, it is safe to increment the data set time interval to different textures. This adds a surrounding loop incrementing the interval until `query` remains 0 in two subsequent render passes. Note that this approach is efficient because a render pass with `query` $= 0$ is inexpensive.

4.2.6 Flow Data Sets Exceeding GPU Memory

The handling of data sets which are too large to be entirely stored in GPU memory is a task that requires special attention. Only those parts of the flow data which are momentarily necessary should be present in graphics memory. While a spatial division into bricks is rather hard to achieve for arbitrary lines, the idea of lazily transferring time records maps well to the line generator: The synchronization method proposed in Sec. 4.2.5 ensures that the generator breaks whenever a necessary time record is missing. Before the next iteration, the framework can upload appropriate time records so that the line generation is continued in the next pass.

In order to speed-up this scheme, `OpenGL`'s *pixel buffer object* extension is used for asynchronously transferring the texture data using intermediate system memory (compare with Sec. 4.1.6). Additionally, multiple textures can be used to cache time records which are likely to be used in subsequent iterations. For path line generation involving several time records during the generation of one line, this can be of great benefit as one can see in the results (Sec. 4.2.8).

4.2.7 Line Storage and Rendering

The line renderer requires a specific storage of the line data stream. More precisely, each rendered segment $(\mathbf{p}_1, \mathbf{p}_2)$ is provided with its adjacent points, such that normals can be computed for both $\mathbf{p}_1$ and $\mathbf{p}_2$. These normals can be used for lighting and for a sophisticated rendering of tube-like geometries. Additionally, the storage layout is designed to support the splitting of lines into distinct parts, which is necessary in Alg. 4.1, and it supports end point markers for Alg. 4.2.

Line Data Storage

A special `BREAK` symbol identifies line parts generated by Alg. 4.1. This break element also contains an adjacent point used for rendering. The `SEP` symbol marks the end points of each line, notifying the renderer to terminate the geometry appropriately. The following list is an example of how a stream or path line can be represented. The + operator specifies a combination of vertex and marker information.

$$\begin{array}{ll} \texttt{SEP},\ \mathbf{x}_1,\ \ldots,\ \mathbf{x}_5 + \texttt{BREAK}, & \text{(part 1)} \\ \texttt{SEP},\ \mathbf{y}_1,\ \ldots,\ \mathbf{y}_7 + \texttt{BREAK}, & \\ \mathbf{x}_3 + \texttt{BREAK},\ \mathbf{x}_4,\ \ldots,\ \mathbf{x}_6,\ \texttt{SEP}, & \text{(part 2)} \\ \mathbf{y}_5 + \texttt{BREAK},\ \mathbf{y}_6,\ \ldots,\ \mathbf{y}_9,\ \texttt{SEP} & \end{array}$$

Fig. 4.10 shows how the line parts 1 and 2 are combined to one geometry during rendering.

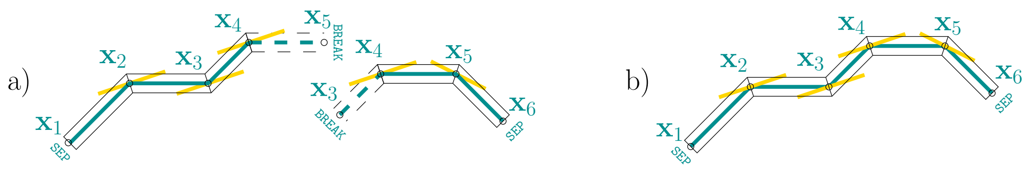


Figure 4.10: *How the renderer combines two line parts (a) to one single line (b). Due to adjacent points stored in the* `BREAK` *elements, the renderer can compute proper normals.*

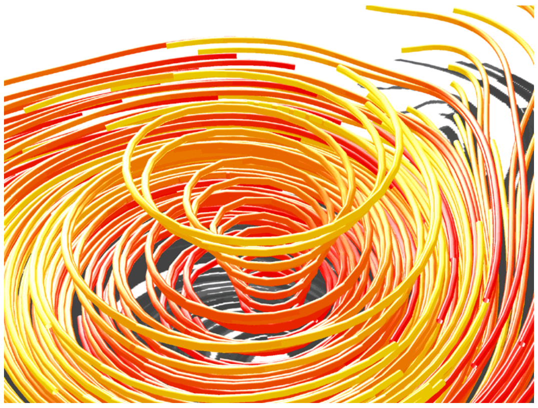


Figure 4.11: *The stylized rendering combines quadrilateral segments with explicit tube geometries (approx. 20 FPS).*

Separator elements (`BREAK` and `SEP`) are marked by setting the fourth component of the according vertex to a predefined large constant number. Streak lines require an additional identifier for distinguishing lines (see Alg. 4.2, line 6). This ID is stored in the starting and ending separator of each line. Streak line data also includes time stamps for all points, which are stored in a separate (texture coordinate) stream.

It should be noted that the exact generation of the line data requires minor modifications to Alg. 4.1 and Alg. 4.2 in order to produce a correct sequence of points, `BREAK`s and `SEP`s. This aspect has been omitted due to its implementation specific nature and in favor of a clear explanation.

In total, the memory consumption for the line data is asymptotically bounded by the number of segments necessary for visualizing all lines. In the implementation, a fixed limit must be given to transform feedback, which was chosen to be 262,144. An overflow can be detected by using an `OpenGL` query. Thus, the algorithm could be adapted to dynamically reinstantiate the buffer if more segments are produced.

Rendering

The renderer directly visualizes the line data as described in the previous part. A geometry shader functionality called line adjacency is used to compute normal information. This can be used for lighting and for a non-trivial rendering of the lines. The stylized line approach by Stoll et al. producing quadrilateral tube [SGS05] approximations has been chosen. These quads are directly generated in the geometry program. The code is even able to generate explicit tube geometries in intricate situations when the view vector coincides with the tangent of a segment (see Fig. 4.11).

4.2.8 Evaluation

In the following, the line-based flow visualization is evaluated by presenting performance and accuracy results for different examples. The hardware used for testing is a PC with a GeForce 8800 GTX graphics card, except for the tests featuring large

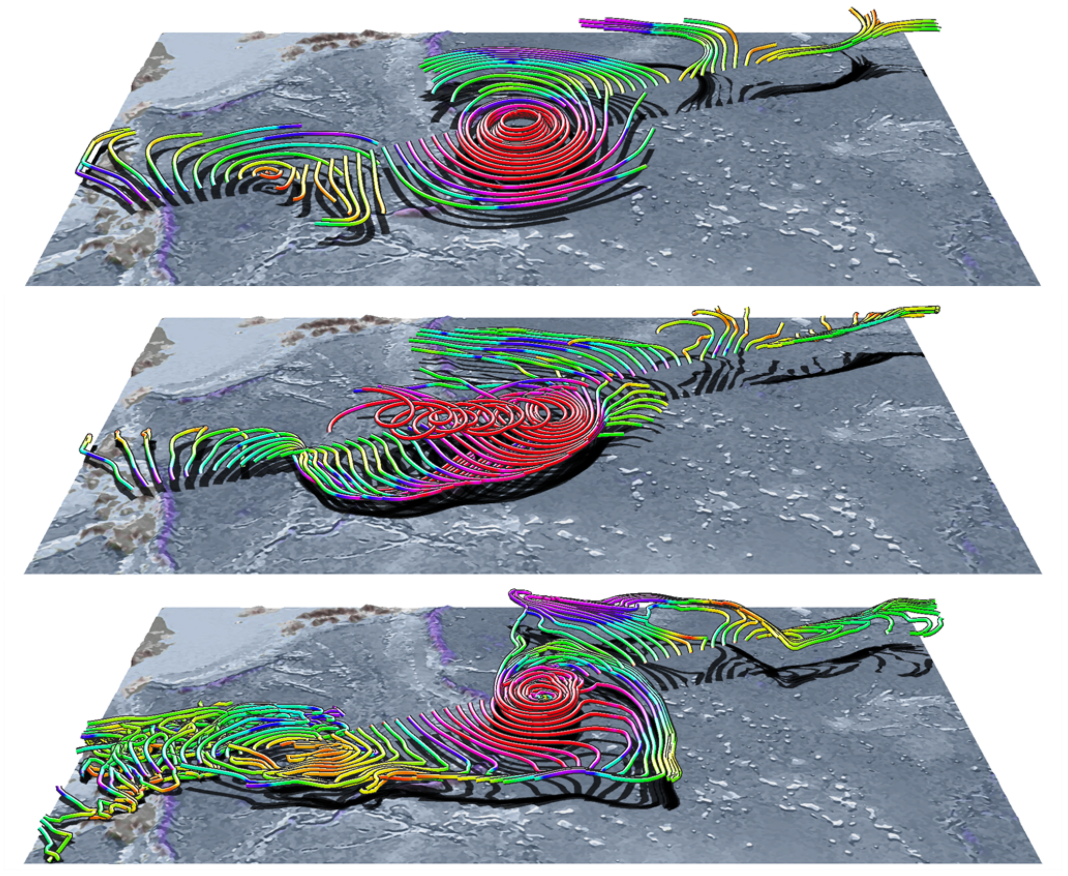


Figure 4.12: *The typhoon flow visualized using adaptive stream, path and streak lines and stylized rendering, number of lines: 64, frame rates (in FPS): 74.89, 37.38, 96.22, segments (approx.): 3,780, 5,900, 9,500.*

flow data set tests. The latter have been performed on a PC with 2 GB RAM and a GeForce 9600 graphics card. This computer has been chosen because of its very stable multi-disk setup which was crucial for memory swapping purposes.

The main example consists of the unsteady typhoon data set which was introduced in Sec. 4.1.1. Since two different adaptive time steps, namely one for stream and one for path and streak lines, must be stored, the total size of the data set adds up to $106 \times 53 \times 39 \times 5 \times 4$ bytes $= 134$ MB (5 components and 4 byte floats). Slices are arranged in a non-uniform way (see Sec. 4.1.5). This problem is solved by performing a binary search in the program in order to find the correct slice when sampling the data. Fig. 4.12 shows the entire visualization including the terrain heightfield and line shadows.

A performance evaluation (see figure captions) reveals that the streak line generator is faster than the stream/path line generator when comparing the number of segments produced. This happens because the streak line approach processes several *segments* in parallel while the stream/path approach processes several *lines* in parallel. Obviously, the number of segments is much higher than the number of lines.

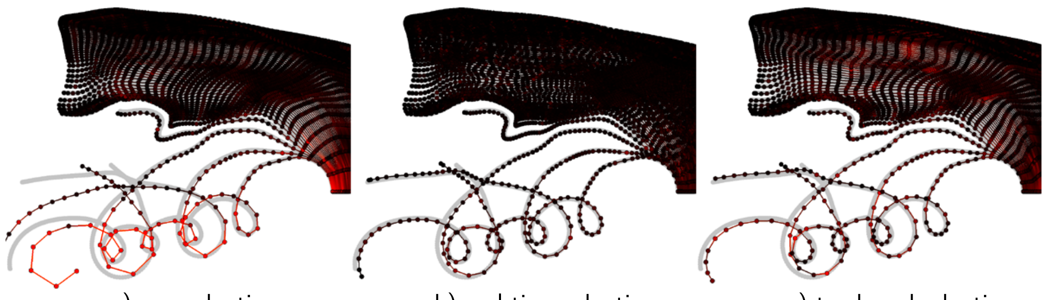


Figure 4.13: *The local step-doubling error for the typhoon example. The error is marked in red, dark regions are accurate and strongly colored regions show a larger error. Frame rates (in FPS, without error computation): 37.28, 11.94, 37.83, number of segments: 11,776, 13,886, 10,505.*

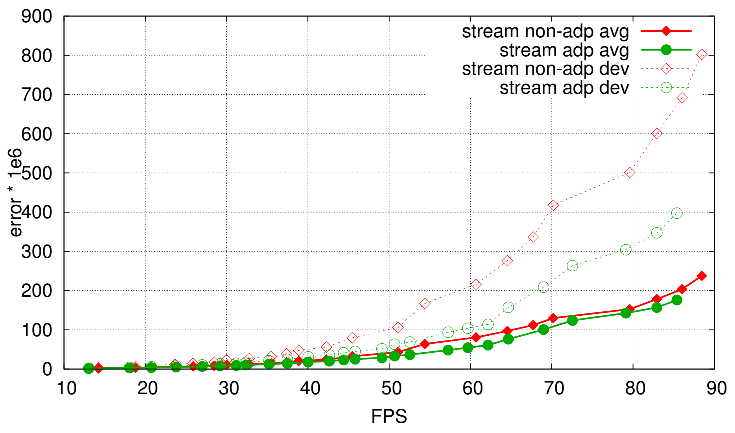
In order to get an impression of the quality of the results, the typhoon example is evaluated both visually and statistically in the next paragraph.

Evaluation of the Typhoon Example

Fig. 4.13 visualizes the local error provided by the step-doubling approach (compare with Sec. 1.2.2), i.e. a measure for the truncation error during a single integration step. It shows an interesting relation between speed and accuracy: the non-adaptive path lines are slower than the adaptive counterpart (with texture-fetched time step), yet exhibiting a much larger error in the difficult twirl region. This advantage of the time-adaptive version in both speed and accuracy is due to the number of segments, which is balanced by the time-adaptivity.

Fig. 4.13 also shows that in comparison to on-the-fly step-doubling, texture-fetching the time step produces a small but visible error due to trilinear interpolation of the time step. The preprocessing storing the adaptive time-step into the textures takes 30 minutes on the CPU and approx. 10 seconds in a GPU implementation. By using our *long double*-based CPU preprocessor, a slight improvement in accuracy can be obtained.

In Fig. 4.14, the local step-doubling error is plotted for different frame rates and line types. For the non-adaptive lines, different values have been measured by varying the integration time step. For the adaptive lines, the step-doubling accuracy has been varied using the GPU-based preprocessor before each measurement. Both diagrams show in a very clear way that adaptive stream and path lines produce both a smaller average error as well as a smaller standard deviation of the error compared to the non-adaptive counterparts at same frame rate. This result coincides with the visual analysis in Fig. 4.13. For streak lines, a similar analysis is hard to provide because of the different nature of adaptivity. Pure time-adaptivity (i.e. without geometric refinement) is insufficient for increasing both speed and accuracy. This is due to



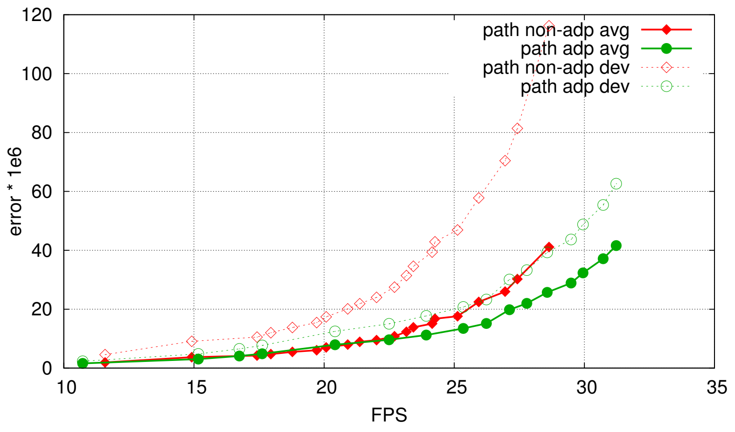


Figure 4.14: *Frame rate versus local error for the example shown in Fig. 4.5, with 256 stream/path lines instead of five path lines. "adp": adaptive, "avg": average, "dev": standard deviation. The error has been multiplied by* 10^6.

the fact that the performance of the streak line generator is less sensitive to a high number of line segments than the one for stream or path lines.

Evaluation of Analytical Flows

In order to provide further results, two analytical flows have been evaluated. Sampling an analytical function avoids the bottleneck of texture fetching, resulting in much higher frame rates. We first consider a flow defined by the function $\vec{\mathbf{v}}(\mathbf{x}, t) = (1, \sin(t) \cdot \cos(\mathbf{x}_x), 0)$. While the adaptive line generator produces approx. 10,000 texture-fetched segments in real-time (stream: 10,076 @ 22.64 FPS, path: 10,023 @ 18.66 FPS, streak: 10,079 @ 91.15 FPS), it can produce more than 10 times more segments when evaluating the analytical function (stream: 115,840 @ 30.99 FPS, path: 103,637 @ 29 FPS, streak: 100,057 @ 126.81 FPS).

In order to get a better idea of the benefit of time-adaptivity, another analytical flow $\vec{\mathbf{v}}_2$ produced by a repetitive cubic B-spline (see Fig. 4.15) has been evaluated. This flow has the useful property that it can be integrated analytically in order to get

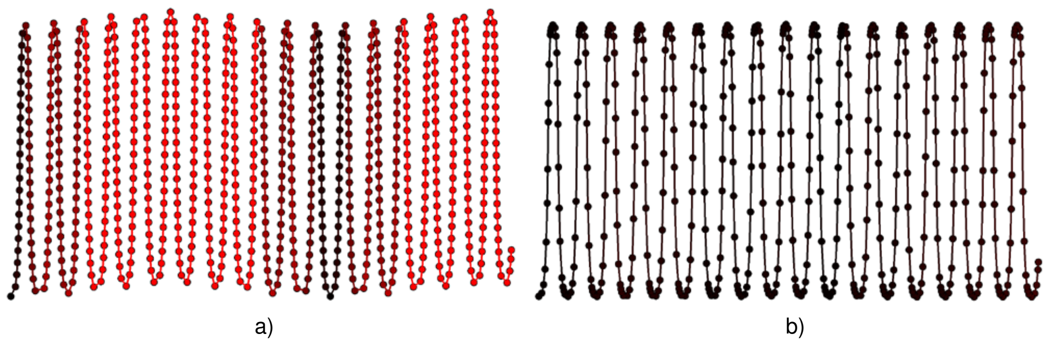


Figure 4.15: *The global error for a cubic B-spline function (marked in red). a) non-adaptive, 47.38 FPS, 776 segments, b) time-adaptive 62.16 FPS, 445 segments.*

an exact path line representation. Velocities are normalized in order to emphasize the difficulty near the curved extrema. See Eq. 1.4 in Sec. 1.2.2 for the formal definition of $\vec{\mathbf{v}}_2$. A global error has been evaluated by taking the distance to exact line locations according to the time parameter of the computed line. The left (non-adaptive) line shows a large error despite the fact that more segments are rendered than in the right line, which adapts well to the characteristics of the flow. Note that the periodic nature of the function causes interferences in the error, thus canceling the error in certain regions.

Evaluation of Large Flow Data Sets

In practice, one might be interested in visualizing large flow data sets. In order to test the applicability of the approach to this task, various data sets of different size have been generated, all representing the same flow defined by the function $\vec{\mathbf{v}}(\mathbf{x}, t) = (1, \sin(t) \cdot \cos(\mathbf{x}^x), 0)$.

Tab. 4.2 shows the timings for all tested data sets. During visualization, one time record covers ten frames. All three types of lines have been tested in different memory configurations: Besides three textures representing the current time records needed for one single Runge-Kutta integration step, texture transfer has been accelerated by dedicating a PBO for asynchronous transfer (second column of the FPS results). A third configuration has been tested using a stack of 8 cached textures. This number has been chosen because eight 150^3 textures plus PBO and the line data (approx. 32 MB in this configuration) amounts to nearly 512 MB of memory, which is the limit of the graphics card used for the tests.

The results show two interesting aspects: First, texture transfer is fairly fast for stream and streak lines. Path lines require multiple switches of the textures during one frame, as lines are covering several time records. Thus, path lines highly benefit from asynchronous transfer and caching of subsequent records. This yields interactive frame rates even for data sets which do not fit entirely in GPU memory. However, one can see that system memory is a bottleneck as soon as disk swapping is involved. While the very large data sets (> 3 GB) can still be visualized with few frames per second in average, fetching necessary time records from the hard disk results in

Table 4.2: *Runtime for stream/path/streak lines in a flow data set of varying resolution. In all tests, 512 lines cover nearly the whole flow volume and at least 4 time records.*

res.	steps	size	FPS (stream/path/streak)		
			normal	async	async & cached
50^3	100	238M	40/26/44	39/38/44	39/93/44
50^3	250	596M	40/26/44	40/39/45	39/83/45
50^3	500	1.2G	40/27/44	39/37/45	39/92/44
100^3	50	954M	55/5/44	59/8/48	53/79/48
100^3	100	1.9G	58/3/44	74/8/47	69/42/47
100^3	200	3.7G	6/5/13	18/5/15	10/11/11
150^3	10	644M	37/1/34	48/3/40	23/55/31
150^3	50	3.1G	5/1/7	4/3/8	6/5/11

Notation: *res.*: resolution of one time record, *steps*: number of time records, *size*: file size of the data set (containing 2 different adaptive time-steps for stream and path/streak lines), *normal*: no use of PBOs, *async*: asynchronous texture transfer using PBOs, *async & cached*: 1 PBO and 8 instead of 3 textures in GPU memory.

unpleasant delays disturbing the animation. For handling such large data sets, a more elaborate memory management or a special hardware for fast memory access would be necessary so that a smooth visualization is ensured.

4.3 Flow Volumes Based on Particle Level Sets

In this section, the GPU-based PLS framework of Chap. 3 is expanded to support volumetric flow primitives. The set of new primitives includes time surfaces, path volumes and streak volumes. The technique has been published in [CKSW08].

Flow volumes are the volumetric equivalent to flow lines. A flow volume is a subset of space, the inner region, with a sharp closed boundary. Level sets are suited for the description of flow volumes, because they ensure a proper representation of closed objects independently of the applied flow. As motivated in Chap. 3, the PLS method significantly improves grid-based level sets w.r.t. to accuracy and volume preservation. This work demonstrates that the PLS technique can be adapted to the visualization by time surfaces and path volumes, and to volumetric dye advection via streak volumes. To achieve this, the PLS framework needs to be extended, including a model for dye injection.

4.3.1 Flow Volumes and the Problem of Dye Advection

Three different types of flow volumes are presented in this section. The most basic one, the time surface, just corresponds to the advection of a closed surface. This type is directly supported by the PLS approach. The second type, the path volume, draws the trace of the initial volume as it is transported through the unsteady flow.

More specifically, a path volume is formed by the union of all states the initial shape is taking during advection. Very similarly, a stream volume can be described as a path volume constructed without advancing the flow in time. The third type, the streak volume, is built by continuously injecting the initial volume while advecting the complete volume at each time step. A visualization of a time surface, a path volume and a streak volume in a simple sinusoidal flow is given in Fig. 4.16.

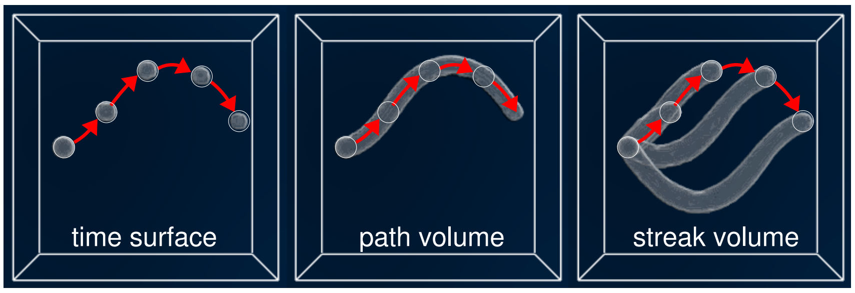


Figure 4.16: *Comparison of a time surface, a path volume and a streak volume (also called dye advection). Note the similarity to path and streak lines (Sec. 4.2).*

The concept behind streak volumes is also called dye advection, as an initial kernel, the dye, is constantly inserted. Fig. 4.17 compares previous dye advection techniques with the technique proposed in this section. Most previous work on interactive dye advection [JEH02, vW02] is based on semi-Lagrangian grid advection, which is affected by numerical diffusion due to repeated resampling via trilinear reconstruction (Fig. 4.17a). The extension to level set based dye advection (Fig. 4.17b) eliminates the diffusion problem, as the level set function is constantly reinitialized. However, the problem of volume loss remains, despite dye injection. The PLS method can be used to reduce volume loss (Fig. 4.17c).

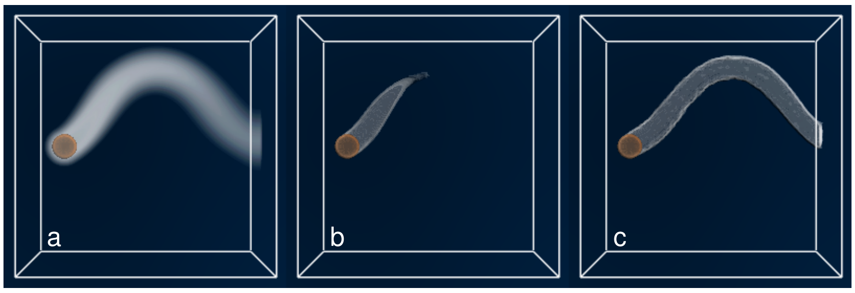


Figure 4.17: *Dye advection of a spherical volume in a time-dependent sine wave. The injection volume is marked in red. The flow spreads the dye over time and the resulting trace depends on the computation of this process. Diffusive advection (a), level set advection (b), PLS advection (c) – (64^3 grid, 262,144 particles, 78.2, 55.9, 36.6 FPS).*

This section contains the following contribution: the PLS method is adapted for the visualization of 3D unsteady flow by extending it to the representation of time surfaces, path volumes and streak volumes. Consequently, a fast dye injection mechanism for the hybrid grid-particle model of PLS is introduced. Using PLS, the numerical diffusion, and thus the volume loss, can be minimized in order to achieve high quality flow volumes.

4.3.2 Prior Work

An early example of stream volumes is described by Max et al. [MBC93]. They use an explicit volume representation based on tetrahedra. Unfortunately, such explicit representations are difficult in the case of intricate flow, because adaptive removal and addition of vertices and changes of the topology need to be considered. Even the simpler problem of stream surfaces already requires advanced algorithms to handle these issues [Hul92, GTS*04]. Point-based representations of stream surfaces and path surfaces avoid issues of mesh connectivity but still require a complicated point generation and removal [STWE07].

In contrast, implicit representations easily allow topological changes and do not require control of vertex arrangement and density. Examples include implicit stream surfaces by Van Wijk [vW93], the particle travel time method by Westermann et al. [WJE00], and the application of direct volume rendering to visualize implicit representations according to Xue et al. [XZC04]. Texture advection (see Sec. 1.3.1) can be modified in the form of 2D image-based flow visualization [vW02] and 2D Lagrangian-Eulerian advection [JEH02], which both support texture-based dye advection in order to generate streak lines. Texture advection can be extended to GPU-based 3D flow visualization as well [TvW03, WSE07]. For example, 3D dye visualization can be employed to highlight features [SJM96]. One issue of most texture advection methods is numerical diffusion due to resampling. Lagrangian-Eulerian advection [JEH02] addresses this problem by frequently restoring the contrast of the transported dye. An alternative approach is the use of distance field level sets in combination with level set reinitialization, which leads to a non-diffused dye background interface but is affected by volume loss [Wei04]. For an overview of texture-based flow visualization in general, the reader is referred to Laramee et al. [LHD*04].

4.3.3 Flow Volume Construction

The task of applying PLS to interactive flow visualization poses several challenges: First, fast parallel algorithms are necessary for level set reinitialization, particle reseeding, and the interchange of data between grids and particles. Second, there is a trade-off between speed and accuracy. Accuracy in the context of PLS means no volume loss and the preservation of surface features. Third, time surfaces, path, and streak volumes require different handling of the grid and the particle structure and thus have to be considered separately. Finally, streak volumes (i.e. dye advection) require special attention in order to synchronize the grid and the particle structure.

The first aspect is solved by the framework presented in Chap. 3. The remaining tasks are addressed in the following paragraphs. As all modifications remain in the spirit of the original PLS, we will refer to the steps in Alg. 1.1.

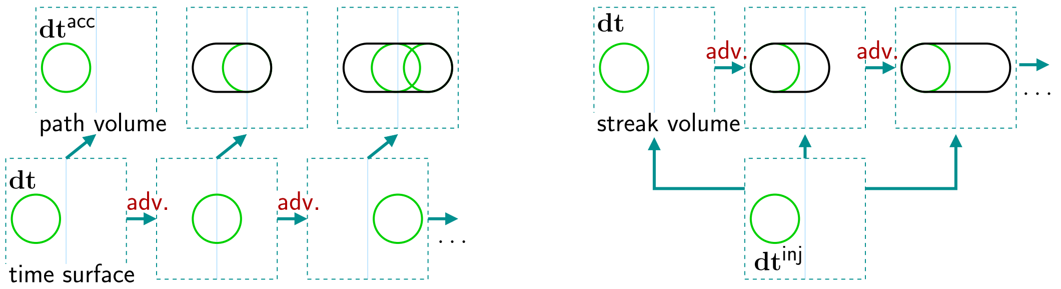


Figure 4.18: *Schematic illustration of a path volume advection (left) and a streak volume advection (right). Colors: green is the initial/injection volume, black is the path/streak volume. The relation between time surfaces and path volumes is made clear in the left image.*

Time Surfaces

The PLS framework proposed in Chap. 3 includes all steps required for the visualization of time surfaces, i.e. a bounded volume moving in a flow while changing its shape.

Path Volumes

A path volume can be interpreted as the accumulation of the volume left behind by a time surface (compare with Fig. 4.18, left). This idea fits well into the PLS algorithm. The path volume is accumulated into a separate grid $\mathbf{dt}^{\text{acc}}$ in an additional step after step 6 in Alg. 1.1. The accumulated grid is the result of a minimum operation on the distance component of the previous $\mathbf{dt}^{\text{acc}}$ and of the current $\mathbf{dt}$, taking the according reference $\mathbf{dt}_\delta$. The result of this operation is a union of the negative distances, thus of the inner regions of both level sets. An example is given in Fig. 4.19, left.

It should be noted that only one accumulation step is done in each frame. Thus, the path volume is steadily growing during visualization, in contrast to the path lines (Sec. 4.2) being constructed en bloc after each time step.

Streak Volumes

Streak volumes are produced by repeatedly injecting a volume $\mathbf{dt}^{\text{inj}} = (\mathbf{dt}_d^{\text{inj}}, \mathbf{dt}_\delta^{\text{inj}})$ to the current level set (compare with Fig. 4.18, right). The injection $\mathbf{dt}^{\text{inj}}$ is usually generated analytically using implicit geometries. The volume injection involves an update of both the grid and the particle representation of the level set.

The grid update is similar to path volume accumulation. A minimum of $\mathbf{dt}^{\text{inj}}$ and $\mathbf{dt}$ computes the union of both volumes after step 5 of Alg. 1.1. After step 6, the particle set is extended in order to cover the new interface added by the injection. Two methods have been tested: 1. explicitly adding new particles at the injection's interface or 2. relying on the standard particle reseeding. In both cases, the reseeding scheme presented in Sec. 3.3.4 can be adapted by binding the appropriate DT texture. The second approach yields nearly as good results as the first one in all

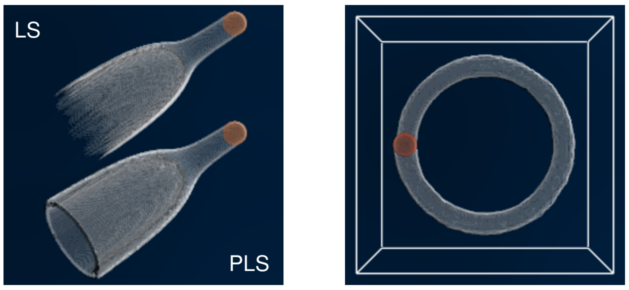


Figure 4.19: *Left: A path volume is pushed into a tight band around a flow source. Due to numerical diffusion, the LS volume disappears. In contrast, PLS maintains the volume. Right: A difficult case where the streak volume reaches the injection volume. Both: The injection is marked in red.*

examples, despite the fact that the particle density near the injection is lower. The reason is that the injected volume is less susceptible to numerical diffusion because it is reemitted in each frame.

Both injected particles and old particles can disturb the PLS correction. In Fig. 4.20, those wrong particles are marked in red. The green circle stands for the injection volume, which partly overlaps the advected volume (blue circle) created in the last level set advection step.

Fig. 4.20a shows the situation when injecting new particles. Wrong particles are those with $\mathbf{dt}_d(\mathbf{x}_p) < r_p$. They can be efficiently reseeded during the injection step by using the unified DT. Fig. 4.20b shows wrong particles coming from the last PLS step. They can be identified by checking $\mathbf{dt}_d^{\text{inj}}(\mathbf{x}_p) < r_p$ in a separate pass over all particles. The identified particles are either removed or reseeded. Removing them is cheaper and can be achieved by moving them out of the volume. However in some examples, too many particles might be lost in this approach.

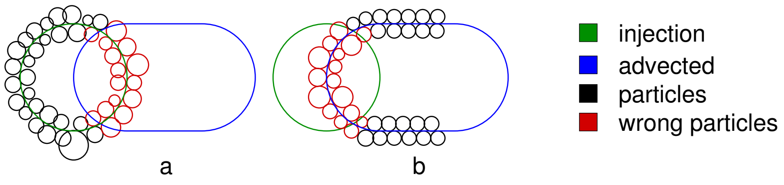


Figure 4.20: *Two cases (in 2D) where particles must be removed or reseeded when generating streak volumes (a: injected particles, b: old particles).*

Fig. 4.17 and Fig. 4.21 show examples of streak volumes. The right side of Fig. 4.19 shows that the reseeding scheme even handles the difficult case where the streak volume reaches the injection area.

4.3.4 Rendering

The volume renderer is based on a back-to-front slicing technique using view-aligned polygons. Only fragments within a small iso-value range $[-\epsilon, \epsilon]$ around the interface

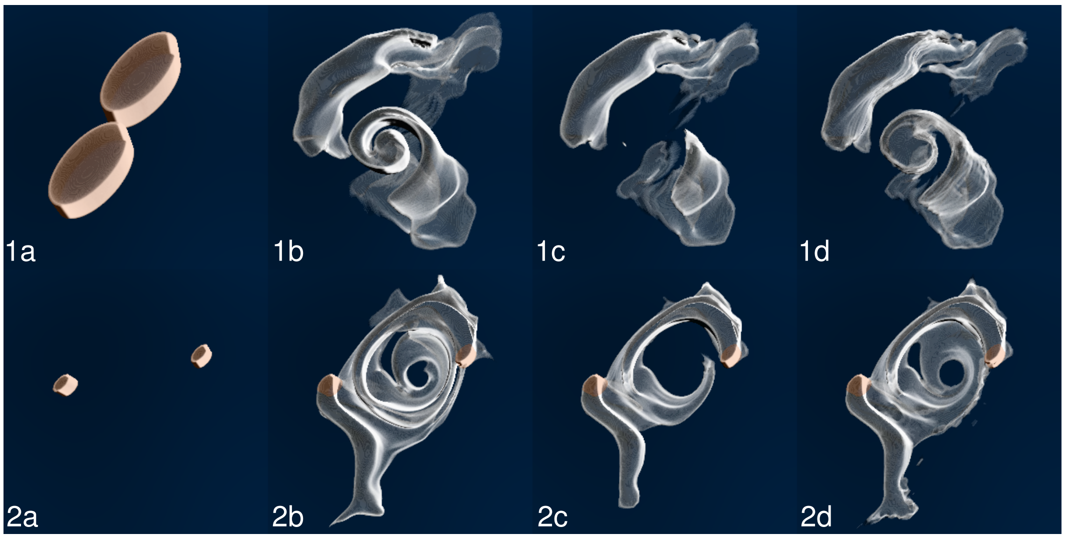


Figure 4.21: *Top row: Time surface after 190 advection steps. Starting geometry (1a), ground-truth using* 128^3 *LS (1b),* 64^3 *LS (1c), the enhanced* 64^3 *PLS (1d), 262,144 particles (FPS for 1b–1d: 10.6, 19.1, 16.5, time without rendering (in ms): 51.6, 11.1, 22.1). Second row: Streak volume after 275 advections, same parameters (FPS for 2b–2d: 12.5, 28.2, 22.1, time without rendering (in ms): 52.1, 10.1, 19.7). Both: The starting/injection shape is marked in red.*

are colored using Phong lighting, for which normal vectors are determined by computing gradients using a central difference scheme. Applying alpha blending makes internal structures visible, which is important e.g. for complex flow volumes. This approach is similar to semi-transparent interval volume rendering [FMST96]. When using appropriate semi-transparent transfer functions, the visual results of rendering PLS resemble those of illustrative techniques that highlight 3D flow structures, such as [SJEG05], which supports the spatial perception of flow volume boundaries.

4.3.5 Evaluation

The flow volumes engine has been tested on a GeForce 8800 GTS graphics card. For an evaluation of the GPU-based PLS framework, refer to Sec. 3.3.8. Fig. 4.21 shows the results of a time surface and a streak volume in the unsteady typhoon flow from Sec. 4.1.1. Both rows compare the pure LS and the GPU-based PLS method, showing the advantage of PLS when using the same grid resolution.

The typhoon data set exhibits prominent swirling features, which can be depicted by both time surfaces and streak volumes. The latter are particularly useful because they resemble the well known dye advection metaphor from experimental flow visualization and they show the temporal evolution of the flow. Fig. 4.21 demonstrates that the high quality and volume preservation of the PLS approach is critical for showing all details of the swirling features, whereas LS fails to depict those details.

The performance of streak volume injection depends on the applied reseeding

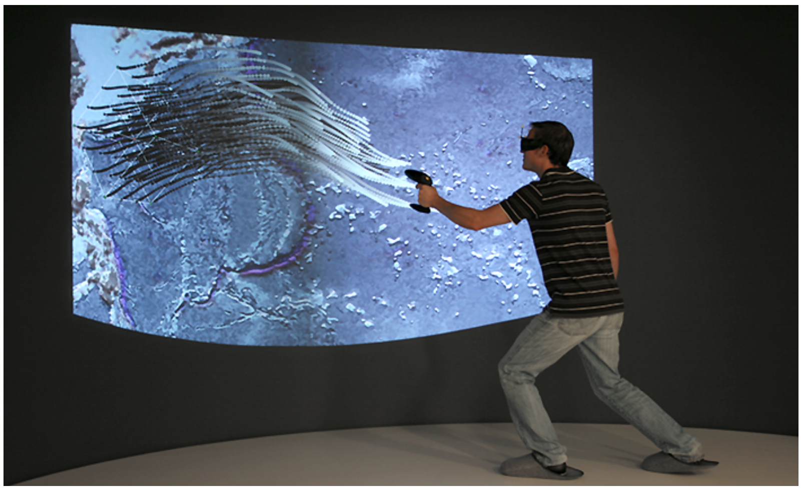


Figure 4.22: *Flow visualization of a typhoon in the VR laboratory of the University of Siegen, including user interaction using tracking devices. The glasses are tracked in order to render the correct point of view. The so-called flystick controls the placement, orientation and size of the emitter box from which streak lines originate.*

scheme. In general, the time consumed for reseeding is in the same range as general PLS reseeding. Dye injection takes about 5 percent, dye reseeding about 2 percent of the time for the overall algorithm in the example shown in Fig. 4.17c.

4.4 Virtual Reality Support

The usability of the flow visualization system presented in the previous sections has been enhanced by adding the support for virtual reality (VR) environments.

The VR laboratory in which the application has been tested is located at the University of Siegen. It is characterized by its elegant curved projection area and a design allowing single user interaction *or* group presentations. The 3-dimensional immersion is increased by a stereoscopic projection. In single-user mode, the glasses are tracked in order to provide the correct viewing transformation at the user's position.

The application has been ported to the VR environment using the `equalizer` toolkit [EMP09]. The main challenge when distributing processing to multiple nodes (in this case seven PCs) is to synchronize any input for simulation and rendering. This implies that a stream of information must be transferred to all client nodes. This stream contains all user inputs like parameter adjustment or navigational commands. In the application, a very generic information flow has been realized which can be directed to the network or to a file storing the current configuration. This allows synchronization in the VR cluster as well as a load-and-save functionality. Together with temporal markers, this stream can also be used to replay a specific sequence of commands for demonstration purposes. All computations, e.g. particle tracing, line generation or PLS advection, are duplicated on each node. Clearly, the use of multiple nodes would have the potential to significantly improve the performance. However, this would require synchronization routines depending on the computation.

Stereo viewing, user tracking and the so-called *flystick* can be used to improve the usability of the flow visualization. Clearly, the ability to examine the scene from any point of view in 3D helps to distinguish the spatial location of flow primitives and thus flow features. The flystick is a tracking device which can be moved in space freely, and it possesses special controls, i.e. a navigational apparatus and a set of buttons. In the example shown in Fig. 4.22, the user can pick up and move the particle emitter by holding the flystick into the scene, pushing a button and then adjusting the location, orientation and size. This way, the user can precisely choose the region of interest and visualize specific local sections in the flow field.

Various geometric flow visualization techniques have been presented in this chapter. It has been shown that high quality *and* real-time control is achievable by using the parallel concepts of modern graphics hardware.

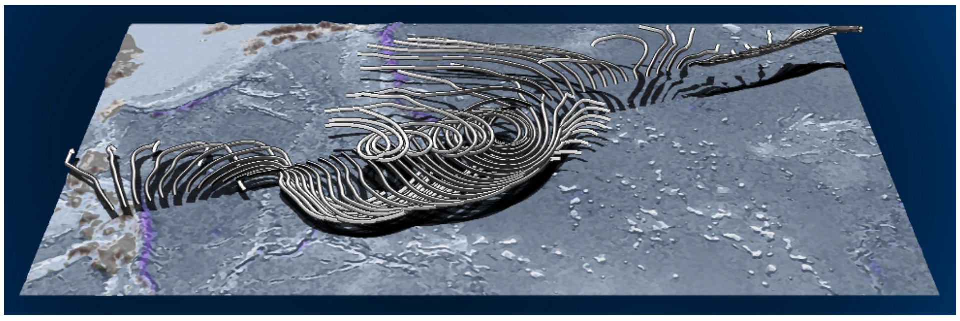


Figure 4.23: *An example of advanced geometric flow visualization: Path lines are moved in real-time within the unsteady typhoon flow (data set: courtesy of MPI for Meteorology).*

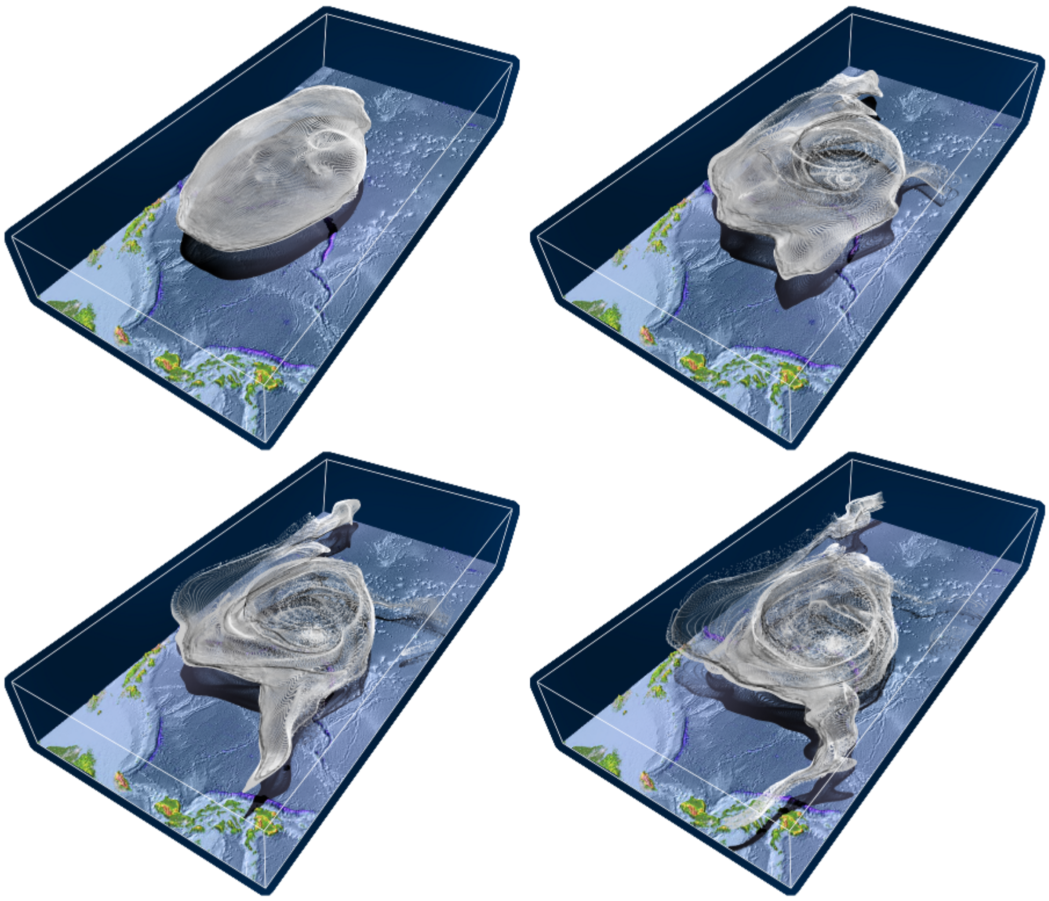


Figure 4.24: *Layered rendering of a pure particle-based time surface and a corresponding particle level set representation (data set: courtesy of MPI for Meteorology).*

Chapter 5
Epilogue

Simplicity is the final achievement. After one has played a vast quantity of notes and more notes, it is simplicity that emerges as the crowning reward of art.

Frédéric Chopin

In this chapter, an overview of all techniques is provided, with the intention to bring the major aspects together and to reflect both their strengths and weaknesses. As an implication to this discussion, a set of ideas is listed, which reveal the directions in which the research specific to the thesis' context could advance in future. Finally, a conclusion is provided.

Graphics technology is in full growth. Today's graphics hardware can compete with the world's fastest super-computers available ten years ago, and the computational power is steadily increasing. Properly designed algorithms directly benefit from this growth: For example, the size of a particle system that could be processed in real-time three years ago using older hardware has increased to a factor of more than ten since that time. Similarly, the increase of GPU memory makes the use of larger structures and a higher precision possible.

In the same time, conceptual improvements of the hardware, i.e. extensions to the set of features or new programming interfaces, widen the range of possible applications by relaxing the difficulty of designing algorithms which take advantage of the parallel processing paradigm. For example, using general purpose APIs like CUDA allows one to manage memory on a much more fine-granulated level than using shader programs. Another example is the support for double floating point arithmetic, integers and pointers – features which are already available or in discussion. Generally, one can say that GPUs head for general capabilities like those of CPUs, while being based on the concept of multi-core processing and massive threading. Actually, this concept is also present in the development of current CPUs, which are built of up to 8-12 cores, a number which is steadily increasing. The future will show whether the GPU will remain a coprocessor, or whether CPU and GPU will fuse together in favor of a powerful all-purpose chip.

Making heavy use of the outstanding capabilities of the GPU is one of the main topics of this work. Accordingly, everything can be summarized to one purpose, namely the investigation of new GPU-based techniques in the context of fluids and flow visualization. The results show that it is possible to design these technique in such a way as to benefit from the parallel architecture of graphics hardware. However, there is room for improvements, as we will see in the next two sections. And, the active development of graphics hardware will further unclose new possibilities, in particular towards high-quality real-time interactive solutions.

5.1 Summary

In this work, new techniques in the field of GPU-based fluid simulation and flow visualization have been presented.

The first one consists of a concept for dynamically coupling particles (Chap. 2). The concept allows one to solve the problem of processing particle neighborhoods within an SPH-based fluid simulation by interchanging information between a grid and the particle set. In order to compute proper fluid forces, the approach must separate computational steps in a very specific way. The results show that the approach is fast, however, it suffers from numerical problems due to the discretization of fluid forces. Another weakness is the restriction to fixed bounds defined by the grid covering the volume of interest.

The idea of simultaneously using a grid and a particle set on the GPU is also one of the main aspects of the PLS-based surface tracing framework presented in Chap. 3. Like in the original PLS method, the grid-based level set representation of the advected object is improved by reducing diffusive errors and thus improving volume preservation. The key idea is to rely on the information contained in particles located at the object's surface, i.e. the interface, because particle tracing is very accurate. The reformulation of PLS for the GPU significantly improves the efficiency, as it allows fast parallel processing of much larger resolutions and larger particle sets than previous CPU-based approaches. One of the main challenges of realizing a real-time PLS framework is the reinitialization of the level set function. In the presented framework, this task is solved by a new fast algorithm which is based on hierarchical distance propagation. While this algorithm can be seen as a fast, yet approximate, generic solution for computing grid-based distance transform, its fast nature and the closest-point information stored in the DT's references is adequate for efficiently solving another step of the PLS algorithm, namely particle reseeding. The results show that using the DT references for reseeding works well, and that sampling references can improve the accuracy when computing particle radii. The level set correction is realized by scattering the particles into the level set grid in a way resembling the fluid particle coupling approach. In summary, the presented GPU-based PLS framework is fast due to its parallel nature, and it thus can handle large resolutions and particle sets. The main disadvantage of the approach is the high memory consumption of the grid-based distance transform. Another problem, which can be observed when using a rather small number of corrective particles, is that the surface tends to fluctuate due to the local influence of single particles.

The next part, handling flow visualization, is arranged in three blocks, each covering a specific type of flow primitives. In the first block, GPU-based flow particles are amplified to suit the field of climate visualization. This includes a discussion of the requirements and a system providing basic features like different rendering modes, shadowing in combination with a terrain, and the support for a basic non-uniformity of the input grid. The flow data set introduced in this block constitutes the main example used in the remainder of the chapter. The second block is dedicated to the construction of time-adaptive, real-time controlled flow lines. Two GPU-based algorithms are provided for the fast generation of stream, path and streak lines. The main difficulty related to the time-adaptivity is to deal with the asynchronism of flow data access during parallel processing. A new processing scheme called selective transform feedback helps to solve this problem. As stream and path lines are generated in each frame, they can directly benefit from classical time-adaptivity schemes like step-doubling. For streak lines, true time-adaptivity is difficult to achieve. Instead, a concept is proposed which includes inter-segment adaptivity and a refinement step ensuring a balanced curve progression without undesirable peaks. The results show that high-quality flow lines can be visualized efficiently within large flow data sets, even in the case where the size of the data exceeds the available GPU memory. Even larger data sets which do not fit in main memory can be visualized in principle. However, a more elaborate memory management including efficient disk caching would be necessary in this case to ensure a smooth visualization without any delay. The third block extends the GPU-based PLS framework presented before in order to support three new types of volumetric flow primitives: time surfaces, path volumes and streak volumes. The latter, which is also known as dye advection, requires a special scheme for injecting a new volume which is constantly added during visualization. The model for dye injection must take care of maintaining a correct level set representation in both the grid and the particle set. It turned out that the proposed model constitutes a suitable extension to PLS towards the visualization of flow volumes. A weakness of the approach is, again, a high memory consumption for the grid-based distance transform and the problem that smoothed surfaces are difficult to achieve due to the local per-particle correction.

In order to test the usability of the presented techniques, various examples have been given. The typhoon example discussed in the flow chapter is the main case study, showing the applicability in the context of climate flow visualization. An alternative interface with stronger user immersion is provided by the support for virtual reality environments. It has also been shown that PLS surfaces can be combined with flow or fluid particles by using the DT-based level set representation for an efficient collision detection and reaction. This combination handles scenarios where an object is interactively deformed while a particle system, e.g. modelling snowflakes, is moving around. Similarly, a deformable object can be placed within a particle-based fluid. There are, certainly, many more possible applications, which have not been investigated yet.

5.2 Future Directions

This section contains some proposals for possible extensions and improvements, including technical aspects as well as general future directions, investigations and applications.

Clearly, the GPU-based particle fluid solver presented in Chap. 2 exhibits many possibilities for improvements, and fluid simulation is a general subject which is relevant in many physical systems. Many improvements already exist at the time of writing this document, e.g. the more accurate index method by Harada et al. [HKK07b], which is also discussed at the end of Chap. 2. Zhang et al. present an adaptive SPH sampling using the GPU [ZSP08], i.e. supporting different particle radii. Additionally, the CUDA-based implementation of granular media included in the NVidia SDK [Gre08] shows that investigations are active in this field. One of the main problems still remaining is the restriction of the volume of interest to the coverage of the index texture. Fluids represented by a dense particle set, i.e. using a small smoothing kernel, are especially difficult to handle, as a high resolution must be chosen for the grid. One solution to this problem is to cut the grid into slices containing only non-empty regions [HKK07a]. However, redundancies in the storage are still inevitable when the fluid is highly scattered. Consequently, future research could go into the direction of a grid-free neighboring, possibly relying on a pointer structure or on a special kind of particle sorting. Rendering of the fluid is an aspect which is especially important in graphics related applications. For example, an iso-surface instead of a particle-wise rendering would lead to a much more realistic appearance of the simulated medium. Yasuda et al. recently presented a ray casting approach working on the index texture used for particle neighborhoods [YHK09]. The integration of such a rendering approach into a composite scene consisting of fluid elements and other objects like polygonal meshes would be an interesting and practical extension. Real-time caustics would also improve the realism of fluid rendering.

The GPU-based PLS framework presented in Chap. 3 using a texture representation of the level set suffers from a high memory consumption. A narrow band approach based on a page table, like in [LKHW04], or using a hierarchical pointer structure could reduce the memory problem and might even accelerate the algorithm by avoiding processing of unspecified regions. However, such schemes require a careful design in order to keep the overhead at a minimum. Pointer structures are still difficult to handle on the GPU, i.e. without transferring any information to the CPU, but future work will surely provide new solutions in this direction. The original PLS method was designed to trace the exact surface of a fluid. Until now, highly accurate surface tracing based on PLS was a time-consuming task which could be solved only for preanimated sequences, e.g. in the case of computer animated motion pictures. Applying Navier Stokes to the GPU-based PLS framework could move the scope towards a real-time simulation of a fine-granulated fluid surface. This would provide a system for high quality fluids in computer games or similar interactive applications. The bumpy surface problem of PLS surfaces could be addressed by a feature-preserving smoothing technique like the anisotropic diffusion proposed by Preußer and Rumpf [PR02]. This kind of smoothing pass might be able to reduce the effect, however it is clear that avoiding bumps during level set correction would be much more powerful. This could go into the direction of the marker level set method by Mihalef et al. [VMS07], which might provide a smoother object surface.

The flow visualization presented in Chap. 4 is a first step towards a fully functional real-time application for climate visualization. However, future investigations should include even more intuitive user interfaces and advanced means to interactively probe the data. The support of virtual reality environments is one example, and it is clear that such an application could provide a far larger set of tools for flow exploration: For example, an intuitive virtual toolkit could be used to chose specific flow primitives or specific flow attributes to be color-coded. Besides virtual reality, there are many further possibilities which would improve the usability. The use of transfer functions could potentially be of help for the isolation of interesting structures in the flow. Additionally, particle visualization can be enhanced by encoding additional information by the use of vector or tensor glyphs. Such moving glyphs could be efficiently generated using the geometry shader stage of the current graphics pipeline. In order to make the data flow, from climate simulation to visualization, easier and more efficient, a more elaborate support of curvilinear grids should be addressed in future work. The access of very large data directly in the visualization would require advanced caching techniques. This could also involve a bricking technique for transferring smaller portions of the data set. A multi-node approach with several computers or graphics cards, each holding only a part of the flow data, might lead to further improvements in data access.

Next to the particle-based approach, a method for efficiently generating and visualizing adaptive stream, path and streak lines using modern graphics hardware has been presented in Chap. 4. Due to the generic nature of the line generators, it should be possible to incorporate more sophisticated adaptive time-stepping methods that replace step-doubling. Similarly, streak line refinement could be improved without changing the algorithmic idea. This could expand the applicability of the method to areas where an even higher accuracy is necessary.

The last part in Chap. 4 proposes a modified and enhanced GPU-based PLS method for the representation of flow volumes. The marker level set method by Mihalef et al. [VMS07] promises to be superior and might be adapted to dye injection for the visualization of flow volumes. The possibility of coding texture information in surface particles could be used to color-code specific flow attributes. A GPU-based concept for marker level sets has been presented recently [MDHZ08].

5.3 Conclusion

Computer graphics is a research field which is very active and subject to fast developments on both the theoretical and the hardware side. The latter is an important factor, as improvements are not only related to the raw computational power, but also to the conceptual design. The parallel architecture of the GPU requires a new way of thinking. The new paradigm has already reached many applications, still, others might benefit from a GPU-friendly realization, but never have been brought into the GPU world seriously.

The design of GPU-based methods requires new algorithms *and* data structures. One of the difficulties is to provide processing schemes which smoothly cooperate with the data structures. Accordingly, this work has investigated particle systems and their combination with grids. An important statement is that such grid-particle methods have the potential to accelerate the processing of accurate volumetric representations.

As a result from these investigations, it has been shown that such efficient processing schemes can be provided in order to realize fast fluids, fast surface tracing, and a set of various geometric flow visualization techniques.

The other statement which has been made in this work is that GPU-based flow visualization techniques can highly improve flow exploration in the context of climate research. Obviously, climate research is, like fluid dynamics, a very active and important field which might become crucial w.r.t. to current changes in climate. In this context, one typically has to cope with hard requirements to both the simulation and the visualization stage, i.e. high resolutions and the desire for real-time support. Graphics hardware is especially well-suited to realize an interactive visual analysis, and it could be enhanced to cope with the demanding requirements by providing support for even larger data sets and a complete set of tools for flow exploration. One of the next challenges in this context will be to convince potential users of such systems of the suitability and power of the graphics hardware.

Bibliography

[BFH*04] Buck I., Foley T., Horn D., Sugerman J., Fatahalian K., Houston M., Hanrahan P.: Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics 23*, 3 (2004), 777–786.

[Buc05] Buck I.: Taking the plunge into GPU computing. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, 2005, ch. 32, pp. 509–519.

[CFL28] Courant R., Friedrichs K., Lewy H.: Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen 100*, 1 (1928), 32–74.

[CHH02] Carr N. A., Hall J. D., Hart J. C.: The ray engine. In *Proc. Graphics Hardware* (2002), pp. 37–46.

[CK07] Cuntz N., Kolb A.: Fast hierarchical 3D distance transforms on the GPU. In *Proc. Eurographics (Short Paper)* (2007), pp. 93–96.

[CKL*07] Cuntz N., Kolb A., Leidl M., Rezk-Salama C., Böttinger M.: GPU-based dynamic flow visualization for climate research applications. In *Proc. SimVis* (2007), pp. 371–384.

[CKSW08] Cuntz N., Kolb A., Strzodka R., Weiskopf D.: Particle level set advection for the interactive visualization of unsteady 3D flow. *Computer Graphics Forum (EuroVis) 27*, 3 (2008), 719–726.

[CL93] Cabral B., Leedom L. C.: Imaging vector fields using line integral convolution. In *Proc. ACM SIGGRAPH* (1993), pp. 263–270.

[CM99] Cuisenaire O., Macq B.: Fast Euclidean distance transformation by propagation using multiple neighborhoods. *Computer Vision and Image Understanding 76*, 2 (1999), 163–172.

[CP92] Cartwright J. H. E., Piro O.: The dynamics of Runge-Kutta methods. *Int. Journal of Bifurcation and Chaos 2*, 3 (1992), 427–449.

[CPK09] Cuntz N., Pritzkau A., Kolb A.: Time-adaptive lines for the interactive visualization of unsteady flow data sets. *Computer Graphics Forum (accepted for publication)* (2009).

[Cui99] Cuisenaire O.: *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing.* PhD thesis, UCL, Louvain-la-Neuve, Belgium, 1999.

[CV00] Chan T. F., Vese L. A.: *Image Segmentation using Level Sets and the Piecewise-constant Mumford-Shah Model.* Tech. rep., Computational Applied Math Group, 2000.

[DC96] Desbrun M., Cani M.-P.: Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proc. Computer Animation and Simulation* (1996), pp. 61–76.

[Die92] Dietz H. G.: Common subexpression induction. In *Proc. Int. Conf. on Parallel Processing* (1992), pp. 174–182.

[DK93] Dietz H. G., Krishnamurthy G.: Meta-state conversion. In *Proc. Int. Conf. on Parallel Processing* (1993), pp. 47–56.

[DNB*05] Duca N., Niski K., Bilodeau J., Bolitho M., Chen Y., Cohen J.: A relational debugging engine for the graphics pipeline. *ACM Trans. on Graphics 24*, 3 (2005), 453–463.

[EFFM02] Enright D., Fedkiw R., Ferziger J., Mitchell I.: A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics 183*, 1 (2002), 83–116.

[EHK*06] Engel K., Hadwiger M., Kniss J., Lefohn A., Rezk Salama C., Weiskopf D.: *Real-time Volume Graphics.* AK Peters Ltd., 2006.

[ELF05] Enright D., Losasso F., Fedkiw R.: A fast and accurate semi-Lagrangian particle level set method. *Computers & Structures 83*, 6-7 (2005), 479–490.

[EMF02] Enright D., Marschner S., Fedkiw R.: Animation and rendering of complex water surfaces. *ACM Trans. on Graphics 21*, 3 (2002), 736–744.

[EMP09] Eilemann S., Makhinya M., Pajarola R.: Equalizer: A scalable parallel rendering framework. *IEEE Trans. on Visualization and Computer Graphics 15*, 3 (2009), 436–452.

[FM96] Foster N., Metaxas D.: Realistic animation of liquids. *Graphical Models and Image Processing 58*, 5 (1996), 471–483.

[FMST96] Fujishiro I., Maeda Y., Sato H., Takeshima Y.: Volumetric data exploration using interval volume. *IEEE Trans. on Visualization and Computer Graphics 2*, 2 (1996), 144–155.

[FSJ01] Fedkiw R., Stam J., Jensen H. W.: Visual simulation of smoke. In *Proc. ACM SIGGRAPH* (2001), pp. 23–30.

[GM77] Gingold R., Monaghan J.: Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society 181* (1977), 375–389.

[Gre08] Green S.: *CUDA Particles.* Tech. rep., NVidia CUDA SDK, presented at Game Developers Conference, 2008.

[GRNG05] Griesser A., Roeck S. D., Neubeck A., Gool L. V.: GPU-based foreground-background segmentation using an extended colinearity criterion. In *Proc. Vision, Modeling and Visualization* (2005), pp. 319–326.

[GTS*04] Garth C., Tricoche X., Salzbrunn T., Bobach T., Scheuermann G.: Surface techniques for vortex visualization. In *Proc. EG/IEEE TCVG Symp. on Visualization* (2004), pp. 155–164.

[Har03] Harris M. J.: *Real-Time Cloud Simulation and Rendering.* PhD thesis, CS Dep., University of N. Carolina at Chapel Hill, 2003.

[Har05] Harris M. J.: Mapping computational concepts to GPUs. In *GPU Gems 2*. Addison Wesley, 2005, ch. 31, pp. 493–508.

[Har07] Harada T.: Real-time rigid body simulation on GPUs. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, 2007, ch. 29, pp. 611–632.

[HBISL03] Harris M. J., Baxter III W., Scheuermann T., Lastra A.: Simulation of cloud dynamics on graphics hardware. In *Proc. Graphics Hardware* (2003), pp. 92–101.

[HBSB02] Hibbard B., Böttinger M., Schultz M., Biercamp J.: Visualization in earth system science. *ACM SIGGRAPH Computer Graphics 36*, 4 (2002), 5–9.

[HCM06] Hegeman K., Carr N., Miller G.: Particle-based fluid simulation on the GPU. In *Proc. Int. Conf. on Computational Science* (2006), pp. 228–235.

[HCSL02] Harris M. J., Coombe G., Scheuermann T., Lastra A.: Physically-based visual simulation on graphics hardware. In *Proc. Graphics Hardware* (2002), pp. 109–118.

[HH89] Hämmerlin G., Hoffmann K.-H.: *Numerische Mathematik.* Springer Verlag, 1989.

[HHN*02] Humphreys G., Houston M., Ng R., Frank R., Ahern S., Kirchner P. D., Klosowski J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. on Graphics 21*, 3 (2002), 693–702.

[HKK07a] Harada T., Koshizuka S., Kawaguchi Y.: Sliced data structure for particle-based simulations on GPUs. In *Proc. Int. Conf. on Computer Graphics and Interactive Techniques* (2007), pp. 55–62.

[HKK07b] Harada T., Koshizuka S., Kawaguchi Y.: Smoothed particle hydrodynamics on GPUs. In *Proc. Computer Graphics International* (2007).

[HKL*99] Hoff K., Keyser J., Lin M., Manocha D., Culver T.: Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH* (1999), pp. 277–286.

[HS90] Hibbard B., Santek D.: The vis-5d system for easy interactive visualization. In *Proc. IEEE Conf. on Visualization* (1990), pp. 28–35.

[HSO07] Harris M., Sengupta S., Owens J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, 2007, ch. 39, pp. 851–875.

[HSS*05] Hadwiger M., Sigg C., Scharsach H., Bühler K., Gross M.: Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proc. Eurographics* (2005), vol. 24, pp. 303–312.

[Hul92] Hultquist J. P. M.: Constructing stream surfaces in steady 3D vector fields. In *Proc. IEEE Conf. on Visualization* (1992), pp. 171–178.

[JEH01] Jobard B., Erlebacher G., Hussaini M. Y.: Lagrangian-Eulerian advection for unsteady flow visualization. In *Proc. IEEE Conf. on Visualization* (2001), pp. 53–60.

[JEH02] Jobard B., Erlebacher G., Hussaini M. Y.: Lagrangian-Eulerian advection of noise and dye textures for unsteady flow visualization. *IEEE Trans. on Visualization and Computer Graphics 8*, 3 (2002), 211–222.

[KC05] Kolb A., Cuntz N.: Dynamic particle coupling for GPU-based fluid simulation. In *Proc. Symp. on Simulation Technique* (2005), pp. 722–727.

[Ker90] Kerlick G. D.: Moving iconic objects in scientific visualization. In *Proc. IEEE Conf. on Visualization* (1990), pp. 124–130.

[KKKW05] Krüger J., Kipfer P., Kondratieva P., Westermann R.: A particle system for interactive visualization of 3D flows. *IEEE Trans. on Visualization and Computer Graphics 11*, 6 (2005).

[KLRS04] Kolb A., Latta L., Rezk-Salama C.: Hardware-based simulation and collision detection for large particle systems. In *Proc. Graphics Hardware* (2004), ACM/Eurographics, pp. 123–131.

[KSW04] Kipfer P., Segal M., Westermann R.: Uberflow: A GPU-based particle engine. In *Proc. Graphics Hardware* (2004), ACM/Eurographics, pp. 115–122.

[KW05] Krüger J., Westermann R.: GPU simulation and rendering of volumetric effects for computer games and virtual environments. In *Proc. Eurographics* (2005), pp. 685–693.

[Lar04] Laramee R. S.: *Interactive 3D Flow Visualization Based on Textures and Geometric Primitives.* PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2004.

[LHD*04] Laramee R. S., Hauser H., Doleisch H., Vrolijk B., Post F. H., Weiskopf D.: The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum 23*, 2 (2004), 203–222.

[LKHW04] Lefohn A., Kniss J., Hansen C., Whitaker R.: A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces. *IEEE Trans. on Visualization and Computer Graphics 10*, 4 (2004), 422–433.

[LKS*06] Lefohn A., Kniss J. M., Strzodka R., Sengupta S., Owens J. D.: Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. on Graphics 25*, 1 (2006), 60–99.

[LMG06] Liu Z., Moorhead R. J., Groner J.: An advanced evenly-spaced streamline placement algorithm. *IEEE Trans. on Visualization and Computer Graphics 12*, 5 (2006), 965–972.

[MB95] Max N., Becker B.: Flow visualization using moving textures. In *Proc. Symp. on Visualizing Time-Varying Data* (1995), pp. 77–87.

[MBC93] Max N., Becker B., Crawfis R.: Flow volumes for interactive vector field visualization. In *Proc. IEEE Conf. on Visualization* (1993), pp. 25–29.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. In *Proc. ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2003), pp. 154–159.

[MDHZ08] Mei X., Decaudin P., Hu B., Zhang X.: Real-time marker level set on GPU. In *Proc. Int. Conf. on Cyberworlds* (2008), pp. 209–2016.

[MDTP*04] McCool M., Du Toit S., Popa T., Chan B., Moule K.: Shader algebra. In *Proc. ACM SIGGRAPH* (2004), pp. 787–795.

[MF06] Mokberi E., Faloutsos P.: *A Particle Level Set Library.* Tech. rep., UCLA Computer Graphics Lab, 2006.

[MGAK03] Mark W., Glanville R., Akeley K., Kilgard M.: Cg: A system for programming graphics hardware in a c-like language. In *Proc. ACM SIGGRAPH* (2003), vol. 22, pp. 896–907.

[MIA*04] McCormick P. S., Inman J., Ahrens J. P., Hansen C., Roth G.: Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proc. IEEE Conf. on Visualization* (2004), pp. 171–178.

[Moo65] Moore G. E.: Cramming more components onto integrated circuits. *Electronics 38*, 8 (1965).

[NVi07] NVidia: *CUDA Compute Unified Device Architecture, Programming Guide.* Tech. rep., 2007.

[OF02] Osher S., Fedkiw R.: *Level Set Methods and Dynamic Implicit Surfaces.* Springer Verlag, 2002.

[OLG*05] Owens J., Luebke D., Govindaraju N., Harris M., Krueger J., Lefohn A., Purcell T.: A survey of general-purpose computation on graphics hardware. In *Proc. Eurographics (State of the Art Report)* (2005), pp. 21–51.

[OLG*07] Owens J., Luebke D., Govindaraju N., Harris M., Krueger J., Lefohn A., Purcell T.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[OS88] Osher S., Sethian J.: Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics 79* (1988), 12–49.

[PBL*04] Park S. W., Budge B., Linsen L., Hamann B., Joy K. I.: Multi-dimensional transfer functions for interactive 3D flow visualization. In *Proc. Pacific Graphics* (2004), pp. 1–8.

[PBL*05] Park S. W., Budge B., Linsen L., Hamann B., Joy K. I.: Dense geometric flow visualization. In *Proc. EG/IEEE TCVG Symp. on Visualization* (2005), pp. 177–185.

[PBMH02] Purcell T., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. In *Proc. ACM SIGGRAPH* (2002), vol. 21, pp. 703–712.

[PBP02] Prautzsch H., Boehm W., Paluszny M.: *Bézier and B-Spline Techniques.* Springer Verlag, 2002.

[PR02] Preusser T., Rumpf M.: A level set method for anisotropic geometric diffusion in 3d image processing. *SIAM Journal on Applied Mathematics 62* (2002), 1772–1793.

[PVH*02] Post F. H., Vrolijk B., Hauser H., Laramee R., Doleisch H.: Feature extraction and visualisation of flow fields. In *Proc. Eurographics (State of the Art Report)* (2002), pp. 69–100.

[PVH*03] Post F. H., Vrolijk B., Hauser H., Laramee R. S., Doleisch H.: The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum 22*, 4 (2003), 775–792.

[RBB*03] Roeckner E., Baeuml G., Bonaventura L., Brokopf R., Esch M., Giorgetta M., Hagemann S., Kirchner I., Kornblueh L., Manzini E., Rhodin A., Schlese U., Schulzweidea U., Tompkins A.: *The Atmospheric General Circulation Model ECHAM5, Part I.* Tech. rep., Max Planck Institute for Meteorology, 2003.

[Ree83] Reeves W.: Particle systems - technique for modeling a class of fuzzy objects. In *Proc. ACM SIGGRAPH* (1983), vol. 2, pp. 91–108.

[Rey87] Reynolds C. W.: Flocks, herds and schools: A distributed behavioral model. In *Proc. ACM SIGGRAPH* (1987), pp. 25–34.

[RS01] Rumpf M., Strzodka R.: Level set segmentation in graphics hardware. In *Proc. of IEEE International Conf. on Image Processing* (2001), pp. 1103–1106.

[RS02] Rezk-Salama C.: *Volume Rendering Techniques for General Purpose Graphics Hardware.* PhD thesis, Technische Fakultät der Universität Erlangen-Nürnberg, 2002.

[RT06] Rong G., Tan T.-S.: Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proc. ACM Symp. on Interactive 3D Graphics and Games* (2006), pp. 109–116.

[SA08] Segal M., Akeley K.: *The OpenGL Graphics System: A Specification.* Tech. rep., 2008.

[SB00] Stoer J., Bulirsch R.: *Numerische Mathematik 2*, 4. ed. Springer Verlag, 2000.

[SC91] Staniforth A., Cot J.: Semi-Lagrangian integration schemes for atmospheric models – a review. *Monthly Weather Review of the American Meteorological Society 119* (1991), 2206–2223.

[SDK05] Strzodka R., Doggett M., Kolb A.: Scientific computation for simulations on programmable graphics hardware. *Simulation Practice & Theory 13*, 8 (2005), 667–680.

[Set96] Sethian J. A.: A fast marching level set method for monotonically advancing fronts. In *Proc. National Academy of Sciences* (1996), pp. 1591–1595.

[SGGM06] SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3D distance field computation using linear factorization. In *Proc. ACM Symp. on Interactive 3D Graphics and Games* (2006), pp. 117–124.

[SGS05] STOLL C., GUMHOLD S., SEIDEL H.-P.: Visualization with stylized line primitives. In *Proc. IEEE Conf. on Visualization* (2005), pp. 659–702.

[Sim90] SIMS K.: Particle animation and rendering using data parallel computation. In *Proc. ACM SIGGRAPH* (1990), pp. 405–413.

[SJEG05] SVAKHINE N. A., JANG Y., EBERT D., GAITHER K.: Illustration and photography inspired visualization of flows and volumes. In *Proc. IEEE Conf. on Visualization* (2005), pp. 687–694.

[SJM96] SHEN H.-W., JOHNSON C. R., MA K.-L.: Visualizing vector fields using line integral convolution and dye advection. In *Proc. Symp. Volume Visualization* (1996), pp. 63–70.

[SK98] SHEN H.-W., KAO D. L.: A new line integral convolution algorithm for visualizing time-varying flow fields. *IEEE Trans. on Visualization and Computer Graphics 4*, 2 (1998), 98–108.

[SMA00] SANNA A., MONTRUCCHIO B., ARINAZ R.: Visualizing unsteady flows by adaptive streaklines. In *Proc. Int. Conf. on Computer Graphics, Visualization and Interactive Digital Media* (2000).

[Söd02] SÖDERLIND G.: Automatic control and adaptive time-stepping. *Numerical Algorithms 31*, 1–4 (2002), 281–310.

[SPG03] SIGG C., PEIKERT R., GROSS M.: Signed distance transform using graphics hardware. In *Proc. IEEE Conf. on Visualization* (2003), pp. 12–19.

[ST04] STRZODKA R., TELEA A.: Generalized distance transforms and skeletons in graphics hardware. In *Proc. EG/IEEE TCVG Symp. on Visualization* (2004), pp. 221–230.

[Sta99] STAM J.: Stable fluids. In *Proc. ACM SIGGRAPH* (1999), pp. 121–128.

[Stö95] STÖCKER H.: *Taschenbuch mathematischer Formeln und moderner Verfahren*. Verlag Harri Deutsch, 1995, ch. 18.12 Numerische Integration von Differentialgleichungen, pp. 635–ff.

[Str02] STRZODKA R.: Virtual 16 bit precise operations on RGBA8 textures. In *Proc. Vision, Modeling and Visualization* (2002), pp. 171–178.

[STWE07] SCHAFHITZEL T., TEJADA E., WEISKOPF D., ERTL T.: Point-based stream surfaces and path surfaces. In *Proc. Graphics Interface* (2007), pp. 289–296.

[TGE97] TEITZEL C., GROSSO R., ERTL T.: Efficient and reliable integration methods for particle tracing in unsteady flows on discrete meshes. In *Proc. Eurographics Workshop on Visualization in Scientific Computing* (1997), Lefer W., Grave M., (Eds.), pp. 49–56.

[TS00] TRENDALL C., STEWART A. J.: General calculations using graphics hardware, with application to interactive caustics. In *Proc. Eurographics Workshop on Rendering* (2000), pp. 287–298.

[TvW03] TELEA A., VAN WIJK J. J.: 3D IBFV: Hardware-accelerated 3D flow visualization. In *Proc. IEEE Conf. on Visualization* (2003), pp. 233–240.

[vFTS06] VON FUNCK W., THEISEL H., SEIDEL H.-P.: Vector field based shape deformations. *ACM Trans. on Graphics 25*, 3 (2006), 1118–1125.

[VKG05] VIOLA I., KANITSAR A., GRÖLLER E.: Importance-driven feature enhancement in volume visualization. In *Proc. IEEE Conf. on Visualization* (2005), pp. 408–418.

[VMS07] VIOREL M., METAXAS D., SUSSMAN M.: Textured liquids based on the marker level set. *Computer Graphics Forum 26*, 3 (2007), 457–466.

[vW93] VAN WIJK J.: Implicit stream surfaces. In *Proc. IEEE Conf. on Visualization* (1993), pp. 245–252.

[vW02] VAN WIJK J. J.: Image based flow visualization. *ACM Trans. on Graphics 21*, 3 (2002), 745–754.

[Wei04] WEISKOPF D.: Dye advection without the blur: A level-set approach for texture-based visualization of unsteady flow. *Computer Graphics Forum 23*, 3 (2004), 479–488.

[WJE00] WESTERMANN R., JOHNSON C., ERTL T.: A level-set method for flow visualization. In *Proc. IEEE Conf. on Visualization* (2000), pp. 147–154.

[WSE07] WEISKOPF D., SCHAFHITZEL T., ERTL T.: Texture-based visualization of unsteady 3D flow by real-time advection and volumetric illumination. *IEEE Trans. on Visualization and Computer Graphics 13*, 3 (2007), 569–582.

[WSEE05] WEISKOPF D., SCHRAMM F., ERLEBACHER G., ERTL T.: Particle and texture based spatiotemporal visualization of time-dependent vector fields. In *Proc. IEEE Conf. on Visualization* (2005), p. 81.

[WTS*07] WIEBEL A., TRICOCHE X., SCHNEIDER D., JAENICKE H., SCHEUERMANN G.: Generalized streak lines: Analysis and visualization of boundary induced vortices. *IEEE Trans. on Visualization and Computer Graphics 13*, 6 (2007), 1735–1742.

[XZC04] XUE D., ZHANG C., CRAWFIS R.: Rendering implicit flow volumes. In *Proc. IEEE Conf. on Visualization* (2004), pp. 99–106.

[YHK09] YASUDA R., HARADA T., KAWAGUCHI Y.: Fast rendering of particle-based fluid by utilizing simulation data. In *Proc. Eurographics (Short Paper)* (2009).

[ZSH96] ZÖCKLER M., STALLING D., HEGE H.-C.: Interactive visualization of 3d-vector fields using illuminated stream lines. In *Proc. IEEE Conf. on Visualization* (1996), pp. 107–ff.

[ZSP08] ZHANG Y., SOLENTHALER B., PAJAROLA R.: Adaptive sampling and rendering of fluids on the GPU. In *Proc. IEEE/EG Int. Symp. on Point-Based Graphics* (2008), pp. 137–146.

Index