**Institut für Informatik**
**TECHNISCHE UNIVERSITÄT MÜNCHEN**

# An Embedding Method for Interactive Simulation on Dynamic Surfaces

*Stefan Auer*

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

**Doktors der Naturwissenschaften (Dr. rer. nat.)**

genehmigten Dissertation.

| | |
|---|---|
| Vorsitzende: | Univ.-Prof. Gudrun J. Klinker, Ph.D. |
| Prüfer der Dissertation: | 1. Univ.-Prof. Dr. Rüdiger Westermann |
| | 2. Univ.-Prof. Dr. Jens Krüger |
| | Universität Duisburg-Essen |

Die Dissertation wurde am 02.04.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.05.2013 angenommen.

*To my family and friends*

# Abstract

Numerical simulation on curved surfaces enables the dynamic texturing of three-dimensional objects. In this thesis, I introduce methods for the real-time simulation and visualization of intrinsic fluid dynamics on deforming surfaces. These novel techniques support arbitrary, including open or non-orientable, surfaces and are universally applicable to a wide range of partial differential equation (PDE) problems involving differential operators up to the second order. The applications within this work focus particularly on interactive visual effects, however. All presented techniques utilize optimized parallel algorithms to employ the computing power and memory bandwidth of current GPUs in the best possible way.

First, I show how the closest point method (CPM) can be applied to the real-time simulation of fluid effects on static surfaces. The CPM is an already existing embedding method that solves completely standard PDEs in the embedding space, using common numerical methods on a uniform Cartesian grid. A novel CUDA method enables the efficient conversation of a triangulated surface into the implicit representation required by the CPM. This implicit surface representation is stored together with the simulation attributes in a GPU-friendly multiblock grid, which facilitates the efficient solution of surface PDEs at high resolution. Yet, at the same time, this data structure is also very suitable for raycasting, which enables the visualization of simulation results as dynamic surface displacements. I demonstrate the capabilities of these techniques by simulating the propagation of fluid-like surface deformations, using either the wave equation or the incompressible Navier-Stokes equations.

Second, I further develop the presented principles to a semi-Lagrangian closest point method (SLCPM) for the real-time simulation on animated surfaces. The method integrates the semi-Lagrangian scheme directly into the explicit closest point method and rigorously exploits the synergies. The SLCPM

can be combined with a wide range of animation techniques for triangulated surfaces. It is unconditionally stable with respect to deformations of the surface and its precision does not depend on the input geometry. To the best of my knowledge, the SLCPM is the first Eulerian method for interactive fluid dynamics on deforming surfaces.

Third, I present a novel contouring algorithm, which reconstructs a high quality triangle mesh from an implicit surface representation based on closest points. It selects the vertex positions directly from the set of closest points. Hence all triangle vertices are guaranteed to lie exactly on the zero-contour and no approximations are necessary. Since the vertex selection is guided by a CPM-based Laplacian analysis of the surface, the resulting contour preserves sharp features and small-scale details. A novel table-based triangulation scheme avoids small or degenerated triangles in smooth areas and enables the contouring of open and non-orientable surfaces. The contouring algorithm is fully integrated into the interactive SLCPM pipeline in order to enable future applications in the simulation of dynamic surface deformations.

# Zusammenfassung

Numerische Simulation auf gekrümmten Flächen ermöglicht die dynamische Texturierung dreidimensionaler Objekte. In dieser Arbeit stelle ich Methoden zur Echtzeitsimulation und Visualisierung von Strömungsdynamik auf sich verformenden Flächen vor. Diese neuartigen Methoden unterstützen beliebige, d.h. auch nicht geschlossene und nicht orientierbare, Flächen und sind universell einsetzbar für eine Vielzahl von partiellen Differentialgleichungs-Problemen (PDE) welche Differentialoperatoren bis zur zweiten Ordnung beinhalten. Die Anwendungen in dieser Arbeit konzentrieren sich jedoch insbesondere auf interaktive visuelle Effekte. Alle vorgestellten Techniken verwenden optimierte parallele Algorithmen um die Rechenleistung und Speicherbandbreite aktueller GPUs in der bestmöglichen Art und Weise einzusetzen.

Erstens zeige ich wie die Closest-Point-Methode (CPM) zur Echtzeitsimulation von Fluideffekten auf statischen Flächen angewandt werden kann. Die CPM ist eine bereits existierende Einbettungsmethode, welche vollkommen übliche PDEs im Einbettungsraum löst und dabei numerische Standardmethoden auf kartesischen Gittern einsetzt. Eine neuartige CUDA-Methode ermöglicht die effiziente Konvertierung einer triangulierten Fläche in die implizite Repräsentation welche von der CPM benötigt wird. Diese implizite Flächenrepräsentation wird zusammen mit den Simulationsattributen in einem GPU-freundlichen Multiblock Gitter gespeichert, welches die effiziente Lösung von Flächen-PDEs bei hoher Auflösung ermöglicht. Zudem ist diese Datenstruktur jedoch auch sehr gut für die Strahltraversierung geeignet, was die Visualisierung der Simulationsergebnisse als dynamischen Oberflächenversatz ermöglicht. Ich demonstriere das Leistungsvermögen dieser Techniken durch die Simulation der Ausbreitung von fluidähnlichen Flächendeformationen, entweder unter Verwendung der Wellengleichung oder der inkompressiblen Navier-Stokes Gleichungen.

Zweitens entwickle ich die vorgestellten Prinzipien weiter zu einer semi-Lagrange'schen Closest-Point-Methode (SLCPM) für die Echtzeitsimulation auf animierten Oberflächen. Die Methode integriert das semi-Lagrange'sche Schema direkt in die Closest-Point-Methode und nutzt die Synergien rigoros aus. Die SLCPM kann mit einer Vielzahl von Animationstechniken für triangulierte Flächen kombiniert werden. Sie ist bedingungslos stabil in Bezug auf Flächendeformationen und ihre Präzision hängt nicht von der Eingangsgeometrie ab. Nach meinem besten Wissen ist die SLCPM die erste Eulersche Methode für interaktive Strömungsdynamik auf sich verformenden Flächen.

Drittens stelle ich einen neuartigen Konturierungsalgorithmus vor, zur Rekonstruktion von hochqualitativen Dreiecksgittern aus impliziten Flächenrepräsentationen basierend auf nächstgelegenen Punkten. Dieser wählt die Eckpunkte direkt aus der Menge der nächstgelegenen Punkte aus. Dadurch wird garantiert, dass die Eckpunkte aller Dreiecke exakt auf der Nulldurchgangskontur liegen und es sind keine Approximationen mehr nötig. Dadurch dass die Eckpunktauswahl durch eine CPM-basierte Analyse des Laplace-Operators geleitet wird, bleiben kleinere und scharfkantige Merkmale der Fläche in der resultierenden Kontur erhalten. Ein neuartiges, tabellenbasiertes Triangulationsschema vermeidet kleine oder degenerierte Dreiecke in glatten Bereichen und ermöglicht die Konturierung von nicht orientierbaren Flächen und Flächen mit Rand. Der Konturierungsalgorithmus ist vollständig in die interaktive SLCPM-Pipeline integriert um zukünftige Anwendungen in der Simulation von dynamischen Flächendeformationen zu ermöglichen.

# Acknowledgements

I gratefully acknowledge the support of all of the people who made this thesis possible. First and foremost, I want to thank my advisor, Prof. Dr. Rüdiger Westermann, for offering me the great possibility to pursue research in the field of computer graphics, and for providing an excellent academic education during my time at his chair. I am very grateful for his supervision, for his commitment to my work, and for the numerous discussions. I also want to thank the co-authors of my papers Roland Fraedrich, Colin Macdonald, Jens Schneider and Marc Treib for helping me to obtain these results by contributing their suggestions and ideas. Furthermore, I would like to thank my current and former colleagues, Kai Bürger, Shunting Cao, Matthäus Chajdas, Ismail Demir, Christian Dick, Florian Ferstl, Roland Fraedrich, Raymund Fülöp, Stefan Hertel, Hans-Georg Menz, Mihaela Mihai, Tobias Pfaffelmoser, Marc Rautenhaus, Florian Reichl, Jens Schneider, Marc Treib, Mikael Vaaraniemi, and Jun Wu, for many interesting discussions. I am enormously thankful to my wife, my parents, and my friends for giving me all the support that I needed. Finally, I want to thank the Munich Centre of Advanced Computing (MAC) of the Technische Universität München for funding my work by a doctoral scholarship.

# Contents

**Abstract** i

**Zusammenfassung** iii

**Acknowledgements** v

**1 Introduction** 1
- 1.1 Contributions . . . 2
- 1.2 Publications . . . 3
- 1.3 Structure of this Thesis . . . 3

**2 Fundamentals** 5
- 2.1 Computational Fluid Dynamics . . . 6
  - 2.1.1 Navier Stokes Equations . . . 9
  - 2.1.2 Wave Equation . . . 15
  - 2.1.3 Discretization Methods . . . 16
  - 2.1.4 Solvers for Large Systems of Linear Equations . . . 19
- 2.2 Stable Fluids . . . 22
  - 2.2.1 Chorin's Projection Scheme . . . 22
  - 2.2.2 Semi-Lagrangian Advection Scheme . . . 24
- 2.3 Partial Differential Equations on Surfaces . . . 25
  - 2.3.1 Mesh-Based Methods . . . 26
  - 2.3.2 Parametrization Methods . . . 26
  - 2.3.3 Embedding Methods . . . 27
- 2.4 The Closest Point Method . . . 28
  - 2.4.1 Equivalence of Gradients . . . 28
  - 2.4.2 Embedding Equations . . . 29
  - 2.4.3 The Explicit Closest Point Method . . . 30

2.4.4 Method of Lines . . . . . 31
2.4.5 Implicit Surface Representation . . . . . 33
2.4.6 Interpolation . . . . . 34
2.5 GPU Computing . . . . . 36
2.5.1 NVIDIA GPU Architecture . . . . . 37
2.5.2 Direct3D . . . . . 40
2.5.3 CUDA . . . . . 44

**3 Fluid Effects on Surfaces using the CPM 47**
3.1 Introduction . . . . . 47
3.2 Contribution . . . . . 49
3.3 Related Work . . . . . 50
3.4 Adaptive Multiblock Closest Point Method . . . . . 51
3.5 Closest Point Method on the GPU . . . . . 53
3.5.1 Data Layout and Grid Generation . . . . . 53
3.5.2 Closest Point Computation . . . . . 56
3.6 Fluid Simulation . . . . . 58
3.6.1 Initialization . . . . . 59
3.6.2 Evolving the PDE . . . . . 60
3.6.3 Closest Point Extension . . . . . 60
3.6.4 Wave equation . . . . . 63
3.6.5 Incompressible Navier–Stokes equations . . . . . 64
3.7 Rendering . . . . . 68
3.7.1 Volume Raycasting . . . . . 68
3.7.2 Rendering Surface Displacements . . . . . 72
3.8 Performance Analysis . . . . . 72
3.8.1 Surface Deformations . . . . . 75
3.9 Conclusion . . . . . 76

**4 A Semi-Lagrangian CPM for Deforming Surfaces 79**
4.1 Introduction . . . . . 79
4.2 Related Work . . . . . 81
4.3 A Semi-Lagrangian Closest Point Method . . . . . 83
4.3.1 Semi-Lagrangian Closest Point Extension . . . . . 85
4.3.2 Extended Closest Point Representation . . . . . 87
4.4 Results and Performance Analysis . . . . . 90
4.5 Conclusion and Further Work . . . . . 93

**5 Closest Point Contouring** **95**
5.1 Introduction . . . . . 95
5.2 Contribution . . . . . 97
5.3 Related Work . . . . . 98
5.4 The Closest Point Contouring Algorithm . . . . . 99
5.4.1 Triangulation-Aware Closest Point Selection . . . . . 100
5.4.2 Table-Based Closest Point Triangulation . . . . . 101
5.4.3 Feature-Sensitive Closest Point Selection . . . . . 104
5.5 Closest Point Contouring on the GPU . . . . . 107
5.6 Results and Performance Analysis . . . . . 108
5.7 Conclusion and Future Work . . . . . 111

**6 Conclusion and Future Work** **113**

**Bibliography** **115**

# 1

# Introduction

In computer graphics, we often try hard to create deceptively realistic images. Realism is only one of many important goals, however. Almost as much effort is put into the creation of clearly unnatural visual effects with a realistic look. For both purposes, we are concerned with the appearance of three-dimensional objects and with their motion. The appearance of non-translucent objects depends mainly on the properties of the surface materials. Hence techniques like texture mapping and texture synthesis were developed, to accurately store properties at the surface and to assist the creation of surface textures which cannot be created or captured by traditional methods. In a similar way, the motion of objects can be stored using a variety of animation techniques, and it can be generated using either capturing, creation by hand or using procedural synthesis. The procedural approach is especially important for the motion of liquids, which can seldom be modeled by hand or captured accurately. Today it is therefore common to employ computational fluid dynamics methods for the physically-based simulation of liquid motion.

This thesis concerns the creation of physically-based, but unnatural fluid motions along *arbitrary surfaces* in interactive real-time environments. Striking visual effects are generated, in spite of, or actually due to, the fact that no real-world fluid physics are modeled. The proposed techniques are not per se limited to unnatural behavior, however. Actually, they could also be applied to problems in fluid dynamics dealing, for example, with surface tension or capillary waves on fluid interfaces. The employed and presented methods for the solution of partial differential equations (PDEs) on surfaces, are also fundamentally useful in applications of other sciences, for example in the biological modeling of wound healing or the accretion of ice on airfoils.

The solution of PDEs posed on surfaces is generally a demanding problem, because of the complicated discretization of the intrinsic differential operators. In our situation, we additionally deal with real-time requirements, *deforming* surfaces and the high-quality rendering of the simulated fluid effects. We approach these challenges with optimized data structures and parallel algorithms, specifically designed for GPUs.

## 1.1 Contributions

**Real-time fluid simulation for visual effects.** Building upon recent work on the solution of partial differential equations on surfaces [RM08], we present a real-time closest point method for the simulation and rendering of fluid dynamics on surfaces. Our technique is based on the incompressible Navier-Stokes equations and the wave equation, posed on surfaces, and employs a uniform finite-difference discretization of the embedding three-dimensional space. We propose data structures and algorithms for the generation of computational grids adapting to complex surface shapes, that are specifically designed for the massively parallel architectures of GPUs. These computational grids not only permit the efficient parallel solution of surface PDEs, they also facilitate the high-quality simulation of surface displacements via raycasting. Our approach thus enables the interactive simulation of fluid effects on surfaces on commodity desktop hardware, at unprecedented level of detail.

**Simulation on deforming surfaces.** In order to solve PDEs on deforming surfaces, we integrate the semi-Lagrangian scheme into the closest point method. We combine this novel discretization technique with an efficient algorithm for generating implicit closet point surface representations. This enables, for the first time, the interactive fluid simulation on animated surfaces, based on completely static Cartesian grids. Employing a uniform Eulerian discretization, in particular sidesteps all problems with dynamic parametrizations, which plague Lagrangian methods.

**Contouring of implicit surface representations.** Separated from the application of fluid dynamics for visual effects, we additionally investigate the advantages of implicit surface representations based on closest points for contouring. While distance fields are omnipresent implicit surface representations in geometry processing and in numerical simulations, the fact that most distance computation algorithms could also output closest points has received

little attention. We present a table-based contouring algorithm for closest point representations, that is as efficient as the still very popular marching cubes algorithm, yet avoids the approximation of vertex positions and the generation of degenerated triangles. By employing our efficient simulation techniques for a Laplacian analysis of the surface, this novel contouring technique is also able to preserve small and sharp features.

## 1.2 Publications

Some of the research results presented in this thesis have been originally published in the following peer-reviewed conference papers and journal articles:

1. R. Fraedrich, S. Auer, and R. Westermann, *Efficient High-Quality Volume Rendering of SPH Data*, IEEE Transactions on Visualization and Computer Graphics 16(6): 1533-1540, 2010 (Proceedings of IEEE Visualization 2010).

2. S. Auer, C.B. Macdonald, M. Treib, J. Schneider, and R. Westermann, *Real-Time Fluid Effects on Surfaces using the Closest Point Method*, Computer Graphics Forum 31, 6 (2012), 1909-1923.

3. S. Auer and R. Westermann, *Direct Contouring of Implicit Closest Point Surfaces*, Eurographics 2013 Short Papers.

## 1.3 Structure of this Thesis

In the next chapter, we review the fundamentals of computational fluid dynamics, partial differential equations on surfaces, and GPU computing that are employed in this thesis. In Chapter 3, we present our fast GPU method for simulating and rendering intrinsic fluid dynamics on static surfaces, based on the closest point method. Our semi-Lagrangian closest point method for the Eulerian simulation on deforming surfaces is discussed in Chapter 4. In Chapter 5, we present our very efficient contouring method for implicit surface representations based on closest points. In Chapter 6, we summarize our results, and give an outlook on future work.

# 2

# Fundamentals

In this thesis, we discuss the physically motivated simulation of fluid phenomena on moving surfaces, based on the theory of computational fluid dynamics (CFD) combined with embedding methods for the solution of partial differential equations posed on surfaces (surface PDEs). We employ three-dimensional Cartesian embedding grids for the implicit representation of the surface and for the spatial discretization of the simulation domain, which enables the use of proven and well-understood finite difference methods.

This chapter summarizes the fundamentals of computational fluid dynamics (Section 2.1) and PDEs on surfaces (Section 2.3). A brief introduction is provided for the particular methods which are used in this thesis (Sections 2.2 and 2.4). Since all presented techniques in this work rely on the graphics processing unit (GPU) to achieve interactive simulation and visualization update rates, also a brief introduction to general purpose computing and rendering on recent NVIDIA GPU architectures (Section 2.5) is provided.

Parts of this chapter were created on the basis of introductory textbooks, course materials and technical documentations for CFD [And95, FP02, Kuz12], analysis [Cla11] and GPU computing [NVI12c, Mic12, NVI12b].

## 2.1 Computational Fluid Dynamics

The goal of computational fluid dynamics is to accurately predict fluid flows using numerical solution methods. Historically, CFD was primarily developed as a research and design tool for aerospace engineering, where it complemented the theoretical and experimental analysis. Soon, the technique found its way into other engineering disciplines and sciences, such as marine engineering, automotive engineering, environmental engineering, chemical engineering, biology, and meteorology. In the nineties, fluid simulation also became a topic in the computer graphics community, which adapted the CFD methods to create realistic fluid animations in movies and video games. Figure 2.1 shows some examples of modern CFD applications in different disciplines. To create an appropriate CFD method for a particular flow problem, one must wisely choose the ingredients of the numerical solution method.

The first of these ingredients is the mathematical model which describes the flow behavior using a set of (possibly non-linear) partial differential equations (PDEs) and boundary conditions. Each model makes some simplifying assumptions about the fluid flow, so that specialized solution techniques can focus on the relevant effects. This restriction on the more import flow properties is necessary, since a general purpose solution technique suitable for all flows does not exist yet. Sections 2.1.1 and 2.1.2 introduce the mathematical flow models that used in this thesis.

The second ingredient is the discretization method, which tells us how the (non-linear) model PDEs can be approximated by a system of (non-linear) algebraic equations for a discrete set of points in space and time. The most popular discretization methods in the CFD community are finite differences (FD), finite volumes (FV) and finite elements (FE). All three methods tesselate the spatial simulation domain using some sort of structured or unstructured grid. The model PDEs are approximated by a number of algebraic equations, which relate the dependent variables at a node or cell to other known or unknown values in the grid neighborhood.

Since the PDE model for unsteady flows is time-dependent, discretization is required not only for the spatial domain, but also for time. The time discretization is essentially decoupled from the spatial discretization. It can be realized with a different discretization method, and either before or after the spatial discretization. The most common case is a spatial semi-discretization

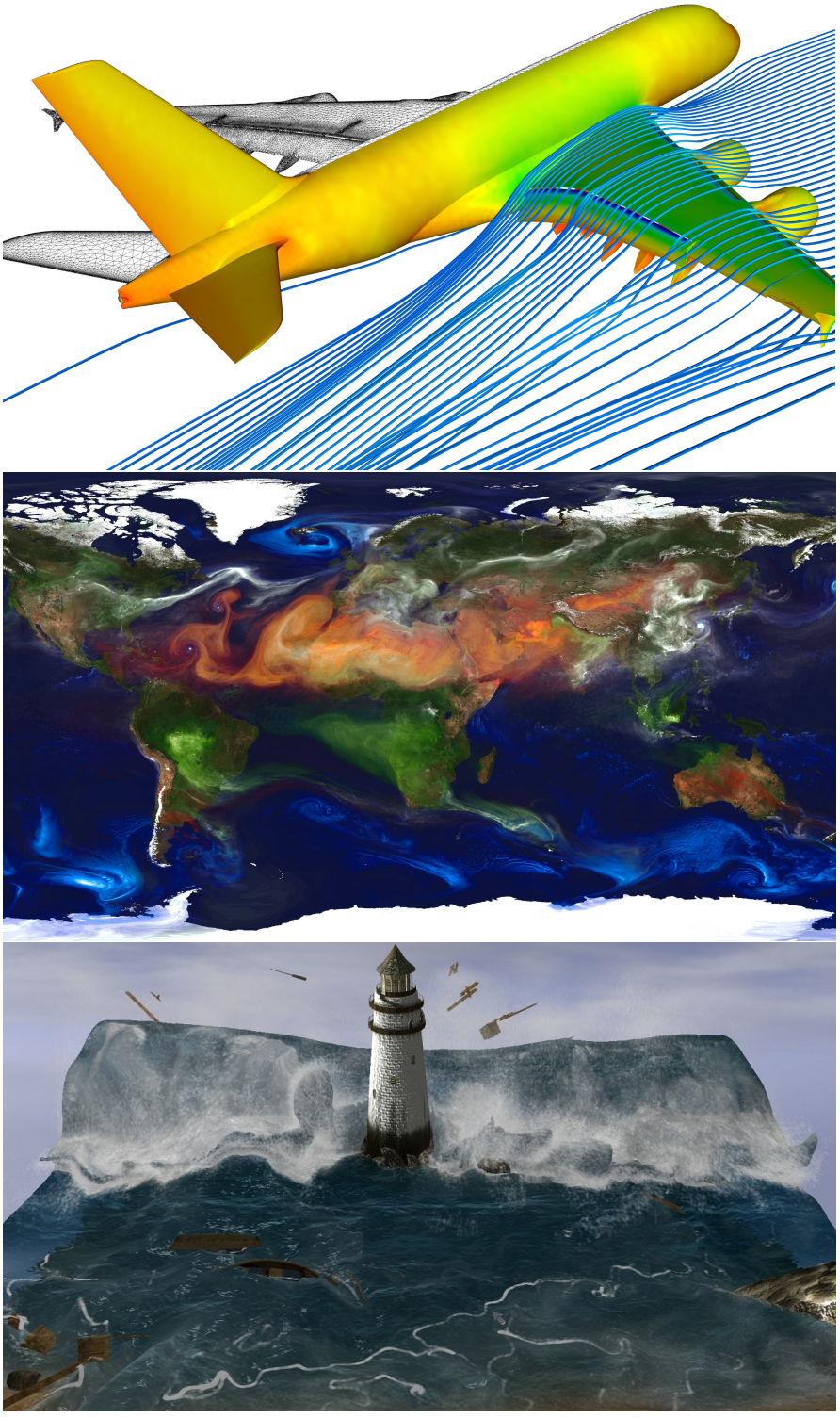

**Figure 2.1**: *Numerical flow simulations in engineering, meteorology and computer graphics. From top to bottom: Airbus A380 (Deutsches Zentrum für Luft- und Raumfahrt (DLR) news archive, licensed under Creative Commons CC-BY 3.0), Portrait of Global Aerosols (NASA Image of the Day Gallery, not copyrighted), Real-Time Demo 'Lighthouse' (Chentanez and Müller [CM11])*

(method of lines) with an arbitrary discretization method, followed by some time stepping scheme based on the finite difference method.

This independence does not imply, however, that time is irrelevant to the spatial discretization. In fact, an important differentiation comes from the question if sampling points are static or time-dependent. Spatial discretizations with fixed sampling positions are called Eulerian. The majority of the grid-based discretization methods in CFD belong to this class.

Lagrangian methods, on the other hand, discretize the spatial domain on a number of sampling points that move together with the fluid. These techniques usually deal with dynamic numerical stencils, i.e., different neighbors may appear in the algebraic equations of one sampling point in every time step. This is the reason why most Lagrangian methods are grid-free, at least in principle. Smoothed particle hydrodynamics (SPH), for example, requires spatial neighborhood queries to determine the numerical stencil for each particle. Many strengths and weaknesses of the Eulerian and the Lagrangian methods are complementary to each other, which is why hybrid methods have been developed to get the best from both approaches.

While Lagrangian and hybrid methods are currently very popular in the computer graphics community, they are less frequently found in the modern CFD literature. One reason for this may be the fact that the stability, convergence and accuracy of the Lagrangian methods are not yet understood as well as those of the Eulerian techniques. Section 2.1.3 gives a broader overview of the most common discretization methods.

The final ingredients in every CFD recipe are the numerical techniques to solve the large systems of (nonlinear) algebraic equations resulting from the discretization. For segregated discretizations, which treat the model PDEs and their separate terms individually from each other, only one independent variable appears in each resulting system of equations. Coupled discretizations, on the other hand, generate one very large system of equations, in which multiple independent variables appear. Successive linearization, e.g., using Newton's method, can be applied if the equations still contain nonlinear terms. In the end, most CFD methods arrive at one or more large but sparse linear systems of equations, which can be solved using a variety of iterative techniques. Section 2.1.4 presents some of the most common linear system solvers in CFD.

### 2.1.1 Navier Stokes Equations

The physical foundation of CFD comprises the fundamental governing equations of fluid dynamics, known today as the *complete Navier-Stokes equations*. The governing equations are mathematical statements of the conservation laws in physics:

- Continuity Equation: Conservation of Mass
- Momentum Equation: Newton's second law
- Energy Equation: First law of thermodynamics.

Historically, only the various forms of the momentum equation were called the Navier-Stokes equations. In this thesis, however, we follow the terminology in modern CFD literature and include the whole set of governing equations when referring to the full Navier-Stokes equations. The full Navier-Stokes equations are given in conservation form by the following system of PDEs:

**Continuity equation**

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \left( \rho \boldsymbol{v} \right) = 0 \tag{2.1}$$

**Momentum equation**

$$\frac{\partial \left( \rho \boldsymbol{v} \right)}{\partial t} + \nabla \cdot \left( \rho \boldsymbol{v} \otimes \boldsymbol{v} \right) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \boldsymbol{f} \tag{2.2}$$

**Internal energy equation**

$$\frac{\partial \left( \rho e \right)}{\partial t} + \nabla \cdot \left( \rho e \boldsymbol{v} \right) = \nabla \cdot (\kappa \nabla T) + \rho q - p \nabla \cdot \boldsymbol{v} + \nabla \boldsymbol{v} : \boldsymbol{\tau} \tag{2.3}$$

Independent variables

| | |
|---|---|
| $t$ | Time |
| $\boldsymbol{x}$ | Spatial coordinates vector |

Dependent variables

| | |
|---|---|
| $e(\boldsymbol{x},t)$ | Internal energy |
| $p(\boldsymbol{x},t)$ | Pressure |
| $\rho(\boldsymbol{x},t)$ | Density |
| $T(\boldsymbol{x},t)$ | Temperature |
| $\boldsymbol{\tau}(\boldsymbol{x},t)$ | Deviatoric stress tensor |
| $\boldsymbol{v}(\boldsymbol{x},t)$ | Velocity vector |

Constants

| | |
|---|---|
| $\boldsymbol{f}$ | Body forces vector (weight, electromagnetic forces, etc.) |
| $\kappa$ | Thermal conductivity |
| $q$ | Internal heat sources |

Here we should note that this thesis follows the ISO 80000 standard for formulas involving physical quantities. Hence, to distinguish scalars from vectors and higher-order tensors, the latter are written in boldface serif, boldface sans serif, respectively (i.e., $t$, $T$, $\tau$ are scalars, $\boldsymbol{t}$, $\boldsymbol{T}$, $\boldsymbol{\tau}$ are vectors, and $\mathbf{t}$, $\mathbf{T}$, $\boldsymbol{\tau}$ are higher-order tensors). Furthermore, the symbols "$\cdot$", ":", "$\otimes$", "$\times$" denote the inner product, double inner product, outer product respectively cross product between two tensors.

The PDE problem consisting of Equations 2.1, 2.2 and 2.3 is not well-posed. It misses initial and boundary conditions and the system is underdetermined, since only three equations are provided for six unknowns. Hence, besides appropriate initial and boundary conditions, also further assumptions about the behavior of the fluid flow are required to derive a complete PDE model for a particular CFD problem.

Two common assumptions are, for example, that the fluid is Newtonian and that it obeys the classical ideal gas law. In this case, three of the unknowns can be expressed in terms of the three other dependent variables, by the algebraic equations $\boldsymbol{\tau} = (\lambda \nabla \cdot \boldsymbol{v})\,\mathbf{I} + \mu\left(\nabla \boldsymbol{v} + \nabla \boldsymbol{v}^T\right)$, $p = \rho R T$ and $e = c_v T$, where $\mathbf{I}$ is the second-order identity tensor, $(\cdot)^T$ is the transpose, and $\lambda$, $\mu$, $R$ and $c_v$ are fluid specific constants. The closed PDE system resulting from these assumptions is often called *the compressible Navier-Stokes equations.*

In general, the necessary assumptions are answers to the questions:

- Is the fluid isotropic?
- Is the fluid Newtonian?
- Is the flow isothermal?
- Is the flow incompressible, compressible, transonic, supersonic, or hypersonic?
- Is the flow laminar or turbulent?
- Can viscosity be completely neglected?

According to these questions, flow problems are classified into distinct categories, which correspond to particular mathematical models of the flow. Some assumptions lead to significant simplifications in the resulting PDE model, which also affect the complexity of the required numerical methods.

For most classic CFD applications the primary goal is physical accuracy, measured by comparing the simulation results with observations from real world experiments. In aerodynamics, for example, the accuracy of a CFD simulation can often be verified in a wind tunnel experiment. For these applications, the nature of the investigated flow problem determines which mathematical and numerical models are required to achieve the desired accuracy.

The purpose of CFD simulations in computer graphics, on the other hand, is usually to generate impressive visual effects, which should look plausible, but which are not required to be physically accurate. Often less complex flow models and particularly efficient numerical solutions are chosen to maximize the resolution, as well as the spatial and temporal extents of the simulations.

For interactive simulations, like the ones discussed in this thesis, the typical combination of real-time requirements with limited computational resources results in tight efficiency restrictions for the numerical methods. These simulations are usually based on the computationally least expensive flow models. Desirable effects that would require complexer flow models are often integrated as rough approximations.

The PDE model that we use in this thesis describes the incompressible, isothermal flow of isotropic Newtonian fluids with constant viscosity:

**Continuity equation**

$$\nabla \cdot \boldsymbol{v} = 0 \tag{2.4}$$

**Momentum equation**

$$\rho \left( \frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} \right) = -\nabla p + \nu \nabla^2 \boldsymbol{v} + \boldsymbol{f} \tag{2.5}$$

Here $\nu = \mu/\rho$ represents the kinematic viscosity of the fluid. The density $\rho$ stays constant within incompressible flows and, therefore only the velocity $\boldsymbol{v}$ and the pressure $p$ remain as dependent variables. Consequentially the energy equation can be omitted.

Up to this point, all governing equations have been given in their conservation form, i.e., from a Eulerian point of view. The physical and mathematical meaning of Equation 2.5 becomes more apparent, however, if we look at its equivalent Lagrangian form:

**Momentum equation in nonconservation form**

$$\rho \frac{D \boldsymbol{v}}{D t} = -\nabla p + \nu \nabla^2 \boldsymbol{v} + \boldsymbol{f} \tag{2.6}$$

The term $\frac{D(\cdot)}{Dt}$ on the left hand side is called substantial derivative or material derivative. It describes the time rate of change of a physical quantity (in this case velocity), measured on a *moving* fluid at a *fixed* point in space. The substantial derivative is written in vector notation as

$$\frac{D(\cdot)}{Dt} = \frac{\partial(\cdot)}{\partial t} + \boldsymbol{v} \cdot \nabla(\cdot),$$

where $\frac{\partial(\cdot)}{\partial t}$ is the partial derivative with respect to time, $\boldsymbol{v}$ is the velocity of the fluid, and $\nabla(\cdot)$ is the gradient of a physical quantity. The term $\frac{\partial(\cdot)}{\partial t}$ is often called local derivative, while $\boldsymbol{v} \cdot \nabla(\cdot)$ is called convective derivative or advection operator. Now let us consider that in a Cartesian coordinate system with spatial coordinates $\boldsymbol{x} = (x, y, z)^T$ the gradient of a scalar field $f(\boldsymbol{x}, t) : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}$ has the form

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{pmatrix}.$$

Hence, the substantial derivative is written in Cartesian coordinates as

$$\frac{Df}{Dt} = \frac{\partial f}{\partial t} + v_x \frac{\partial f}{\partial x} + v_y \frac{\partial f}{\partial y} + v_z \frac{\partial f}{\partial z}, \tag{2.7}$$

with $v_x, v_y, v_z$ being the three components of the fluid's velocity $\boldsymbol{v}(\boldsymbol{x}, t) : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3$.

Since velocity is the time rate of change of position, the substantial derivative $\frac{D}{Dt}$ is essentially the same as the total derivative $\frac{d}{dt}$ (from calculus) with respect to time. This becomes apparent if we look at the total derivative of a function $h(x, y, z, t) : \mathbb{R}^4 \rightarrow \mathbb{R}$ with respect to $t$

$$\frac{dh}{dt} = \frac{\partial h}{\partial t} + \frac{\partial h}{\partial x}\frac{dx}{dt} + \frac{\partial h}{\partial y}\frac{dy}{dt} + \frac{\partial h}{\partial z}\frac{dz}{dt},$$

and consider that in Equation 2.7 $v_x = \frac{dx}{dt}, v_y = \frac{dy}{dt}$ and $v_z = \frac{dz}{dt}$. When we now also remember that density is mass per unit volume and that $\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}$, we can see that the momentum equation is only Newton's second law $\boldsymbol{F} = m\boldsymbol{a}$ in disguise. In Equation 2.6, the total force $\boldsymbol{F}$ per unit volume is represented by the sum of the pressure, viscosity and body forces on the right hand side, while the left hand side represents mass times acceleration.

To conclude this brief introduction to the Navier Stokes equations, we clarify the meaning of the other occurrences of the $\nabla$ operator in the governing equations and throughout this thesis. Depending on the notation, these differential terms represent either the gradient, divergence, curl, Laplacian or the convective derivative of a physical quantity. The definitions of the remaining operators in Cartesian coordinates are given below.

Laplacian of a scalar field

$$\nabla^2 f = \nabla \cdot (\nabla f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2}$$

Jacobian of a vector field $\boldsymbol{u}(\boldsymbol{x}, t) = \left(u_x, u_y, u_z\right)^T : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3$

$$\nabla \boldsymbol{u} = J(\boldsymbol{u}) = \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} & \frac{\partial u_z}{\partial x} \\ \frac{\partial u_x}{\partial y} & \frac{\partial u_y}{\partial y} & \frac{\partial u_z}{\partial y} \\ \frac{\partial u_x}{\partial z} & \frac{\partial u_y}{\partial z} & \frac{\partial u_z}{\partial z} \end{bmatrix}$$

Divergence of a vector field

$$\nabla \cdot \boldsymbol{u} = \operatorname{tr}(J(\boldsymbol{u})) = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$$

Curl of a vector field

$$\nabla \times \boldsymbol{u} = \begin{pmatrix} \frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z} \\ \frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x} \\ \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \end{pmatrix}$$

Laplacian of a vector field

$$\nabla^2 \boldsymbol{u} = \begin{pmatrix} \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2} + \frac{\partial^2 u_x}{\partial z^2} \\ \frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_y}{\partial z^2} \\ \frac{\partial^2 u_z}{\partial x^2} + \frac{\partial^2 u_z}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2} \end{pmatrix}$$

Convective derivative of a vector field

$$(\boldsymbol{v} \cdot \nabla)\boldsymbol{u} = \begin{pmatrix} v_x \frac{\partial u_x}{\partial x} + v_y \frac{\partial u_x}{\partial y} + v_z \frac{\partial u_x}{\partial z} \\ v_x \frac{\partial u_y}{\partial x} + v_y \frac{\partial u_y}{\partial y} + v_z \frac{\partial u_y}{\partial z} \\ v_x \frac{\partial u_z}{\partial x} + v_y \frac{\partial u_z}{\partial y} + v_z \frac{\partial u_z}{\partial z} \end{pmatrix}$$

Divergence of a tensor field $\boldsymbol{T}(\boldsymbol{x}, t) = \begin{bmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{bmatrix} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^{3\times 3}$:

$$\nabla \cdot \boldsymbol{T} = \begin{pmatrix} \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{zx}}{\partial z} \\ \frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y} + \frac{\partial T_{zy}}{\partial z} \\ \frac{\partial T_{xz}}{\partial x} + \frac{\partial T_{yz}}{\partial y} + \frac{\partial T_{zz}}{\partial z} \end{pmatrix}$$

### 2.1.2 Wave Equation

The classic wave equation is the prototype of a linear, second-order, hyperbolic PDE:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{2.8}$$

Independent variables
- $t$ Time
- $\boldsymbol{x}$ Spatial coordinates vector

Dependent variables
- $u$ Physical quantity disturbed by the wave

Constants
- $c$ Wave speed / phase velocity

In physics, this equation describes all waves that propagate with constant speed, such as vibrations in elastic strings, sound waves ($c$ = speed of sound) or electromagnetic waves ($c$ = speed of light). Gravity waves and capillary waves on the surface of water or other fluids are, most notably, not covered by the equation above, since their phase velocity generally depends on the wavelength and the water depth. In scientific computing, these kind of waves is usually treated with significantly simplified versions of the Navier-Stokes equations, such as, for example, the non-linear shallow water equations. In computer graphics, however, Equation 2.8 is often used for the simulation of ripples on fluid surfaces, since the influence of dispersion (i.e., the wave length dependence) and water depth is negligible under certain circumstances. The computer graphics literature also knows a number of advanced linear approximations of fluid gravity waves. The survey of Darles et al. [DCGG11] provides an excellent overview of the common techniques. The more advanced models have a number of advantages, but they are not entirely physically sound either, involve less simple differential operators and are computationally more expensive. In this work, we therefore choose the wave equation to serve as an alternative mathematical fluid model, which simplifies the introductions to our simulation and rendering techniques.

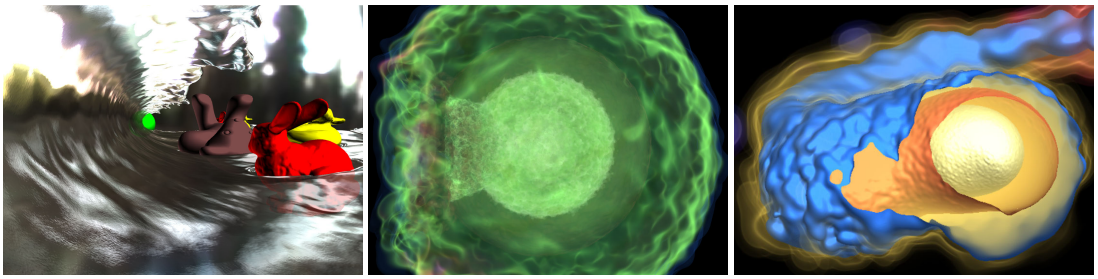

**Figure 2.2**: *High-quality volume rendering of SPH data with perspective resampling grids. Our technique supports iso-surface rendering (left), volume rendering (middle) and mixed modes (right).*

### 2.1.3 Discretization Methods

A vast amount of discretization methods has been applied to the numerical solution of fluid dynamic problems. In this overview, we name only the techniques which were most frequently used in the computer graphics community in the last years.

The *finite difference method* (FDM) starts with the differential form of a conservation equation. It covers the solution domain with a number of sampling points connected by edges. In principle, this grid can be of any type, but the most common case in CFD are structured grids with edges parallel to the local coordinate lines. At each grid vertex the partial differences with respect to the spatial coordinates are approximated using suitable difference quotients. These finite differences are usually based on Taylor series expansion or polynomial fitting, which makes it quite simple to find higher-order approximations. If required, solution values at points between grid vertices can be interpolated using the same techniques. The result of the finite difference approximation of a PDE is one algebraic equation per grid vertex where the unknowns are the values of the dependent variables at the vertices within a certain neighborhood in the grid. The FDM is especially effective for regular and Cartesian grids for which even higher-order schemes are relatively simple to implement and to analyze. Without special care, however, the FDM does not necessarily enforce conservation and it cannot handle complexer grid structures as straight-forward as other grid-based methods.

The starting point for the *finite volume method* (FVM) is the integral form of a conservation equation. The domain is subdivided into contiguous control volumes which are centered at the computational nodes where the solution is

being calculated. Values of the solution in the interior and on the surface of the control volumes are defined through interpolation. For each control volume the surface and volume integrals in the conservation equation are approximated using suitable quadrature rules based on the interpolation functions. The result of the finite volume approximation is one algebraic equation for each control volume where the unknowns are the center values of the neighboring control volumes. The FVM handles complex computational grids in a simple way, since the grid is not related to the coordinate system and only defines the boundaries of the control volumes. Conservation is ensured by construction, so long as the surface integral is the same for all control volumes sharing a boundary. Compared to the FDM it is harder to construct high order finite volume methods, however, since interpolation, differentiation and integration go hand in hand.

A typical *finite element method* (FEM) first transforms each conservation law from a boundary-value PDE problem into a variational problem. For this, the PDE is multiplied with an appropriate test function and integrated over the entire domain to obtain a functional which measures the quality (in the form of a weighted residual) of a solution function. To find an optimal numerical solution for this *weak form* of the original PDE problem, the solution domain is broken into a discrete set of finite elements. This step restricts the set of considered solution functions to a continuous, finite-dimensional subspace of the infinite-dimensional solution space. Each finite element represents one subset of the triangulated spatial domain (in 3D often a tetrahedron or hexahedron). Every degree of freedom of the solution space is described locally by a piecewise polynomial shape function and a nodal variable (often defined at the vertices of the triangulation). The result of the finite element approximation is one algebraic equation for each computational node where the unknowns are the neighboring nodal variables. The FEM is very flexible when it comes to grid generation, since the grids can be unstructured in general and since refinement is handled by simple subdivision. It is relatively easy to construct high-order approximations and to analyze the method mathematically, and its properties are optimal for certain types of equations. Compared to the FVM, stability is a smaller issue, but special care is required to guarantee conservation.

The smoothed particle hydrodynamics (SPH) method is the typical example for a mesh-free discretization which is based on the Lagrangian formula-

tion of the conservation laws. In the spatial domain, the fluid body is covered with an unstructured set of sampling points. Each sampling point is a pseudo-particle, which carries the properties of a small fluid volume and which moves according to the forces in the momentum equation. A continuous distribution of the fluid properties is realized using an interpolation based on overlapping radial smoothing kernels. The isotropic kernel functions have compact support and are centered at the particle positions. The result of the SPH approximation of a conservation law is one algebraic equation per particle, where the unknowns are the properties of the neighbor particles within the support radius. As a mesh-free method, SPH has advantages for sparse fluid distributions in large simulation domains and for free-surface flows with scattered splashes and drops. The Lagrangian discretization is mass-conserving by construction and it considerably simplifies the handling of the (nonlinear) convection terms. SPH is difficult to analyze numerically, however, and compared to grid-based discretizations it is less suited for high-order approximations. Constructing a computationally efficient SPH method can be a challenging task, since dynamic spatial neighborhood queries are required and since incompressibility can be ensured only with very small time steps in general. The dynamic, mesh-free discretization also poses problems for the high-quality visualization of the simulation results, which requires order-dependent integration of the fluid properties along the view rays. In Fraedrich et al. [FAW10] we have presented a method to accomplish this task very efficiently using a structured grid aligned with the view frustum. Figure 2.2 presents some results from this work.

The fluid implicit particle (FLIP) method is a simple variation of the particle in cell (PIC) method, which is a hybrid between grid-based Eulerian and particle-based Lagrangian techniques. FLIP starts with the differential form of the governing equations, but uses different discretizations for the convective and the non-convective terms. The simulation domain is covered with both, a static grid (usually regular and axis-aligned) and a number of freely movable particles. The grid is only an auxiliary structure, however, since all fluid properties are carried by the particles. In each time step an interpolation method is used to initialize the computational grid with the particle properties. Finite difference techniques are employed to calculate a solution for the next time step for all non-convective parts of the governing equations. At each grid vertex the differences in the grid properties between the current and

the next time step are calculated and another interpolation propagated these incremental updates back to the particle properties. Finally the convective terms of the governing equations are handled by advecting each particle using its updated velocity. FLIP can be recommended especially for large-scale free-surface flows. In this scenario it is often difficult to preserve the mass of small fluid features in purely Eulerian methods, while incompressibility is harder to guarantee with Lagrangian methods. Weak points of the standard FLIP method are its first-order accuracy and numerical dissipation resulting from inaccurate interpolation

### 2.1.4 Solvers for Large Systems of Linear Equations

Almost all discretizations of the Navier-Stokes equations require the solution of one or more large systems of linear equations of the form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2.9}$$

where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is a square matrix with components $a_{ij}$, $\boldsymbol{x} \in \mathbb{R}^N$ is the solution vector and $\boldsymbol{b} \in \mathbb{R}^N$ is a vector of constants. The choice of an appropriate solver depends on the problem size $N$ and the structure of the system matrix $\boldsymbol{A}$ (sparse, symmetric, n-diagonal, positive definite, etc.). For all grid-based discretization methods, including FDM, FVM and FEM, the structure of $\boldsymbol{A}$ is influenced by the structure of the discretization grid. An unstructured grid leads to a suboptimal properties of the system matrix, which in turn makes it harder to find an efficient solver.

In practical CFD applications $N$ is generally quite large since it is proportional to the number of computational nodes in the discretization, yet $\boldsymbol{A}$ is very sparse with typically less than ten non-zero values per row. This rules out most direct solvers, based on LU-, QR- or Cholesky-decomposition, due to their time and memory complexity for large problem sizes.

The most simple solvers which can be applied to relevant problem sizes belong tho the class of the stationary iterative methods. The *Jacobi* method, for example, starts with an arbitrary initial guess for the solution vector, e.g. $u_i^{(0)} = 0$, and in each iteration $k$ it refines the solution estimate of the last iter-

ation $k-1$ using the formula

$$x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)} \right), i = 1, ..., N.$$

Jacobi solvers are very easy to implement, have low memory requirements and they can be parallelized trivially. Convergence is guaranteed if $\boldsymbol{A}$ is strictly diagonally dominant, but the convergence process slows down considerably in later iterations where the residuum is dominated by low frequency errors. Other stationary iterative methods, like Gaus-Seidel and successive over-relaxation (SOR), are faster and guarantee convergence if $\boldsymbol{A}$ is symmetric positive definite. These methods cannot be parallelized so well, however, since they require an ordering of the inner iterations.

The more advanced non-stationary iterative solvers use not only the solution estimates from previous iterations, but also additional information on how these approximations were found. The (iterative, preconditioned) *conjugate gradient* (CG) method, for example, computes in each iteration $k$ a residual vector $\boldsymbol{r}^{(k)}$ and a search direction vector $\boldsymbol{p}^{(k)}$, which is used to incrementally update the solution estimate. The method ensures that the residuals $\boldsymbol{r}^{(k)}$ and $\boldsymbol{r}^{(k-1)}$ are orthogonal and that the search directions $\boldsymbol{p}^{(k)}$ and $\boldsymbol{p}^{(k-1)}$ are $\boldsymbol{A}$-conjugate, i.e., ${\boldsymbol{p}^{(k)}}^T \boldsymbol{A} \boldsymbol{p}^{(k-1)} = 0$. This corresponds to a very elegant solution of a minimization problem which is equivalent to Equation 2.9 if $\boldsymbol{A}$ is symmetric positive definite. The CG method achieves considerably better convergence rates compared to stationary methods, but this comes at the price of a more complicated implementation, especially when it comes to parallel architectures. CG is the best known non-stationary iterative solver, but there exist also several sophisticated alternatives like, for example, the minimum residual (MINRES), quasi-minimal residual (QMR) or bi-conjugate gradient stabilized (Bi-CGSTAB) methods. Barrett et. al. [BBC+94] provide an excellent overview and recommend particular techniques depending on the structure of the system matrix.

Multigrid methods, which also belong to the class of iterative solvers, are especially eligible for elliptic problems. This type of PDE is important for CFD methods which solve the Poisson equation in a pressure correction step. The multigrid development was triggered by the observation that point iterative solvers, like Gauss-Seidel, quickly smooth out high frequency modes of the

error field, but require lots of iterations for the low frequency errors. Geometric multigrid methods overcome this problem by replacing the discretization grid with a hierarchy of grids with different resolutions. Each multigrid V-cycle begins at the finest grid level where the high-frequency error is reduced with a few iterations of a standard (i.e., non-multigrid) iterative technique. A restriction operation prolongs the residual to the next coarser level, where the error frequencies appear higher with respect to the larger grid spacing. At each level, the high-frequency errors are reduced using with a number of iterative relaxation cycles before the residual is restricted to the next coarser level. At the coarsest level the procedure is inversed, by replacing the restriction operation with a prolongation operation to the next finer level. When the second phase arrives back at the finest level, the convergence is checked using the remaining residuum and another V-cycle is started if necessary. Geometric multigrid methods can be very effective for certain problems, but they are also entangled with the spatial discretization, since agglomeration of grid nodes, restriction and prolongation all take place on the geometric level. Especially for unstructured grids, it can be hard to find suitable definitions for these essential operations. Algebraic multigrid methods, on the other hand, perform all necessary operations on the matrix level, which makes them more independent from the discretization methods. Their disadvantage is, however, that the required matrix operations have a high computation and memory complexity and are hard to parallelize.

## 2.2 Stable Fluids

> *"Fluid solvers, for computer graphics, should ideally provide a user with a tool that enables her to achieve fluid-like effects in real-time. These factors are more important than strict physical accuracy, which would require too much computational power."*
>
> – Jos Stam, *Stable Fluids*

In this thesis we employ the *Stable Fluids* [Sta99] method, which is the most popular and well-known Eulerian fluid animation technique in computer graphics today. While countless improvements and alternatives where proposed since its introduction, the original algorithm still remains a good starting point, due to its remarkable effectivity and simplicity. This is especially true for more elaborate applications, like interactive fluid simulation on surfaces.

The mathematical model for the algorithm are the incompressible Navier-Stokes Equations 2.4 and 2.5 presented in Section 2.1.1. The discretization is a finite difference method based on Chorin's projection scheme (Section 2.2.1) and the semi-Lagrangian advection scheme (Section 2.2.2). Stable fluids is a fractional step method, which uses differential operator splitting to treat each term of the momentum equation individually and with very different integration methods. The particular integration methods which we employ in this thesis and their adaption to surface PDEs are discussed in Chapter 3.

### 2.2.1 Chorin's Projection Scheme

Chorin's projection scheme [CM00] is a splitting method for the incompressible Navier-Stokes equations, which we reproduce here in a condensed variant for convenience:

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = -\nabla p + \nabla^2 \boldsymbol{v}$$

$$\nabla \cdot \boldsymbol{v} = 0$$

This is a coupled system of PDEs for the velocity $\boldsymbol{v}$ and the pressure $p$. The idea is now to decouple $\boldsymbol{v}$ and $p$ and to separate the convection-diffusion equation from the incompressibility condition using a fractional step method of the form $\boldsymbol{v}^{(t-1)} \rightarrow \boldsymbol{v}^{(t-1/2)} \rightarrow \left(\boldsymbol{v}^{(t)}, p^{(t)}\right)$. In the first substep, $\boldsymbol{v}^{(t-1/2)}$ is determined as an approximation of the viscous Burgers equation, which remains when the

pressure gradient and the continuity equation are disregarded:

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = \nabla^2 \boldsymbol{v}. \tag{2.10}$$

Note, that Stam's method approximates also the equation above with a fractional step method, to obtain an inviscid Burgers equation and a diffusion equation. This second splitting is not a part of the projection scheme, however. For fluids with low viscosity the diffusion equation $\partial \boldsymbol{v} / \partial t = \nabla^2 \boldsymbol{v}$ is often omitted entirely, since numerical dissipation already introduces some artificial viscosity (see, for example, [Bri08] Section 1.5).

Using backward differences in time, the remaining parts of the original equation system are approximated in the second substep as

$$\frac{\boldsymbol{v}^{(t)} - \boldsymbol{v}^{(t-1/2)}}{\Delta t} = -\nabla p^{(t)} \tag{2.11}$$

and

$$\nabla \cdot \boldsymbol{v}^{(t)} = 0.$$

Solving for $\boldsymbol{v}^{(t)}$ in the first equation and substituting the result into the second equation yields the Poisson problem

$$-\nabla^2 p^{(t)} = -\frac{1}{\Delta t} \nabla \cdot \boldsymbol{v}^{(t-1/2)}. \tag{2.12}$$

After obtaining an approximative solution for the pressure $p^{(t)}$ with some iterative solver, the velocity $\boldsymbol{v}^{(t)}$ can be obtained from Equation 2.11 as

$$\boldsymbol{v}^{(t)} = \boldsymbol{v}^{(t-1/2)} - \Delta t \nabla p^{(t)}.$$

This Helmholtz-Hodge decomposition is essentially a projection of $\boldsymbol{v}^{(t-1/2)}$ onto the subspace of solenoidal (divergence-free) functions using the conservative (curl-free) pressure gradient, which is the reason why the method is called projection scheme or pressure correction. The main advantage of the retroactive pressure correction, from a performance point-of-view, is that the incompressibility constraint can be resolved up to any desired accuracy, using very efficient solution methods for Poisson problems. It should be noted, however, that fractional step methods result in a splitting error which puts limits on the accuracy of the time discretization.

### 2.2.2 Semi-Lagrangian Advection Scheme

The semi-Lagrangian scheme (SLS) was originally introduced as a first-order approximation of advection terms in finite difference methods [CIR52] and gained attention in many areas, such as atmospheric sciences [SC91]. The scheme is based on the method of characteristics, which states that for each space-time point $(\boldsymbol{x}, t)$ in a velocity field there exists a characteristic curve $\boldsymbol{p}(\boldsymbol{x}, t; s)$, called particle trajectory, pathline or characteristic, along which the solution of an advection PDE

$$\frac{\partial a}{\partial t} + \boldsymbol{v} \cdot \nabla a = 0 \tag{2.13}$$

remains constant, i.e., $a(\boldsymbol{x}, t) = a(\boldsymbol{p}(\boldsymbol{x}, t; t+\tau), t+\tau)$.

A semi-Lagrangian discretization of Equation 2.13 approximates the solution at each sample point $\boldsymbol{x}_i$ in two steps. First the foot of the characteristic $\boldsymbol{f}(\boldsymbol{x}_i, t) = \boldsymbol{p}(\boldsymbol{x}_i, t; t-\Delta t))$ is found by integrating the ODE of the characteristic backwards in time. Since $\boldsymbol{f}(\boldsymbol{x}_i, t)$ does not coincide with any grid point in general, the second step reconstructs the value at the foot using an interpolation method to obtain the solution as

$$a(\boldsymbol{x}_i, t) = \tilde{a}(\boldsymbol{f}(\boldsymbol{x}_i, t), t-\Delta t). \tag{2.14}$$

The stable fluids method [Sta99] employs the SLS in particular to resolve self-advection in the inviscid Burgers equation

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = 0, \tag{2.15}$$

by approximating each component of the velocity with Equation 2.14.

The original algorithm uses a second-order Runge-Kutta (RK2) method for the approximation of the characteristics and first-order linear interpolation for spatial interpolations. The RK2 approximation

$$\boldsymbol{f}(\boldsymbol{x}_i, t) \approx \boldsymbol{x}_i - \frac{\Delta t}{2}[\boldsymbol{v}(\boldsymbol{x}_i, t-1) + \tilde{\boldsymbol{v}}(\boldsymbol{x}_i - \Delta t\, \boldsymbol{v}(\boldsymbol{x}_i, t-1), t-2)]$$

is second-order accurate with respect to $\Delta t$ and requires only the velocity values of the last two time-steps, without any intermediate values in time. It *does*, however, require intermediate values in space ($\tilde{\boldsymbol{v}}$). The first-order spatial in-

terpolation therefore affects the accuracy of the stable fluids method twice, once in the backtrace along the characteristic (where the velocity is interpolated) and once in the reconstruction at its foot (where the advected quantity is interpolated).

The main advantage of the semi-Lagrangian advection scheme is its unconditional stability which allows simulation at large Courant numbers $C = \frac{v\Delta t}{\Delta x} > 1$. The method is not conservative without special care [LGF11], however, and its accuracy is very dependent on the accuracy of the reconstruction method. In fact, the severe numerical dissipation in the original stable fluids algorithm is mainly a result of the first-order spatial interpolation. Qiu and Shu [QS11] have demonstrated that semi-Lagrangian schemes which employ essentially non-oscillatory (ENO) and weighted ENO (WENO) interpolations for the reconstruction, can achieve high accuracy in smooth parts of the solution and capture sharp interfaces without introducing instability due to spurious oscillations.

## 2.3 Partial Differential Equations on Surfaces

Many scientific applications require solutions for PDEs posed on surfaces. Such problems arise, for example, in biological modeling for the study of wound healing on skin [OMS98], in engineering for the prediction of ice accretion on airfoils and other structures [MCC02, MC04], and in material science where the phase change of a materials on a surface is examined [TQZY05]. In computer graphics, surface PDEs play a role for texture synthesis [Tur91], texture assignment [CLB+09] and for flow simulation on surfaces [Sta03].

In most of these applications one is confronted with an initial value PDE problem which involves differentials intrinsic to some surface embedded in three-dimensional space. A general prototype PDE for a physical transport process on a surface, for example, may be written in non-conservation form as

$$\frac{\partial u}{\partial t} = f(u, \nabla_s u, \nabla_s^2 u).$$

Here the evolution of the function $u : S \to \mathbb{R}$ depends on the gradient $\nabla_s$ intrinsic to the surface $S$ and on the Laplace–Beltrami operator $\nabla_s^2$ on the surface. A variety of techniques has been developed for the solution of such surface PDEs, which can be classified by the applied discretization into mesh-based

(Section 2.3.1), paramterization-based (Section 2.3.2) and embedding-based methods (Section 2.3.3).

### 2.3.1 Mesh-Based Methods

Mesh-based methods discretize surface PDEs directly on a triangulation of the surface. The surface is therefore represented as a polygonal mesh and the dependent variables are stored at each face or vertex of the mesh. The differential operators are approximated using finite difference, finite volume or finite element methods for unstructured grids. Since the differentials are intrinsic to the surface, their discretization usually requires projections onto the surface and differential characteristics such as normal, tangents, principal directions and principal curvatures. It is therefore necessary to provide reliable approximations for these quantities on the triangulated surface.

Mesh-based methods are especially effective for meshes which fulfill high quality standards with respect to regularity and degeneration. For this reason they are very popular in the geometry processing community. The unstructured grids in combination with approximative differential characteristics lead to non-trivial discretizations of the intrinsic differential operators, however, which makes the numerical analysis of these methods quite difficult [BCOS01, RM08].

### 2.3.2 Parametrization Methods

Parametrization methods define a parametric coordinate system on the surface and express the intrinsic differential operators in terms of these coordinates. This requires a parametrization process, which usually defines a mapping between a polygonal triangulation of the surface and parameter domain which can be planar, spherical, cubical, octahedral, or which may be a polygonal domain with a coarse base mesh. The differential operators are typically discretized using standard finite-differences for two-dimensional Cartesian grids. Metric terms in the discrete operators are required, to compensate for the distortion which is introduced by the mapping between surface and parameter domain.

The effectiveness of parametrization methods strongly depends on the quality of the underlying parametrization. For geometries, where smooth and regular global parametrizations are available, relatively precise results can be ob-

tained with very efficient and well-known techniques. Cases where the parametrization contains large distortions or singularities can be quite problematic, however.

### 2.3.3 Embedding Methods

Embedding methods approximate surface PDEs on a Cartesian grid that discretizes the three-dimensional embedding space. They represent the surface implicitly, e.g., as a zero level set of a higher dimensional embedding function. The surface PDE is lifted to the embedding space where solutions are approximated with standard methods for three-dimensional Cartesian grids (e.g., finite differences). Either through modification of the embedding PDE or with additional mechanisms, the solution in the embedding space is obtained in a such way that its restriction to the surface provides a solution to the original problem. Often the efficiency of such methods is improved by limiting the calculations in the embedding space to a small computational band around the surface.

The discretization on volumetric Cartesian grids has the advantage that well-studied numerical techniques can be employed, which enables a thorough error, stability and convergence analysis. The solutions in a higher-dimensional embedding space is computationally very expensive, though.

## 2.4 The Closest Point Method

The closest point method (CPM) is a conceptually simple embedding method for the solution of PDEs posed on surfaces, which was originally introduced by Ruuth and Merriman [RM08]. Later Macdonald and Ruuth presented an implicit alternative to the original two-step explicit algorithm [MR09] and März and Macdonald provided the proofs for the underlying mathematical principles [MM12].

### 2.4.1 Equivalence of Gradients

As in other embedding methods, the computations are performed in an embedding space surrounding the surface (typically a 3D volume) in such a way that the results are consistent with the solution of a surface partial differential equation (cf. Section 2.3.3). The main principle needed in the CPM for this to be true is that of "equivalence of gradients":

$$v(\boldsymbol{x}) = u\big(cp(\boldsymbol{x})\big) \qquad \Rightarrow \qquad \nabla_s u(\boldsymbol{x}) = \nabla v(\boldsymbol{x}), \quad \boldsymbol{x} \in S. \tag{2.16}$$

That is, the intrinsic surface gradient of a surface function $u$ agrees *on the surface* with the standard Cartesian gradient of a volume function $v$, provided $v$ is the closest point extension of $u$ [RM08]. Intuitively this is because the closest point extension $v(\mathbf{x}) = u\big(cp(\mathbf{x})\big)$ is constant in the normal direction to the surface so the change in $v$ must be tangent to the surface. Here, $cp(\mathbf{x})$ denotes the surface point closest to the point $\mathbf{x}$ in Euclidean metric. A second principle applies in a similar fashion to surface divergence operators [RM08]. The two principles can be combined to handle many other differential operators including the Laplace–Beltrami operator. The prove of the equivalence principles also shows that the Jacobian of the closest point function is a projector onto the local tangent space, at points on the surface [MM12]. This observation gives rise to broader class of general closest point functions, which are not limited to the Euclidean metric. Unless specified otherwise, we nevertheless always implicitly refer to the Euclidean metric whenever we use the term 'closest point' within this thesis.

### 2.4.2 Embedding Equations

To build a numerical method on these principles, consider a prototype problem describing the evolution of some attribute $u(\boldsymbol{x}, t)$ on a surface

$$\frac{\partial u}{\partial t} = f(\nabla_S u) \tag{2.17}$$

where $f$ is a general nonlinear function and $\nabla_S u$ is the intrinsic surface gradient, with initial conditions $u(\boldsymbol{x}, 0) = u^{(0)}$. The initial conditions are projected into the embedding space through a closest point extension $v^{(0)}(\boldsymbol{x}) = u^{(0)}\big(cp(\boldsymbol{x})\big)$, so that the equivalence of gradients principle is fulfilled initially. The surface PDE is lifted to the embedding space by replacing the surface gradient with its counterpart in $\mathbb{R}^3$

$$\frac{\partial v}{\partial t} = f(\nabla v) \tag{2.18}$$

and by requiring in a coupled equation that the precondition of the equivalence of gradients principle remains fulfilled at all time

$$v(\boldsymbol{x}, t) = u\big(cp(\boldsymbol{x}), t\big). \tag{2.19}$$

Another requirement, for an embedding method, is that the restriction of a solution $v^{(t)}$ onto the surface is equal to $u^{(t)}$. Instead of Equation 2.19 it is therefore equivalent to say that $v$ must remain its own closest point extension

$$v(\boldsymbol{x}, t) = v\big(cp(\boldsymbol{x}), t\big). \tag{2.20}$$

The system consisting of the embedding PDE 2.18 and the extension side condition 2.20 has only solutions which fulfill the precondition of the equivalence of gradients principle. The restriction of a solution $v^{(t)}$ to the surface is therefore by definition also a solution of the original PDE 2.17.

### 2.4.3 The Explicit Closest Point Method

The explicit closest point method approximates the embedding equations with a fractional-step method. Suppose at some fixed time $t$ we have a solution $v^{(t)}$ defined over the volume which is constant in the normal direction to the surface. We can move the solution of the original PDE 2.18 forward in time by one step of size $\Delta t$ using, for example, a forward Euler discretization. This *evolution phase* requires us to evaluate the right-hand side at time $t$ and by the principles mentioned above, this is exactly the same (for points on the surface) as evaluating $f\left(\nabla v^{(t)}\right)$ in the volume:

$$\tilde{v}^{(t+1)} := v^{(t)} + \Delta t\, f\left(\nabla v^{(t)}\right).$$

The new solution $\tilde{v}^{(t+1)}$ might not be constant in the normal direction and therefore not fulfill the side condition in Equation 2.20. So we then perform a second *extension phase*, a closest point extension which projects values orthogonally off the surface into the surrounding volume to obtain a solution $v^{(t+1)}$ which is constant in the normal direction:

$$v^{(t+1)}(\mathbf{x}) := \tilde{v}^{(t+1)}\big(cp(\mathbf{x})\big), \quad \text{for each point } \mathbf{x}.$$

We can then repeat these two phases over and over to advance the solution in time.

Up to this point we consider only the semi-discrete setting (discrete in time, continuous in space). A complete discretization is required for both the evolution phase and the extension phase, though. The discretization of the embedding PDE in both space and time is typically realized with a finite difference method. Section 2.4.4 exemplifies how a particular finite difference scheme can be obtained with the method of lines. The discretization of the closest point projection includes two steps: Finding the closest point on the surface and obtaining the value of the solution at this point. For the first step an implicit closest point surface representation is required, to efficiently find the closest surface point for each vertex of the discretization grid. Section 2.4.5 names the advantages of this surface representation and shows how it can be generated from an explicit description. Obtaining solution values in the second step requires a spatial interpolation method, since the closest points do not coincide with grid samples in general. Section 2.4.6 presents a viable in-

terpolation techniques and discusses some of their aspects which are relevant to the closest point method.

Before we continue with the details of the discretization, however, we first would like to name a view properties which make the explicit closest point method especially attractive for interactive simulations:

- In the embedding volume, simple well-understood finite difference schemes can be applied to evaluate $f(\nabla v)$.
- Posing the PDE in the embedding volume is easy: in our example, the nonlinear function $f$ is *unchanged* in the volume calcuation and we simply replace intrinsic differential operators with their standard Cartesian counterparts. There are no metric terms to deal with.
- In the extension phase, we need the closest point for each grid point in our embedding volume. This is the only geometry representation of the surface that is needed and it is a very general representation: allowing arbitrary codimension and non-orientable surfaces for example.
- Without effects on the accuracy, the computation can be performed on a narrow band enveloping the surface. Due to the closest point extension, artificial boundary conditions do not need to be applied at the band's exterior.
- The accuracy of the method is well-understood and depends on the accuracy of the time-stepping scheme, the spatial discretization scheme and the interpolation scheme.

### 2.4.4 Method of Lines

The discretization of the embedding PDE, as part of a closest point method, is often realized with the method of lines. This technique approximates an initial value PDE problem with a system of ODEs by discretizing in all but one dimension. Usually the spatial domain is discretized with a structured grid and the spatial derivatives are expressed as finite differences. At each grid point this spatial semi-discretization approximates the original PDE with one ODE in which time remains as the only independent variable. Together with initial and boundary conditions, this results in a coupled system of ODEs that can be solved numerically with a variety of well known time-stepping schemes.

As an example consider the heat equation, respectively linear diffusion equa-

tion

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0, \tag{2.21}$$

where $\alpha$ is the constant thermal diffusivity and $u(\mathbf{x}, t) : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$ is temperature as a function of space and time, with initial conditions $u(\mathbf{x}, 0) = u^{(0)}$. In one dimensional space, $u(x, t)$ can be approximated with a vector $\mathbf{u}(t) : \mathbb{R}^m \to \mathbb{R}^m$ of values $u_i(t)$ on a grid of points $x_i \in R, i = 1, ..., m$. The ODE that approximates Equation 2.21 at each interior point $x_i, i = 2, ..., m-1$ can be defined for example with central differences as

$$\frac{\partial u_i}{\partial t} - \alpha \left( \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} + O(\Delta x^2) \right) = 0.$$

The grid spacing $\Delta x$ generally depends on the type of the spatial discretization grid and the number of grid points. In this thesis we employ Cartesian grids with $\Delta x = 1$ and therefore usually omit the grid spacing terms. The boundary conditions at the endpoints $u_1$ and $u_m$ can be expressed as ODEs in a similar way. The approximation of Equation 2.21 is then given by the coupled system of ODEs

$$\frac{\partial}{\partial t} \mathbf{u}(t) = \mathbf{f}(\mathbf{u}(t)),$$
$$\mathbf{u}(t_0) = \mathbf{u}^{(0)}$$

where $\mathbf{f}$ represents the ODEs for the interior and the boundary points, and $\mathbf{u}^{(0)}$ contains the initial values. To obtain the solution vector $\mathbf{u}$ for a given time $t$, we apply a time-stepping scheme to integrate the ODEs numerically. Applying, for example, the forward Euler method gives for each interior grid point $x_i$ the explicit solution

$$u_i^{(n+1)} = u_i^{(n)} + \alpha \Delta t \left( u_{i-1}^{(n)} - 2u_i^{(n)} + u_{i+1}^{(n)} \right) \tag{2.22}$$

where $n$ is the index for the time variable $t$, which moves forward in steps of $\Delta t$. The method-of-lines approximation above is a FCTS scheme (Forward Time Central Space) with accuracy $O(\Delta x^2) + O(\Delta t)$ and stability condition $\Delta t \leq \frac{\Delta x^2}{2\alpha}$.

Using the method of lines for the lattice-based approximation of PDEs has the advantage that it is easy to interchange the spatial discretization scheme (e.g. higher-order, upwind, non-oscillatory) and the ODE solver (e.g. explicit,

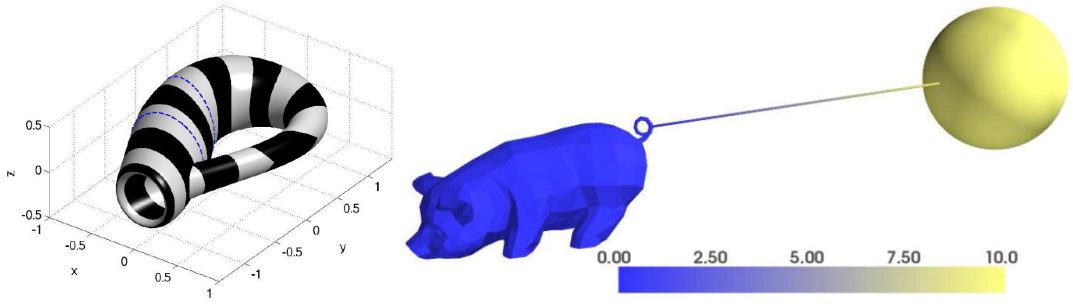


**Figure 2.3**: *The closest point method can be applied to complicated domains. Left: The Klein-bottle is a closed non-orientable surface. Right: The pig-wire-sphere surface consists of two codimension-1 objects connected by a codimension-2 filament. Images courtesy of Colin B. Macdonald [Mac08].*

implicit, linear multistep) independently from each other, to reach high accuracy or high computational efficiency. If we replace, for example, the forward Euler time integration in Equation 2.22 with an Adams-Moulton scheme (average of forward Euler and backward Euler), we obtain

$$u_i^{(n+1)} = u_i^{(n)} + \alpha \Delta t \, \frac{1}{2}\Big( u_{i-1}^{(n+1)} - 2u_i^{(n+1)} + u_{i+1}^{(n+1)} \quad + \quad u_{i-1}^{(n)} - 2u_i^{(n)} + u_{i+1}^{(n)} \Big).$$

This is an unconditionally stable Crank-Nicolson approximation with accuracy $O(\Delta x^2) + O(\Delta t^2)$. In contrast to Equation 2.22 this solution is implicit since it contains unknowns from the next time step and therefore a linear system solve is required for each time step. The explicit CPM does not support implicit time-stepping methods, however, so this approach would only work in combination with the implicit CPM [MR09]. The accuracy of the explicit CPM can still be increased in combination with a higher-order *explicit* time stepping method, though. We can, for example, apply an explicit Runge-Kutta method if we perform a closest point extension after each of its stages.

### 2.4.5 Implicit Surface Representation

The closest point method needs an efficient way to evaluate the closest point function $cp(\boldsymbol{x}) : \mathbb{R}^3 \to \mathbb{R}^3$ at each vertex of the computational grid. It relies therefore typically on a (precomputed) implicit surface representation, in the form of a volumetric Cartesian grid where each vertex stores its closest point on the surface. Vertices in a sufficiently small neighborhood of a smooth surface have a unique closest point. In the vicinity of sharp features (corners,

creases, etc.) and farther away from the surface a grid vertex may have multiple closest points, however. In this case the closest point function is defined to choose one of the closest points arbitrarily. This implicit closest point representation is the only surface representation required for the CPM.

Other embedding methods often use level set surface representations. Many of these methods rely on signed representations, which can result in limitations for surfaces which do not define a clear inside-outside condition in the embedding space. With the closest point method, in contrast, it is straightforward to handle surfaces with boundaries, non-orientable surfaces and surfaces of codimension larger then one, like for example those depicted in Figure 2.3.

The closest point representation can be built in a number of ways. For simple surfaces, like a sphere or a torus, the closest points can be calculated analytically. For parametrized surfaces standard numerical optimization techniques can be employed. In the special case when the surface is already defined implicitly as a level set, a non-Euclidean closest point function can be obtained by following the steepest descent trajectories within an initial value problem [MM12]. For triangle meshes, which are the most common representation in interactive applications, the closest points can be computed either directly or using an approximative closest point transform. Direct methods explicitly compute for each grid vertex the closest points on a number of triangles which come into consideration, and then select the closest one. Closest point transforms, on the other hand, first determine the closest points for a few grid points near the surface and then propagate the closest point information from vertex to vertex using a sweeping algorithm. The second approach has a favorable algorithmic complexity, but its accuracy is limited by the grid spacing. To avoid negative effects on the accuracy of the CPM we employ an exact method in this thesis, which is detailed in chapter 3.

### 2.4.6 Interpolation

An interpolation technique is required to obtain solution values at closest points on the surface in the extension phase of the explicit CPM. For smooth solution functions, Macdonald [Mac08] suggests to use barycentric Lagrange interpolation. This polynomial interpolation technique can be adapted to any desired order of accuracy and its efficiency can benefit from precomputed interpola-

tion weights. An order-$p$ interpolation scheme for a three-dimensional grid is thereby built from the 1D case in the usual dimension-by-dimension manner, i.e., by first performing $(p+1)^2$ interpolations in the $x$ direction, then $p+1$ dependent interpolations in the $y$ direction and then one final interpolation in the $z$ direction.

For non-smooth or discontinuous functions, linear polynomial interpolation techniques suffer from oscillations, however, which can compromise both the accuracy and stability of the closest point method. In such cases it is desirable to adapt the weights to the local behavior of the solution function, for example, to avoid using values from both sides of a discontinuity. For this purpose Macdonald and Ruuth [MR08] introduced an interpolation technique based on previous weighted essentially non-oscillatory (WENO) discretization schemes. Instead of fitting one polynomial with maximum degree through the values in the interpolation stencil, the WENO algorithm considers a number of candidate polynomials from contiguous subsets of the stencil. In smooth regions, the interpolant from the polynomial with maximum degree can be reconstructed from the candidates using a linear combination with ideal weights. Instead of using the ideal weights everywhere, the WENO interpolation includes an additional smoothness indicator, which suppresses the contribution of oscillatory candidate polynomials. By this means the WENO algorithm uses high-order polynomial interpolation only in regions where the solution function is smooth, and falls back to lower-order schemes in non-smooth regions. This behavior has advantages even for PDEs which develop only smooth solutions, since the closest point extension can introduce discontinuities for non-smooth surfaces or surfaces which are purely resolved in the computational grid [MR08].

## 2.5 GPU Computing

Historically, graphics processing units (GPUs) where processors specifically designed for the rasterization-based rendering of 2D and 3D graphics. This special task is inherently data-parallel, since it requires to apply the same series of operations to a large number of graphical primitives, such as vertices, triangles or pixels. The hardware architecture of GPUs is heavily optimized to exploit this data-parallelism. While modern CPUs and GPUs both rely on multithreading, the key distinction is that the former are optimized to run each individual-thread as fast as possible, whereas the latter are optimized to achieve a high overall throughput for a massive total amount of threads. Compared to CPUs, less space on a GPU die is dedicated to control logic, so that more execution units and broader memory interfaces fit onto the chip. Consequentially, the peak floating point performance and memory throughput is considerably higher than on CPUs. Section 2.5.1 shows how this differences manifest themselves in the hardware design of modern graphics processing units.

The distinction between CPUs and GPUs is also reflected in the preferable parallel programming models for each processor type. For CPUs there are many ways to distribute the workload to a number of execution units (hardware threads, cores, processors, machines), but in all models it is favorable to assign a larger chunk of work to each thread in order to keep it busy for a while. GPUs, on the other hand, are almost always programmed with a single instruction multiple thread (SIMT) programming model, which typically assigns only a single data element to each short-living thread. Here it should be noted that this thesis largely follows the GPU computing terminology introduced by NVIDIA, the GPU vendor who coined the term SIMT to distinguish their own programming model from the less specific single instruction multiple data (SIMD) concept. When we talk about GPU programming models on desktop systems, which excludes game consoles, mobile devices and compute clusters, it is necessary to make another important differentiation: Classical rendering tasks are realized in an established real-time graphics API, whereas general-purpose GPU computing (GPGPU) can be implemented in a number of relatively new programming frameworks.

The two most important desktop graphics APIs are OpenGL, by the Khronos group, and Direct3D, by Microsoft. OpenGL has the advantage of being an in-

dustry standard which is available on a number of hardware platforms and operating systems (including Windows). Due to Microsoft's engagement with the Xbox, however, Direct3D is more widespread in the PC gaming industry, which traditionally was the most important driver for GPU development. For this thesis we chose Direct3D as our rendering API mainly because of the better support from GPU-vendors in the last years. Since Direct3D and OpenGL are anyway very similar, most concepts presented in the brief Direct3D introduction in Section 2.5.2 have one-to-one equivalents in OpenGL.

In their current version, OpenGL and Direct3D both offer also a runtime for general purpose programs. These so-called compute shaders are still more intended for graphics-centric applications, however, since they use exactly the same programming languages, data structures and feature sets as the GPU programs for rendering. For writing completely non-graphics-related GPU programs, there is a number of parallel programming frameworks available, including OpenCL (Khronos), C++ AMP (Microsoft) and CUDA (NVIDIA). CUDA is the oldest framework in this list and the only one which was directly developed by a GPU vendor. As of today, it therefore offers the most proprietary GPU programming infrastructure (only NVIDIA GPUs are supported), but also the one with the best support and optimization potentials. These are two of the reasons which provoked us to write all non-graphics related GPU programs for this thesis in CUDA. Another reason is that CUDA was the only completely functional GPGPU framework available at the beginning of this work. We give a brief introduction to CUDA programming in Section 2.5.3.

### 2.5.1 NVIDIA GPU Architecture

GPUs are specifically designed to achieve high throughputs for massively data-parallel tasks. Typically one thread is spawned for each data-element, so that it is quite common to have millions of short-living threads. Figure 2.4 shows that the NVIDIA Kepler GK110 GPU architecture offers up to 2880 simple compute units to run these threads in parallel. Note that we use NVIDIA's current flagship architecture as a particular example, but most of the following discussion is nevertheless also relevant for other GPU architectures. For each so-called CUDA core the GPU can run one scalar operation in parallel. The CUDA cores are contained in 15 individual shader multiprocessors (SMX) which share the same host-interface, L2 cache and memory controllers. Three shader multi-


**Figure 2.4**: *Block diagram of an NVIDIA GPU with Kepler GK110 architecture (according to [NVI12d], adapted). The chip offers up to 2880 CUDA cores for single-precision arithmetic, 960 for double-precision, 480 for transcendental operations and 480 for memory loads and stores. The number of available cores varies between products.*

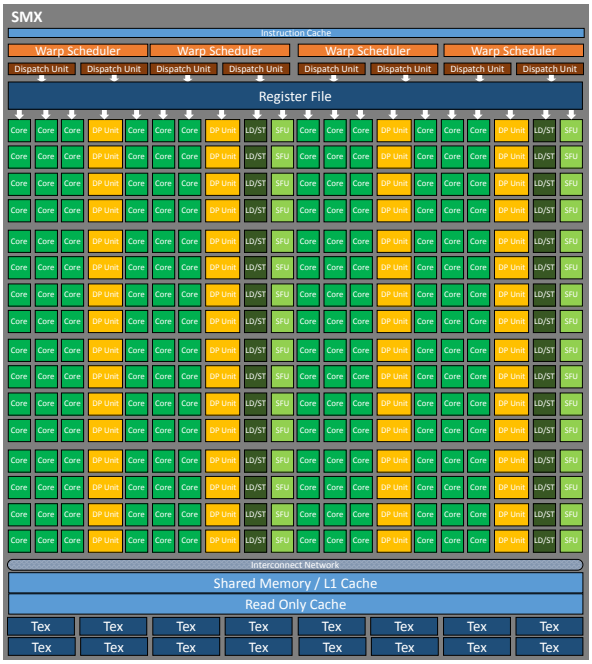


**Figure 2.5**: *Block diagram of one shader multiprocessor (SMX) within an NVIDIA Kepler GK110 GPU (according to [NVI12d], adapted).*

processors together with one rasterization unit form one graphics processor cluster (GPC) [NVI12b, NVI12c] (not depicted in Figure 2.4).

Each SMX, semantically depicted in Figure 2.5, acts as an independent processor core with its own on-chip instruction schedulers, registers, first-level caches and texture units. A SMX can run up to 128 threads concurrently, grouped into 4 *warps* of 32 threads each. The threads within a warp run in lock-step, i.e., all 32 threads must finish the same current instruction before the next dependent instruction can be issued for the entire warp. Two independent instructions can be dispatched concurrently for each of the 4 warps, however, so that the theoretical maximum is 256 instructions per clock-cycle.

To fully utilize all compute units within a SMX, it is necessary to hide latencies resulting from memory access, data dependencies or synchronization. Superscalar pipelines with speculative out-of-order execution and large caches, which are typical for modern CPU cores, would be way to complex for such a large amount of compute units. The shader multiprocessors instead hide latency primarily by switching instantaneously to another warp when the threads within an active warp need to wait for the completion of a high-latency instruction. Each GK110 multiprocessor therefore offers 65536 32-bit registers to keep the stack-data of up to 64 warps (respectively 2048 threads) resident in on-chip memory. Since each thread can use up to 255 registers, it can still be necessary to spill registers to the off-chip device memory, though.

The device memory is connected to the GPU by a 384-bit interface, which provides a higher maximum bandwidth than comparable CPUs. On the Tesla K20X card, for example, there are 6 gigabyte GDDR5 memory available at a bandwidth of 250 gigabyte per second. Two cache layers on the GPU accelerate the access to the device memory. The six memory controllers in the GK110 design buffer all loads and stores in a 1536 kilobyte L2 cache. On top of the L2 cache, each shader multiprocessor provides three additional first-level caches: The 48 kilobyte read-only data cache, formerly known as texture cache, additionally accelerates access to read-only parts of the device memory. The shared memory is a user-managed cache which is directly accessible through the respective compute API. The L1 cache is primarily used for register spills, but not for direct loads and stores. Access to R/W device memory is therefore only handled through the L2 cache on the GK110. The shared memory and the L1 cache reside both in the same 64 kilobyte on-chip memory and the user can decide how this memory is divided among the two.

### 2.5.2 Direct3D

Direct3D is an API within the Windows software development kit which enables the rasterization-based rendering of 3D graphics [Mic12]. The API is based on an abstract graphics pipeline, which describes the drawing of three-dimensional primitives onto two-dimensional raster images using dedicated graphics processing hardware. Figure 2.6 depicts the draw pipeline of the current Direct3D version 11. The rounded rectangles thereby represent programmable stages of the pipeline, called shaders, for which the developer must provide an implementation in the form of an high-level-shader-language (HLSL) function. Each shader has read-only access to resources such as textures, buffers and constants, which are stored in the memory of the graphics device. This programmability allows the developer to utilize each shader stage for a large variety of different purposes, while the predictable and well-defined overall drawing mechanism still enables a very specialized hardware design. In the following we describe the individual stages of the draw pipeline:

- The **input assembler stage** is invoked for each three-dimensional primitive that should be drawn. It reads the input data for each point, line, triangle or patch from user-filled vertex and index buffers and generates streams of assembled primitives which feed the following pipeline stages. Each part of a primitive is augmented with a system-generated value, which helps to avoid redundant calculations and which serves as an identifier within the shaders. Optionally, the input assembler stage can also generate information about the adjacency of the primitives.

- The **vertex shader stage** is invoked (at least) once for each single vertex of a primitive. Its input are the assembled per-vertex attributes and its output contains the position of the vertex and additional user-defined per-vertex attributes. If all optional pipeline stages are omitted, the vertex shader must at least transform the position into the homogeneous clip-coordinates system required by the rasterizer stage.

- The **hull shader stage** is the first of the three optional tessellation stages and responsible for the setup of the two following stages. It is the only programmable stage which consists of two shader functions. The patch-constant hull shader is invoked once for each isoline, triangle patch or quad patch that should be tessellated, whereas the control-point hull

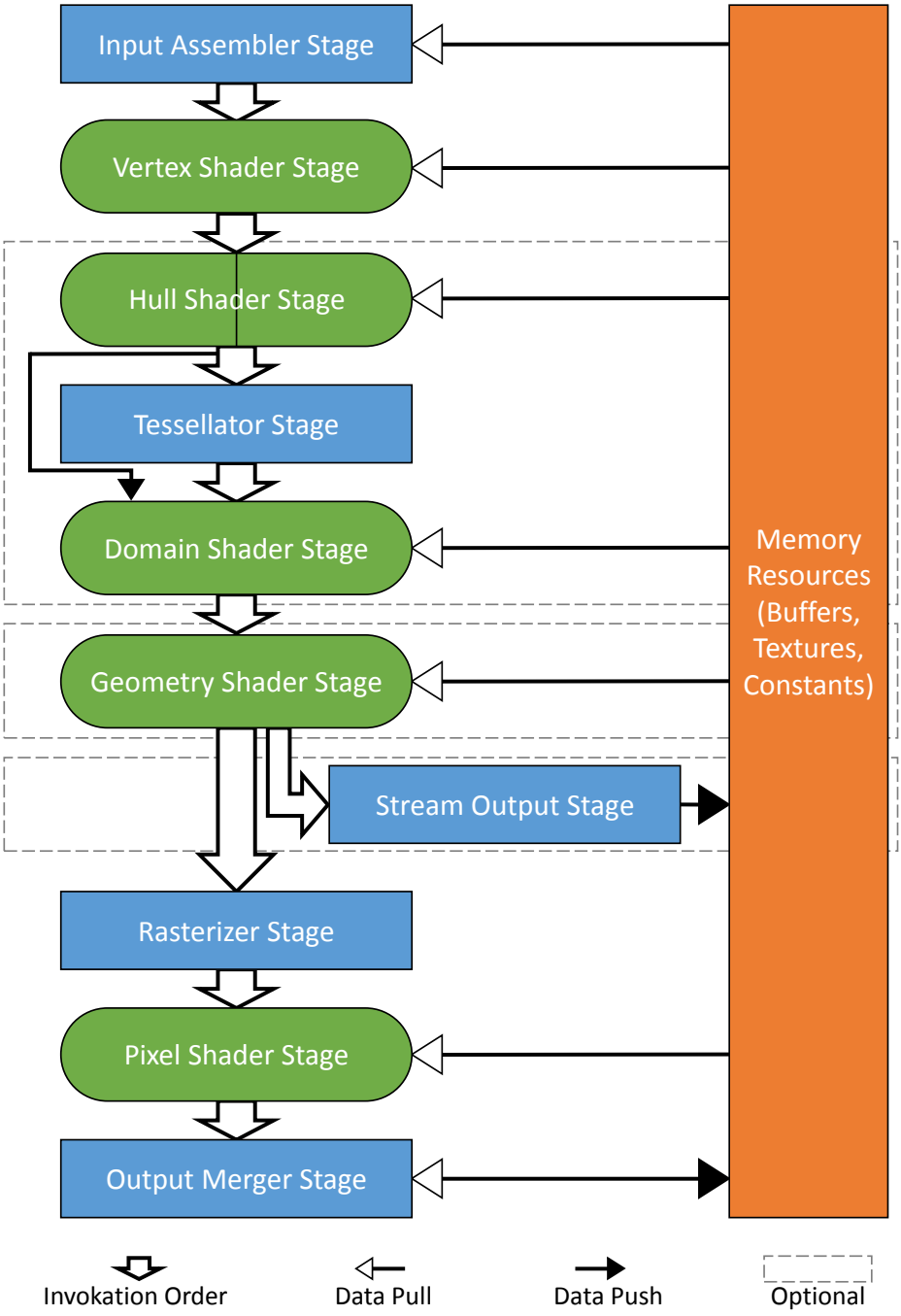


**Figure 2.6**: *Direct3D 11 graphics pipeline (according to [Mic12], adapted)*

shader is invoked once for each output control-point of a patch. Both shaders take as input an array containing the per-vertex attributes of all input control-points of the patch, which do not necessarily correspond to the output control-points. The input of the control-point hull shader typically also includes the system-generated ID of the output control-point. The output of the two hull shaders contains a number of per-patch tessellation factors and additional per-patch or per-output-control-point attributes.

- The **tessellator stage** is invoked once for each line, triangle or quad patch. Its only inputs are the tessellation factors and the type of partitioning. The tessellator subdivides the normalized patch domain into smaller primitives (triangles, lines or points) and outputs the patch-coordinates of each generated vertex as well as the topology information of each generated primitive.

- The **domain shader stage** is invoked once for each output vertex of the tessellator stage. Its input includes the coordinates of this vertex in the normalized patch domain, as well as all per-patch and per-control-point attributes generated by the hull shader stage. Using the patch-domain coordinates, the domain shader typically interpolates the information from the patch and its control points in order to generate the position and additional attributes of the output vertex.

- The optional **geometry shader stage** is invoked once for each primitive that is generated by either the input assembler or the tessellator stage. The geometry shader can discard this primitive or emit one or more new primitives, of the same or of a different type. Its input is an array containing the attributes of all vertices which belong to the input primitive. If adjacency information is available, the input can also contain the per-vertex attributes of adjacent input primitives. The shader must append the per-vertex attributes of the generated primitives to one or more output streams. By using multiple output-streams and system-interpreted values, the geometry shader can redirect the output on a per-primitive basis to different render targets and to the stream output stage.

- The optional **stream output stage** is invoked once for each output primitive of the geometry shader stage. It stores each processed primitive in

a buffer resource within the graphics device memory. For further processing in subsequent draw passes, this primitive buffer can be bound as input to the pipeline or as a shader resource.

- The **rasterizer stage** is invoked once for each primitive generated by either the input assembler, the tessellator or the geometry shader stage, depending on which stages are active. Its inputs are the primitive topology (triangle, line, point), the vertex positions in homogeneous clip space and the additional per-vertex attributes. The rasterizer stage scan-converts the primitive into a raster image and outputs one fragment for each covered pixel. This process includes the clipping to the view-frustum, the mapping to the 2D viewport and the perspective-correct interpolation of all per-vertex attributes.

- The **pixel shader stage** is invoked once for each fragment generated by the rasterization stage. Its inputs are the interpolated vertex attributes of the fragment and its output is the color of the fragment in one or more output images respectively render targets. The pixel shader can optionally also discard fragments, change their depth value or output values for a stencil image.

- The **output merger stage** is invoked once for each fragment generated by the rasterizer and its input is the fragment data generated by the rasterizer and the pixel-shader stages. The output merger discards fragments which fail the depth or stencil tests and writes the remaining fragments into the render-target image. In this process it can perform a blending operation to combine the fragment value with the value stored in the render target.

### 2.5.3 CUDA

The NVIDIA CUDA toolkit is a complete GPU programming framework for the C and C++ language, available for Microsoft Windows, Mac OS X and Linux. In the CUDA programming model, the GPU operates as a co-processor of the CPU, i.e., the graphics card acts as a physically separate *device* controlled by the *host* system which runs the program. Through calls to the CUDA-runtime, the host manages memory allocations on the device as well as data transfers and synchronizations between the host and the device (or between different devices).

The parts of the program which should run on the GPU are encapsulated in so-called kernels. A kernel is a C function which is invoked by a single thread on the host, but which is processed on the device by a large number of threads in parallel. The dynamic total number of GPU threads is determined from the execution configuration, which is specified for each kernel call individually.

The execution configuration organizes the threads in a two-level hierarchy, i.e., a grid of thread blocks. All threads within a block run on the same shader multiprocessor of the GPU and share the same lifetime. They can therefore use synchronization and memory fences to communicate via device memory or via the on-chip shared memory. In the latter case the execution configuration must specify the amount of shared memory required per thread block, additionally to the mandatory block and grid sizes.

Inside a kernel function, each GPU thread has access to system-generated values which identify its position within the thread block, the position of its block within the grid of blocks and its global position within the entire call. These base indices are typically employed to determine the memory addresses of the data elements on which the thread operates. Through the usual C pointer arithmetic and array syntax, each thread has general random read-and-write access to the shared memory and the device memory. To avoid concurrency hazards, such as read-after-write, write-after-read or write-after-write, it is necessary to correctly use atomic operations, block-level synchronization or memory fences if multiple threads operate on the same data-element (within one kernel or within kernels of concurrent streams).

While CUDA makes the development of GPU programs much more accessible for C/C++ developers who are not familiar with computer graphics, it still requires a lot of in-depth knowledge about massively-parallel programming to successfully optimize the performance of such programs. Al-

ready the brief CUDA programming guide dedicates 30 pages to performance guidelines [NVI12b]. On top of that, NVIDIA provides 60 pages of best practices [NVI12a] and additional performance guidelines for each GPU generation, to name only the official first-party information. In summary of these documents, we identify three main strategies to optimize the performance of CUDA programs: Maximize the utilization, maximize the memory throughput and maximize the instruction throughput. For each individual CUDA kernel, the appropriate strategy depends on which factor currently limits the performance the most.

The best utilization is achieved when the GPU can issue the theoretically maximal number of instructions at any given time, i.e., when latency can be completely hidden through active-thread switching. At the application level, developers should therefore decide which tasks fit to the GPU and which are better handled on the CPU, but at the same time they must also avoid idle times resulting from unnecessary CPU-GPU synchronization. At the device level, such idle times can be avoided to some extent with concurrent kernel execution, but possible concurrency hazards must be considered in this case. At the multiprocessor level respectively kernel level, the utilization can be enhanced by avoiding high-latency instructions like global memory access, by avoiding unnecessary data dependencies and synchronization points, and by lowering the register and shared memory requirements in order to fit more threads onto the same multiprocessor in parallel.

To take full advantage of the high memory throughput of the GPU, it is first and foremost always a good idea to use the fastest available memory for each sub-task. This can mean to keep kernel-interim results in device memory and instruction-interim results in shared memory, or to optimize for cache efficiency. Additionally, it is always required to pay attention to the memory access characteristics of each GPU architecture. A global memory transaction is only fully utilized, for example, if the inter-thread access pattern fulfills certain architecture-specific size, alignment and coalescing requirements. Similar requirements exist for the prevention of bank conflicts in shared memory.

When the instruction throughput is the bottleneck of a kernel, this can be the consequence of excessive intra-warp divergence. In this case one can avoid the problematic branches entirely, reduce the number of instructions within a branch or reorganize the threads so that divergent branching becomes less likely within each warp. Also the choice of the instructions themselves can

be suboptimal. The troughput of arithmetic instructions largely depends on the data-type, for example, but too much type conversations can be counterproductive since these operations also have a relatively low throughput. Finally a kernel can often be optimized by identifying unnecessary instructions, which is especially effective if high-latency or low-throughput instructions can be avoided.

# 3

# Real-Time Fluid Effects on Surfaces using the Closest Point Method

In this chapter we present the numerical solution of the wave equation and the incompressible Navier-Stokes equations on surfaces via the closest point method (CPM), and we demonstrate surface appearance and shape variations in real-time using this method. To fully exploit the potential of the closest point method, we present a novel GPU realization of the entire CPM pipeline. We propose a surface-embedding adaptive 3D spatial grid for efficient representation of the surface, and present a high-performance approach using CUDA for converting surfaces given by triangulations into this representation. For real-time performance, CUDA is also used for the numerical procedures of the closest point method. For rendering the surface (and the PDE solution) directly from the closest point representation without the need to reconstruct a triangulated surface, we present a GPU ray-casting method that works on the adaptive 3D grid.

## 3.1 Introduction

Techniques for solving partial differential equations (PDE) have numerous applications in physics-based simulation and modeling, and they are frequently employed in computer graphics for realistically simulating real-world phenomena such as fluids or deformable solids. Most commonly, partial differential equations over $\mathbb{R}^2$ or $\mathbb{R}^3$ are considered, and, since analytical solutions only rarely exist, numerical solutions using some form of spatial discretization are employed.

In computer graphics, numerical solutions of PDEs are also used to produce effects on manifolds such as surfaces in $\mathbb{R}^3$. Prominent examples include surface texture synthesis [Tur91], texture assignment [CLB+09], or flow simulation on surfaces [Sta03]. If an isometric surface parametrization exists, a PDE defined on a surface can be transformed into a PDE on the 2D parameter domain and solved using standard discretizations [Sta03, LWC05, LFW07]. PDEs on surfaces can also be solved directly using finite element discretizations on a surface triangulation [DE07]. These techniques, however, rely on changing the non-parametric form of the PDE to agree with the underlying discretization. An alternative are so-called embedding techniques [BCOS01, NNRW09, CLB+09, RM08, CRT04], which lift the PDE to a narrow band around the surface and solve a transformed PDE on this band.

Embedding techniques are particularly attractive because they do not rely on the existence of a low distortion parametrization. The computation of such a parametrization can be difficult to achieve, if not impossible, and it is not suitable in applications where the surface undergoes frequent shape changes. Another advantage of embedding techniques is that once the PDE has been lifted to the embedding 3D space, standard numerical schemes can be used in this space to efficiently solve the PDE.

The closest point method, introduced in Section 2.4, is one of the less complicated embedding methods in that it only requires the existence of a closest point representation of the surface (i.e., for any point in the embedding space, the surface point with the least Euclidean distance is known) and it solves completely standard PDEs—without any metric terms—in the embedding space using common numerical methods on a uniform Cartesian grid ( [RM08]). In addition, the closest point method can handle general open surfaces and surfaces without orientation, and, thus, is applicable even in scenarios where the surface separates into parts.

Despite its simplicity and accuracy, however, a naïve implementation of the closest point method is exhaustive in memory. Such an implementation would typically construct a full surface-embedding Cartesian grid and precompute at any grid point its closest surface point. It is clear that this severely limits the resolution of the computational grid to be used. Furthermore, the closest point representation remains fixed only under the assumption that the surface does not change. In scenarios where this is not the case, re-computing closest points can have a severe impact on performance.

## 3.2 Contribution

The primary focus of this chapter is the simulation of fluid effects on surfaces in real time using the closest point method. To achieve this, we present a fast GPU method for realizing the closest point method *and* rendering the simulation results. The method can efficiently solve non-linear PDEs on surfaces using a high-resolution embedding grid, and it can calculate and render the resulting fields independently of the input surface resolution. Due to the strict separation of the surface representation and the computational grid, our method supports initial surfaces of arbitrary topology and geometric detail. The method is implemented in CUDA and Direct3D10, and all parts have been optimized to make the best possible use of the massive parallel processing power and memory bandwidth on current GPUs. In particular we contribute:

- A novel CUDA method for constructing an adaptive multiblock closest point grid representing a narrow spatial band around a given triangulated surface.
- A closest point method for numerically solving the wave equation and the Navier-Stokes equations on arbitrary surfaces.
- A GPU ray-caster that works directly on the adaptive closest point grid and can visualize surface appearance attributes and surface displacements.

This chapter is structured as follows: In 3.3 we discuss previous work that is related to ours. Section 3.4 introduces the multi-block discretization grid and the closest point representation which underlie our simulation and rendering techniques. Section 3.5 shows how this adaptive discretization can be generated efficiently for a surface given in a triangulated representation. Section 3.6 sheds light on the numerical solution of the second-order linear wave equation and the incompressible Navier-Stokes equations via the closest point method. Besides discretization aspects, the effects of the closest point extension on stability, accuracy and performance of the simulation are discussed. Section 3.7 introduces the volume raycaster that is used to visualize dynamic color and displacement effects independently of the surface triangulation. Section 3.8 presents a detailed performance analysis. The chapter is

concluded with a discussion of the advantages and limitations of the proposed method and some ideas for future work.

## 3.3 Related Work

An interesting surface-based problem is the simulation of fluid dynamics on surfaces. Stam applied a modified 2D version of his stable fluid dynamics simulation [Sta99] to the patches of Catmull–Clark surfaces and introduced a technique to transfer information in an overlap between the patches [Sta03]. While working well in regular areas of the surface, the approach leads to errors at non-valence-four vertices. For techniques relying on some sort of global parametrization similar problems can be expected at the singularities.

Several other authors directly use the vertices and edges of triangle meshes as a discretization and employ finite element, finite volume, Lattice Boltzmann or related numerical methods in their solvers [SY04, FZKH05, NMZ07, ATBG08]. This however takes away some advantages of a regular grid discretization as finite element methods are generally more complicated. Particularly on surfaces, the resulting algorithms may be quite difficult to implement in practice, e.g. [RWP06]. Finally, if the input surface is not given as a uniform triangulation a costly reconstruction procedure may be required.

The interest in simulation on surfaces also resulted in particle based solutions. Turk [Tur91] simulated reaction diffusion systems in a grid formed by particle relaxation on a surface, with connectivity given by a Voronoi diagram. Later, the technique was adapted to general shallow water equations [WMT07]. Bürger et al. [BKW10] use an orthogonal fragment buffer to trace particles along a surface in order to color the surface.

Chuang and colleagues [CLB+09] proposed a volume embedding scheme based on B-splines as Ansatz-functions in a hexahedral simulation grid. By restricting the basis functions to a surface instead of a sub-volume, a weak form of a surface PDE can be solved independently of the 3D simulation domain, but at the expense of explicitly clipping the surface mesh against the simulation cells.

The closest point method was introduced by Ruuth and Merriman [RM08]. Later works presented an implicit time stepping [MR09] and used the closest point method for the evolution of level sets [MR08] or segmentation on surfaces [TMR09]. Hong et. al [HZQW10] applied the closest point method to

fire simulation on animated surfaces. Due to the embedding used, the closest point method relies on certain properties of distance transforms, such as smoothness of properties close to the surface. Jones [JBS06] provides an excellent survey of these properties.

An additional topic related to our method is GPU-based surface voxelization of triangle meshes. We borrow ideas from several authors [DCB+04, ED06, ZCEP07, SS10, Pan11] and perform a surface voxelization to determine all voxels lying within a computational band of arbitrary thickness around the surface. For these voxels, the closest point method surface representation stores the closest points on the surface. As such, our technique computes a partial distance volume around the surface, including the positions of the closest points. Building upon the work of Pantaleoni, Schwarz and Seidel [SS10, Pan11], we demonstrate the efficient use of the CUDA parallel programming API for constructing a multiblock closest point grid.

## 3.4 Adaptive Multiblock Closest Point Method

As discussed in Section 2.4.5, the discrete closest point surface representation required by the closest point method is in principle a volumetric Cartesian grid, where each grid cell stores the surface point closest to its center. When the surface is smooth, the closest point is unique in a small band around the surface. However, in our application where the closest point method is used to simulate effects on surfaces given by piecewise linear triangle meshes, the surface is only $\mathscr{C}^0$ at the edges. Thus, discontinuities in the closest point field can arise, and for a single grid point more than one closest point can exist on the surface. In this case the closest point method selects one of these closest points arbitrarily. The closest point extension propagates such discontinuities also to the fields which store the simulation attributes. Both the extension phase and the evolution phase must therefore employ numerical schemes that are tolerant of discontinuities.

For high resolution simulations, the use of a uniform simulation grid is impractical. For the closest point method, on the other hand, it is sufficient to work on two narrow computational bands around the surface: The *evolution band* and the *extension band*, which are used in the evolution and extension phase of the closest point method, respectively. This suggests using an adaptive multiblock grid, which stores only those blocks of the full grid which

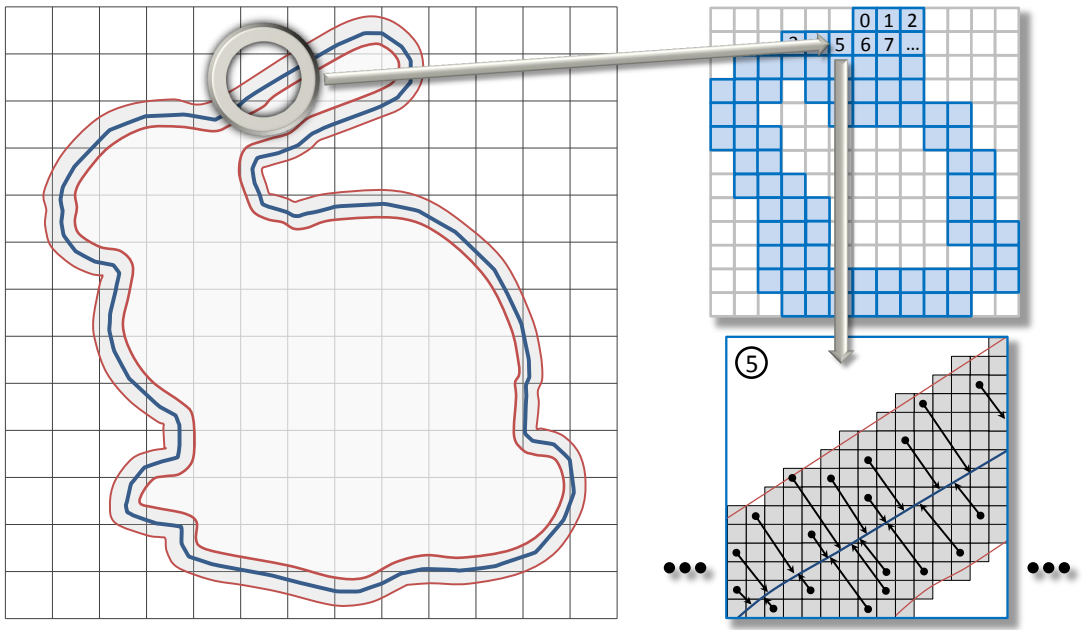


**Figure 3.1**: *Sparse closest point grid. Left: A grid of blocks, some of them overlapping the narrow band around the surface. Top right: Only blocks overlapping the narrow band around the surface are stored in linear order. Bottom right: The closest points of the grid cells within these blocks.*

overlap these bands. Figure 3.1 illustrates such a multiblock grid in 2D. The widths we choose for the bands are the ones proposed by Macdonald and Ruuth [MR08] (Section 4.1.1).

The multiblock grid is comprised of two levels. The coarse level consists of a regular 3D grid, where each cell represents a block of cells in the uniform simulation grid. The fine level consists of the grid cells of those blocks which are intersected by the computational band. All data is stored in linear buffers. The coarse-level cells representing an intersected block store the position in the fine level buffer at which this block is laid out. For constructing a closest point surface representation we first compute the blocks which are intersected by the computational band. This is performed by determining for every triangle separately the blocks which are intersected by the band around it. The cells of a block determined in this way are stored in a contiguous region in the fine-level buffer. To allow for an efficient computation of the closest point of each fine-level cell, we also compute the set of triangles intersecting this block. For each cell of the same block we then iterate over the corresponding set of triangles and compute the closest surface point. Finally, the simulation is carried

out on the fine-level cells.

## 3.5 Closest Point Method on the GPU

Since we are aiming to support both static and time-varying surfaces, for instance surfaces that are modified in turn by the closest point method simulation, an important requirement is that the creation and update of the sparse closest point representation can be performed at high rates even for high-resolution surfaces and simulation grids. For this computationally and bandwidth intense task, we exploit the massively parallel architecture of the GPU.

Our proposed algorithm for constructing a closest point volume is inspired by recent CUDA voxelization approaches [SS10, Pan11]. Similar to the tile-based approaches proposed in these works, we use a sort-middle rasterization pipeline which first assigns triangles to coarse blocks and then performs a fine-grain voxelization and closest point calculation per block. There are, however, some specific differences of our GPU algorithm:

- It computes a surface voxelization using a distance criterion that is based on the thickness of the computational band around a surface.
- It calculates for every voxel within the computational band its closest surface point and the distance to this point.
- It constructs an adaptive 2-level multiblock grid designed for efficient simulation.
- It parallelizes over fine level cells instead of triangles to avoid shared memory atomics in the closest point calculation.

### 3.5.1 Data Layout and Grid Generation

For the sake of clarity, we will first describe the GPU construction of the sparse multiblock grid, including the indexing schemes that are required to efficiently perform the closest point method, before we go into the details of the actual closest point calculation.

To create the sparse simulation grid, the user first selects a simulation resolution by specifying the size $\Delta \boldsymbol{x}$ of one simulation cell. Then the bounding volume of the triangle mesh—enlarged to respect the width of the computational band—is subdivided into $k \times l \times m$ equally sized blocks. We assume that each block contains $b^3$ simulation cells, so that the values of $k, l$, and $m$ can

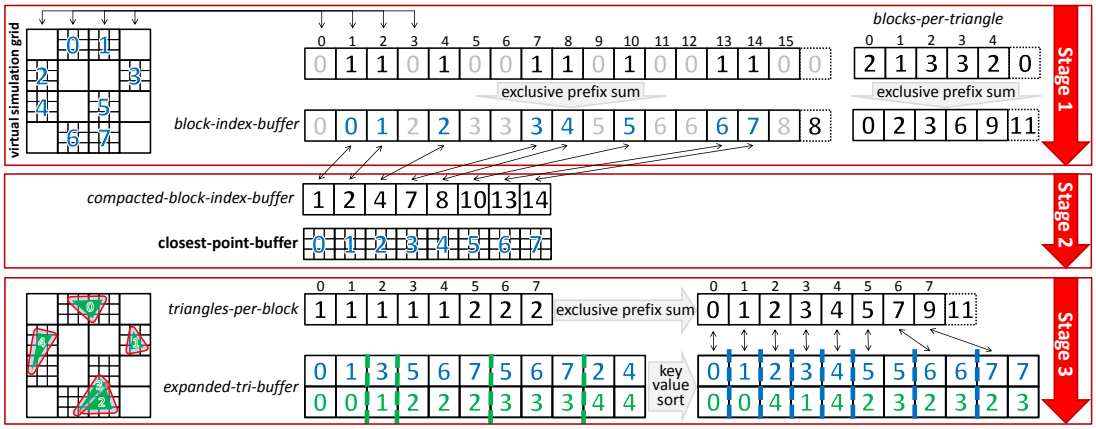


**Figure 3.2**: *Overview of the different GPU buffers that are created to facilitate an efficient realization of the closest point method.*

be computed.

The grid construction on the GPU consists of three stages: First, we determine which and how many blocks are intersected by the computational band. The *block-index-buffer* is used to keep track of this information. Second, the *closest-point-buffer*—large enough to store the closest points of all simulation cells represented by these blocks—is allocated. To accommodate fast access to cells in adjacent blocks, an additional buffer—the *compacted-block-index-buffer*—is created. It has as many entries as there are intersected blocks, and it stores the positions of these blocks in the *block-index-buffer*. From this, and by taking into account that in the *block-index-buffer* all blocks are laid out in x/y/z-order, adjacency information can be resolved. Third, the *expanded-tri-buffer* is constructed to support coalesced memory write operations in the closest point construction (see the following subsection). This buffer stores, in contiguous sections, the sets of triangles that are required by each block to compute the closest point grid. The different buffers are illustrated in Figure 3.2. In the following, we will describe the parallel CUDA implementation of the three stages.

**Stage 1:** One CUDA-thread is launched per triangle, and each thread determines the blocks that are intersected by the computational band around this triangle. Therefore, each thread computes the triangle's axis aligned bounding box—enlarged by the width of the computational band—and computes the blocks that are intersected by this box. Since testing against the bounding box can result in false positives, i.e., blocks that are intersected by the bounding

box but not the computational band, an additional test of the blocks is performed to prune as many of them as possible: For every intersected block, the thread calculates the distance from the block center to the triangle and skips the block if its bounding sphere does not intersect the triangle's computational band. By writing a 1 into *block-index-buffer* at the positions of the remaining blocks, these blocks are marked as intersected. Because all threads write the same value, a synchronization of the write operations is not necessary in this case.

At the end of stage 1, each thread writes the number indicating how many blocks are intersected by the triangle to a temporary buffer—*blocks-per-triangle*—in global GPU memory. Finally, an exclusive parallel prefix sum [Har07] is computed in-place over *block-index-buffer* and *blocks-per-triangle*. At the positions that were marked, the *block-index-buffer* now contains the relative positions of the respective block in the sequence of marked blocks.

**Stage 2:** Since the prefix sum operation also calculates the total number of marked blocks, the *closest-point-buffer* can be allocated in GPU memory. Then, a CUDA kernel with one thread per entry in *block-index-buffer* is executed. The $i^{\text{th}}$ thread reads the index value *val* from the buffer at position $i$ and compares it to the value that is stored at position $i+1$. If the values are different, the thread writes the value $i$ into the *compacted-block-index-buffer* at position *val*.

**Stage 3:** We start by allocating the *expanded-tri-buffer* in GPU global memory using the size indicated by the prefix sum over *blocks-per-triangle*. The elements of this buffer are pairs of block-triangle indices, and for each triangle these pairs are written in succession into the buffer. Since the parallel scan operation also gives the starting positions of each set of block-triangle pairs in *expanded-tri-buffer*, a CUDA kernel with one thread per triangle is executed to first build these sets in shared memory—including the computation of intersected blocks as described before—and then to write them into the buffer at the starting positions (see the first occurrence of *expanded-tri-buffer* in Figure 3.2). In the same kernel, we use CUDA's *atomicAdd* operation to fill a buffer *triangles-per-block*, which stores for every marked block the number of triangles it intersects.

In a second pass, *expanded-tri-buffer* is sorted with respect to the block ID using the CUDA radix sort [Har07], and a prefix sum over *triangles-per-block* is calculated. From the content of the resulting buffers the indices of all triangles

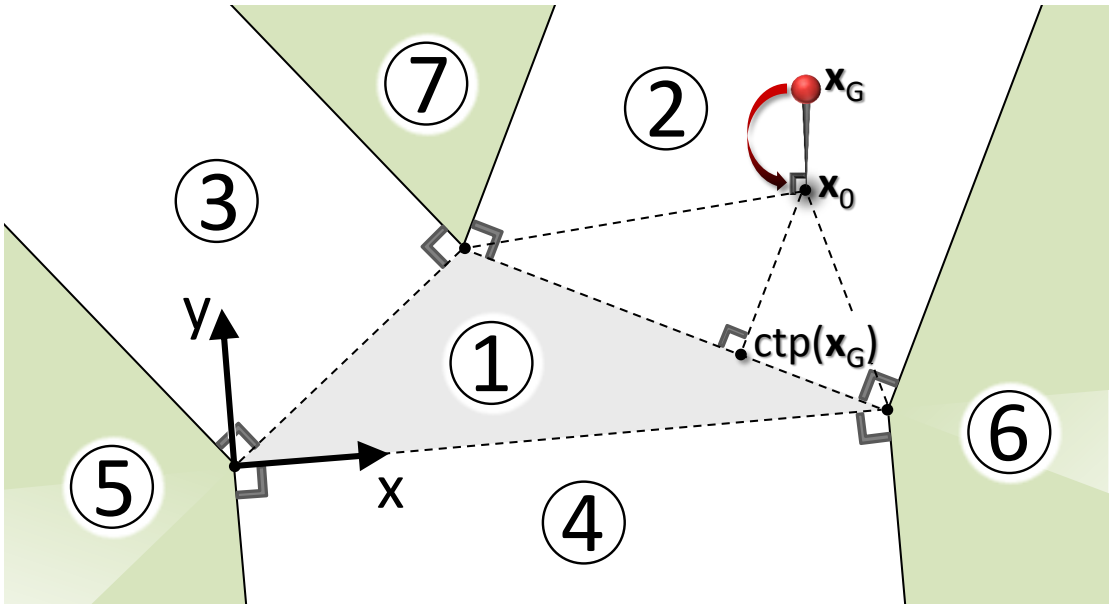


**Figure 3.3**: *In each of the 7 zones, the grid vertex $x_G$ is closest to one region of the triangle (face, edge, corner). Please note that in the literature the outer zones are often defined by extending the triangle edges. This leads to errors if the closest point is incorrectly assumed to lie on an obtuse corner instead of an edge.*

contributing to the closest point representation for a particular block can be determined.

### 3.5.2 Closest Point Computation

For each of the cells in the *closest-point-buffer*, all of which are uniquely defined by the block they belong to and their relative position in the sequence of cells of this block, the point on the surface that is closest to the cell's center has to be computed (we will call these centers the grid vertices from now on). The parallel CUDA implementation we propose is optimized to reduce global memory operations by performing a block-wise closest point computation in shared memory, and writing the results en-block into global memory.

The CUDA kernel to compute the closest point representation executes one warp of CUDA threads per grid block. Every thread computes the closest point of exactly one grid vertex, by sequentially traversing the triangles that are stored in *expanded-tri-buffer* for that block. Thus, the threads of one warp process a number of grid vertices in parallel, in the same order they lie in memory, meaning that all required global memory buffers are accessed only by co-

**Algorithm 1** Closest Point on Triangle

**Input, Grid Vertex**
Position $\boldsymbol{x}_G \in \mathbb{R}^4$, where $\mathbb{R}^4$ denote homogeneous coordinates over $\mathbb{R}^3$

**Input, Triangle**
Triangle vertices $\boldsymbol{x}_U, \boldsymbol{x}_V, \boldsymbol{x}_W \in \mathbb{R}^4$.
Projection matrix $\boldsymbol{M} \in \mathbb{R}^{4\times 2}$ that transforms into a 2D coordinate system in which $U$ is the origin, $V-U$ is on the positive $x$-axis and $W$ lies in the $xy$-plane (first half of a look-at matrix).
Projected triangle vertices $\boldsymbol{x}_1 = \boldsymbol{M}\,\boldsymbol{x}_U, \boldsymbol{x}_2 = \boldsymbol{M}\,\boldsymbol{x}_V, \boldsymbol{x}_3 = \boldsymbol{M}\,\boldsymbol{x}_W$.
Vectors $\boldsymbol{v}_{23} = \boldsymbol{x}_3 - \boldsymbol{x}_2$, $\boldsymbol{v}_{31} = \boldsymbol{x}_1 - \boldsymbol{x}_3$ and $\boldsymbol{v}_{24} = (\boldsymbol{v}_{23}.y, -\boldsymbol{v}_{23}.x)^T$, $\boldsymbol{v}_{35} = (\boldsymbol{v}_{31}.y, -\boldsymbol{v}_{31}.x)^T$.

**Output**
Closest point to $\boldsymbol{x}_G$ on triangle $\boldsymbol{x}_{ctp} \in \mathbb{R}^4$.

**Procedure**

$\boldsymbol{x}_0 \leftarrow \boldsymbol{M}\,\boldsymbol{x}_G, \qquad \boldsymbol{v}_{20} \leftarrow \boldsymbol{x}_0 - \boldsymbol{x}_2, \qquad \boldsymbol{v}_{30} \leftarrow \boldsymbol{x}_0 - \boldsymbol{x}_3$
**if** $\boldsymbol{x}_0.y < 0$ **then**
  **if** $\boldsymbol{x}_0.x < 0$ **return** $\boldsymbol{x}_U$
  **if** $\boldsymbol{v}_{20}.x > 0$ **return** $\boldsymbol{x}_V$
  **return** ClosestPointOnEdge($\boldsymbol{x}_P, \boldsymbol{x}_U, \boldsymbol{x}_V$)
**else if** $\boldsymbol{v}_{24} \cdot \boldsymbol{v}_{20} < 0$ **then**
  **if** $\boldsymbol{v}_{23} \cdot \boldsymbol{v}_{20} < 0$ **return** $\boldsymbol{x}_V$
  **if** $\boldsymbol{v}_{23} \cdot \boldsymbol{v}_{30} > 0$ **return** $\boldsymbol{x}_W$
  **return** ClosestPointOnEdge($\boldsymbol{x}_P, \boldsymbol{x}_V, \boldsymbol{x}_W$)
**else if** $\boldsymbol{v}_{35} \cdot \boldsymbol{v}_{30} < 0$ **then**
  **if** $\boldsymbol{v}_{31} \cdot \boldsymbol{v}_{30} < 0$ **return** $\boldsymbol{x}_W$
  **if** $\boldsymbol{v}_{31} \cdot \boldsymbol{v}_{10} > 0$ **return** $\boldsymbol{x}_U$
  **return** ClosestPointOnEdge($\boldsymbol{x}_P, \boldsymbol{x}_W, \boldsymbol{x}_U$)
**end if**
**return** ClosestPointOnPlane($\boldsymbol{x}_P, \boldsymbol{x}_U, \boldsymbol{x}_V, \boldsymbol{x}_W$)

alesced, fully utilized memory transactions. Furthermore, since the threads of a warp are executed in lock-step and every grid vertex is processed by exactly one thread, all synchronization and atomic instructions can be omitted.

In shared memory, two buffers are allocated to store the closest point coordinates and the distances of the grid vertices to these points. All threads of one warp first perform a coalesced global memory read operation to fetch the triangles from *expanded-tri-buffer* that are needed to compute the closest points. The number of triangles is read from *triangles-per-block*.

Each thread computes the grid vertex for the cell it is working on. Then, it iterates over all triangles, and for each of them it computes the closest point and corresponding distance. If the distance is shorter than the one stored in the shared memory buffer, both the distance and the closest point coordinate are updated. The closest point computation is performed by first determining whether the grid vertex is closest to a corner, an edge, or the triangle's interior. By a coordinate system transformation that brings the triangle into one of the coordinate planes, this essentially breaks down to a 2D problem: According to the illustration in Figure 3.3, we need to identify in which of the 7 zones of the triangle the projected vertex $\boldsymbol{x}_0$ lies. We perform this task with Algorithm 1, which in our settings performs better than other alternatives.

After all triangles have been processed, the shared memory buffers are copied in coalesced, fully utilized memory transactions to buffers in global memory.

## 3.6 Fluid Simulation

In the following we describe the simulation of fluid effects on surfaces using the closest point method on the GPU. We focus on the numerical solution of two different PDEs describing such effects: the wave equation and the incompressible isothermal Navier-Stokes equations (for details see Section 2.1).

Before a PDE solution can be computed via the closest point method, appropriate initial conditions have to be specified (Section 3.6.1). Then, each solution step generally consists of two phases, both implemented on the GPU via CUDA: In the evolution phase, the solution of the embedding PDE (Section 3.6.2) in the next time step is calculated for all grid vertices in the evolution band. After the evolution, the closest point extension (Section 3.6.3) is applied to all vertices in the extension band. In this phase, the solution is propagated from the surface to the computational band by replacing the values at the grid

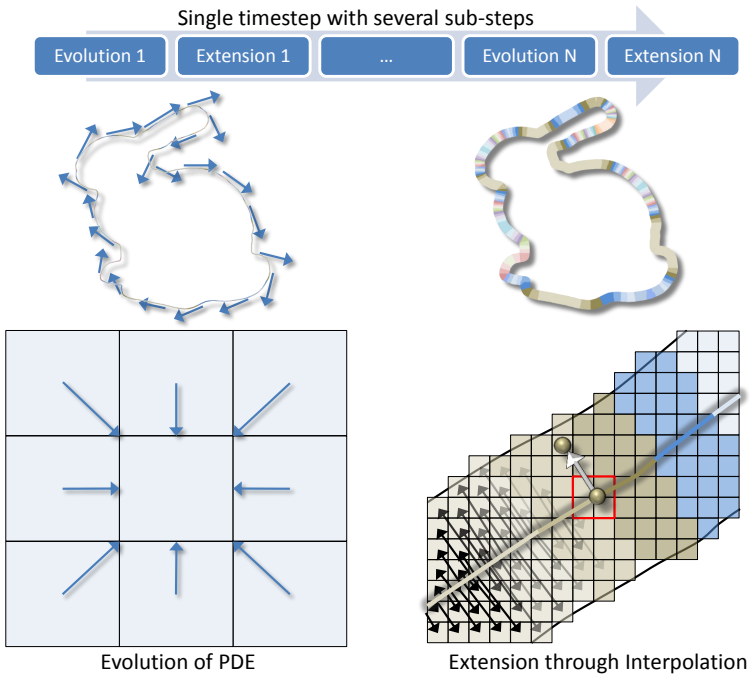


**Figure 3.4**: *A time step may require several sub-steps, consisting of an evolution (left) and an extension phase (right).*

vertices with values interpolated at the closest points on the surface.

For the wave equation (Section 3.6.4), the less involved of the two presented fluid models, just one evolution and one extension is necessary for each time step. In contrast, for the incompressible Navier-Stokes model (Section 3.6.5) a splitting method was used to handle the terms of the equations independently of each other. Consequently, in this model a time step requires several sub-steps and also more than one closest point extension (see Figure 3.4).

### 3.6.1 Initialization

The definition of the computational bands and the construction of the closest point representation (Section 3.5) are necessary to start the closest point method. The initialization of the solution variable completes the initialization phase. In this step, initial data on the surface has to be extended to the simulation grid. When the surface is given as a triangle mesh, the initial data is typically given as per-vertex attributes or as a texture map. Every grid vertex in the extension band interpolates the surface data at the closest point and stores this value at its location in the simulation grid.

### 3.6.2 Evolving the PDE

To update the simulation attributes with the solution for the next time step, we run one CUDA thread per grid vertex of the intersected blocks. If the vertex is not within the evolution band, the thread terminates immediately. For each term of the embedding PDE we choose an adequate evolution strategy (see Sections 3.6.4 and 3.6.5). The embedding of linear terms of the PDE involving spatial differential operators up to order two is found by replacing surface gradients, surface divergences and Laplace–Beltrami operators with their standard Cartesian counterparts in $\mathbb{R}^3$. The discretization can then be done via regular finite difference schemes, as for example described in Section 2.4.4. The appealing property of closest point method in this case is that information is automatically propagated only in tangential directions along the surface. When we have an existing discretization for the problem of choice in $\mathbb{R}^3$, we can reuse it without modification.

The user steers the interactive simulation at runtime, by modifying the simulation attributes temporarily or placing permanent Dirichlet boundary conditions. When this happens, we find the first surface point under the mouse cursor via raycasting (see Section 3.7.1) where we then attach a sphere with a radius equal to the width of the extension band. In every time step, one CUDA thread is started for each grid vertex within one of these spheres in order to replace the respective attribute values.

### 3.6.3 Closest Point Extension

The closest point extension is performed by one CUDA thread per grid point in the extension band. We retrieve the grid point's closest point on the surface and resample the respective attribute at this position using an interpolation scheme. The value at the grid point is then replaced by the interpolated value, thus creating a field that is constant in normal directions to the surface. For values outside the evolution band this can be seen as a natural form of Neumann boundary condition, because gradients in directions normal to the boundary are always zero. The kind of interpolation is important and influences stability, quality and performance of the whole simulation.

A stable interpolation scheme is especially important when dealing with non-smooth surfaces. While the fundamental assumptions of the closest point method are true only for smooth surfaces like the analytic sphere shown in the

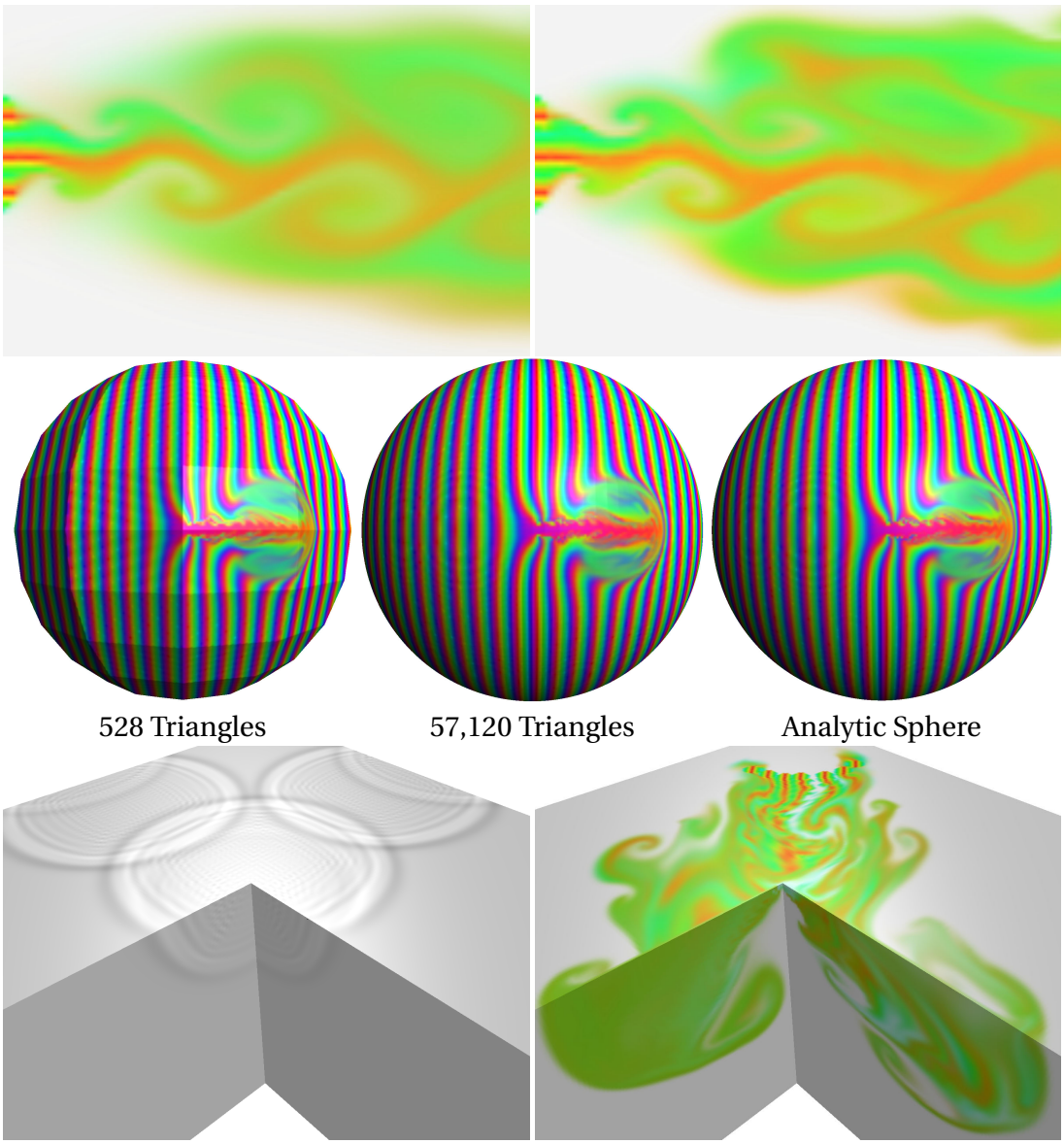


**Figure 3.5**: *Top panel: With linear interpolation (left) instead of WENO4 interpolation (right) the simulation shows artificial diffusion and loses energy. Middle panel: Initialization with a high-quality triangle mesh instead of an analytic surface leads only to subtle differences in the simulation results (grid size:* $256^3$*). Bottom panel: While the closest point method generally requires the surface to be smooth, with clamped WENO interpolation it even gives visually plausible results at sharp creases.*

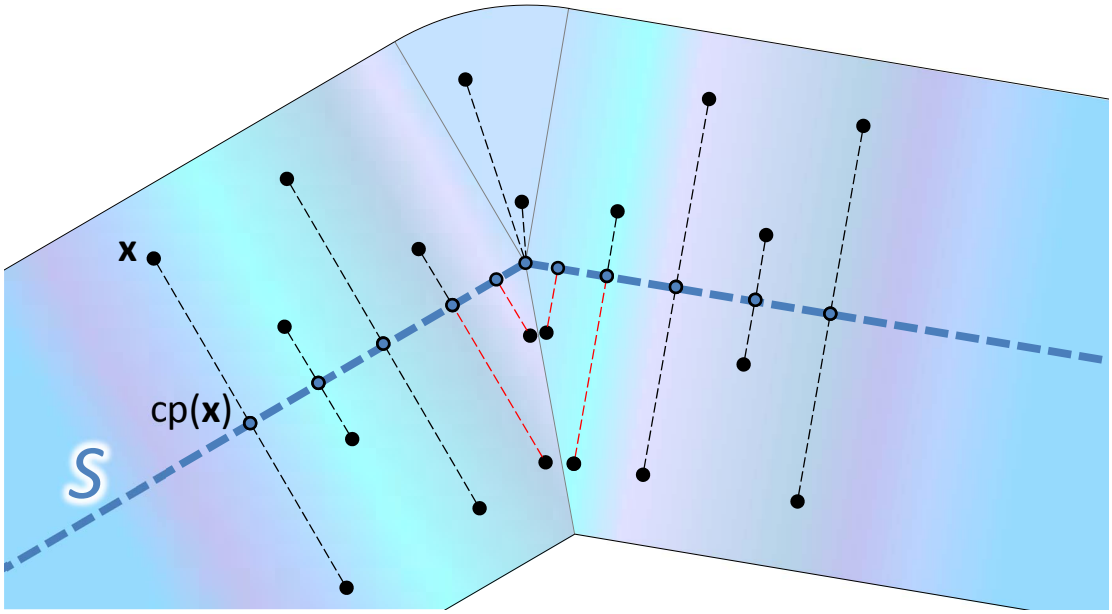


**Figure 3.6**: *At sharp creases the closest point field is discontinuous on the concave side of the surface. This can also lead to discontinuities in the simulation attributes.*

middle panel of Figure 3.5, it is desirable to also handle triangulated surfaces as well as surfaces with sharp features as shown, for instance, in the bottom panel of Figure 3.5. However, the closest point field on the concave side of the surface can contain discontinuities even in the vicinity of such features, and due to the closest point extension this can result in discontinuities in the simulation attributes (see Figure 3.6). This effect is proportional to the distance of the closest points on the two sides of the discontinuity, which itself depends on the sharpness of the feature and the distance from the surface. As a consequence, especially when larger stencils are used the stability of the interpolation scheme becomes important. In general, the same applies to the integration technique used in the evolution phase. Since the proposed integration schemes did not show any stability issues at sharp features, however, special treatment was only necessary in the extension phase.

Linear interpolation is computationally inexpensive and fulfills the convex hull property, which guarantees stability of the interpolation even if the field is discontinuous. Unfortunately, at least for second-order PDE problems, linear interpolation is not accurate enough for the closest point method to ensure consistency [RM08]. At any rate and in practice, linear interpolation introduces a significant amount of artificial numerical diffusion, which manifests

in a loss of energy in physics simulations (cf. Figure 3.5 top panel).

In the work of Macdonald and Ruuth [MR08], a weighted, essentially non-oscillatory (WENO) interpolation appropriate for the closest point method was presented (Section 3.1 and Appendix A of [MR08]). In smooth areas it is of higher order, which prevents numerical diffusion. As with most polynomial interpolations, stability can be critical near discontinuities where the polynomials overshoot the values in the stencil. To reduce this effect, the WENO algorithm calculates a weight for several candidate polynomial interpolants and takes a weighted sum of each interpolant. If a particular candidate is highly oscillatory in a given region it is assigned a very small weight. By this means oscillations are minimized in those areas, as is the formal order-of-accuracy. In the unlikely case that all polynomials oscillate, the WENO interpolation can still overshoot. In our tests, we experienced this effect only for the example shown in Figure 3.5 bottom, near the inner corners. To ensure stability in this event, we restrict (clamp) the interpolation result to the $[u_{smin}, u_{smax}]$ range, with $u_{smin}$ and $u_{smax}$ being the minimum and maximum values in the interpolation stencil.

The desired interpolation order dictates the size of the stencil and thereby the bandwidth requirements. We use the WENO4 interpolation scheme which is built on quadratic candidate interpolants and recovers tri-cubic interpolation in smooth regions (tri-quadratic otherwise) using 64 entries in the stencil. It is sufficiently accurate for the closest point method and we find it to be fast enough for an interactive application.

### 3.6.4 Wave equation

The wave equation is the classical example of a hyperbolic equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla_S^2 \, u$$

In our case $u$ represents the height of an elastic surface over time $t$ for a given wave speed $c$. Note that this is the surface-PDE version of the equation, which means that we can interpret $u$ as the height above some original surface $S$. Because the right-hand side involves only the Laplace-Beltrami operator, we can simply replace it with the Cartesian Laplace operator to retrieve the embedding PDE for the closest point method. We then obtain a discretization for a

3D Cartesian grid with spacing $\Delta \boldsymbol{x}$ and regular time intervals of length $\Delta t$ by substituting the second order derivatives on both sides of the equation with finite differences. For each grid point $(i, j, k)$ this leads to:

$$\frac{\tilde{u}^t_{i,j,k} - 2u^{t-1}_{i,j,k} + u^{t-2}_{i,j,k}}{\Delta t^2} = \frac{c^2}{\Delta \boldsymbol{x}^2}\Big(u^{t-1}_{i+1,j,k} + u^{t-1}_{i-1,j,k} + u^{t-1}_{i,j+1,k} + u^{t-1}_{i,j-1,k} + u^{t-1}_{i,j,k+1} + u^{t-1}_{i,j,k-1} - 6u^{t-1}_{i,j,k}\Big)$$

Solving for the unknown $\tilde{u}^t_{i,j,k}$ leads to the explicit solution

$$\tilde{u}^t_{i,j,k} = \alpha \cdot \Big(u^{t-1}_{i+1,j,k} + u^{t-1}_{i-1,j,k} + u^{t-1}_{i,j+1,k} + u^{t-1}_{i,j-1,k} + u^{t-1}_{i,j,k+1} + u^{t-1}_{i,j,k-1}\Big) + (2 - 6\alpha) \cdot u^{t-1}_{i,j,k} - u^{t-2}_{i,j,k}$$

$$\text{with } \alpha = \frac{\Delta t^2 \cdot c^2}{\Delta \boldsymbol{x}^2},$$

followed by a closest point extension

$$u^t_{i,j,k} := \tilde{u}^t(cp(\boldsymbol{x}_{i,j,k})).$$

Note that this is essentially a Verlet integration [Ver67] since the second derivatives in space as well as in time are approximated with central differences. Compared to an explicit Euler scheme, this method improves the accuracy of the integrator to second order. Since the Verlet integration is stable when the CFL condition is met, i.e., for $\alpha \leq 1/3$ in our case, we always choose this value in all of our experiments. To maximize the accuracy of the simulation, we set the resolution of the computational grid to the highest value which still allows interactive frame rates. Since the integrator requires the height fields of the last two time steps, we copy $u^t$ to $u^{t-1}$ whenever we specify initial conditions or modify the height field during the simulation.

### 3.6.5 Incompressible Navier–Stokes equations

An incompressible, Newtonian fluid with constant temperature can be described by two continuity equations: One for momentum

$$\frac{\partial \boldsymbol{v}}{\partial t} = -(\boldsymbol{v} \cdot \nabla_S)\boldsymbol{v} - \frac{1}{\rho}\nabla_S p + \nu \nabla_S^2 \boldsymbol{v} + \boldsymbol{f}$$

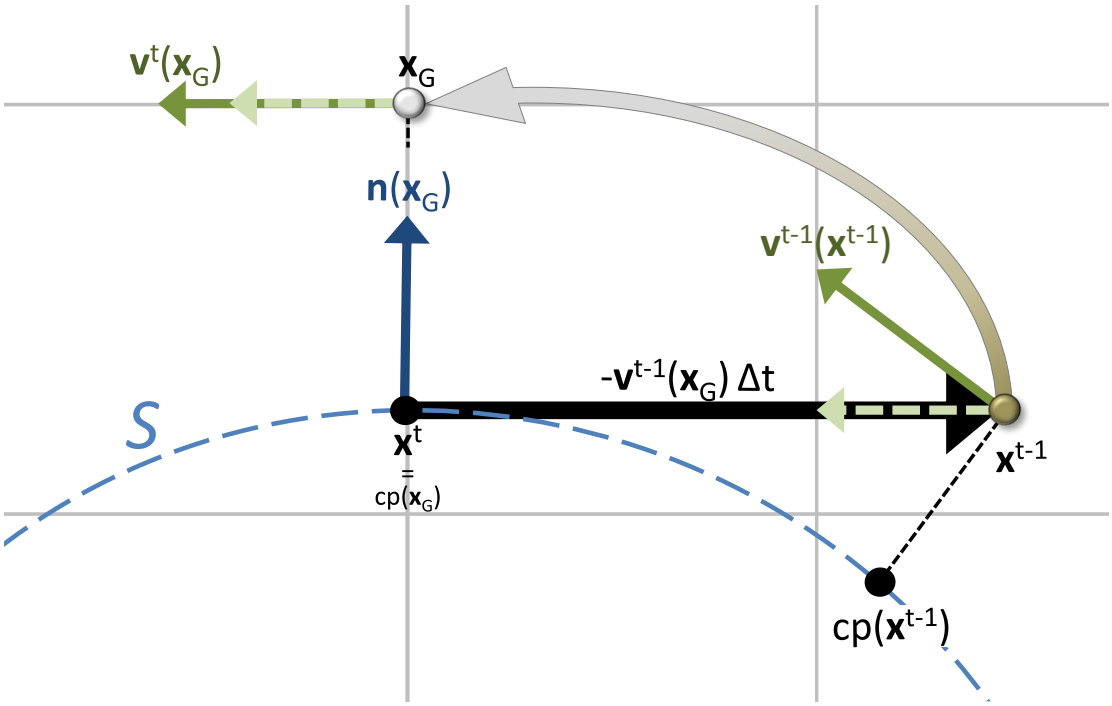


**Figure 3.7**: *Velocity self-advection through a semi-Lagrange back-trace in the simulation grid.*

and one for volume and mass

$$\nabla_S \cdot \boldsymbol{v} = 0.$$

In this variant of the Navier–Stokes equations—formulated as surface PDE—$\boldsymbol{v}$ denotes the fluid's velocity, $\rho$ the constant density, $p$ the pressure and $\nu$ the constant kinematic viscosity. Consequentially, the terms in the first equation represent accelerations resulting from self-advection $(-(\boldsymbol{v}\cdot\nabla_S)\boldsymbol{v})$, pressure $\left(-\frac{1}{\rho}\nabla_S\, p\right)$, viscosity $\left(\nu\nabla_S^2\,\boldsymbol{v}\right)$ and external forces $\left(\boldsymbol{f}\right)$, while the second equation states that the velocity field must be free of divergence. Instead of interpreting this flow problem as conservation laws with constraints, we split up the equations and treat each term individually.

The last two terms of the momentum equation are unrepresented in our solver, since highly viscous flows and external forces are currently not considered. If low order interpolation is used in the extension phase, however, the flow already exhibits some artificial viscosity due to numerical diffusion. We therefore also disregard fluids which are completely inviscid. On the other hand, viscous flows can be integrated either explicitly (with a potentially re-

stricted time-step) or using an implicit closest point method [MR09], and external forces can be added in a straightforward way.

For the nonlinear convective acceleration $-(\boldsymbol{v}\cdot\nabla_S)\boldsymbol{v}$, we use a semi-Lagrangian back-trace as, e.g., described by Stam [Sta99]. By this means a computationally more expensive implicit solution can be avoided. To update the value of $\boldsymbol{v}$ at each computational grid vertex, the back-trace follows the motion of a particle starting at the vertex backwards in time through the velocity field to find the previous location of the particle. At this position, the old velocity field is sampled and the retrieved value replaces $\boldsymbol{v}$ at the grid vertex.

This back-trace could naïvely be implemented in a full simulation grid using its 3D equivalent. In this case, only one change is necessary in order to discretize the embedding of the surface-advection with the back-trace: as the velocity vector must stay in the tangent space also at its new position, it must be projected onto the local tangent plane.

For the sparse grid discussed here, however, it must be ensured that such a back-trace does not leave the computational band. This results in additional constraints on the maximal velocity, the integration time step, and the minimal size of the extension band. To minimize these limitations we perform the back-trace only for points on the surface. Figure 3.7 illustrates this method.

The back-trace starts at the closest point of the current grid vertex $cp(\boldsymbol{x}_G)$ and uses a single step of the explicit Euler method. Because the current velocity values are a closest point extension, they are constant in the normal direction and for the start of the back-trace we can directly use the velocity value stored at the grid vertex $\boldsymbol{x}_G$. At the old particle location $\boldsymbol{x}^{t-1}$ we sample the velocity field using an interpolation. To avoid numerical diffusion due to the interpolation in the back-trace, we use an interpolation of higher order like the one we described in the context of the closest point extension. The interpolated velocity vector from the particle's old position is projected onto the tangent plane at the new position and then rescaled to its old length in order to retrieve the updated value for the grid vertex.

To obtain the surface normal necessary for such a projection at a grid point $\boldsymbol{x}_G$, one can normalize the difference vector $\boldsymbol{d} = \boldsymbol{x}_G - cp(\boldsymbol{x}_G)$ from the closest surface point to the point itself. These normals always point away from the surface on both of its sides. For non-orientable surfaces, for which both directions are equally valid, this behavior is thoroughly intended. A real problem with this approach, however, is that it fails for grid points very close to or on

the surface. To reliably obtain a normal as well for these points, we therefore examine the grid points $\boldsymbol{x}_i$, $i = 1, 2, \ldots, n$ in a small stencil around $\boldsymbol{x}_G$ and calculate a vector $\boldsymbol{d}_i = \boldsymbol{x}_i - cp(\boldsymbol{x}_i)$ for each of them. We accumulate the differences in the vector $\boldsymbol{d}_S^n$ as follows:

$$\begin{aligned} \boldsymbol{d}_S^1 &= \boldsymbol{d}_1 \\ \boldsymbol{d}_S^i &= \boldsymbol{d}_S^{i-1} + \begin{cases} \boldsymbol{d}_i & \text{if } \boldsymbol{d}_i \cdot \boldsymbol{d}_S^{i-1} \geq 0 \\ -\boldsymbol{d}_i & \text{otherwise} \end{cases} \end{aligned}$$

The negation is necessary to account for grid points on different sides of the surface. A surface normal is then retrieved by normalization of $\boldsymbol{d}_S$. Note that the resulting normal points to the side of the surface on which the first off-surface vertex in the stencil lies. In practice, we find using the 6 edge adjacent points around $\boldsymbol{x}_G$ works well.

In fluid mechanics, the pressure term $-\frac{1}{\rho}\nabla_S\, p$ is often used within a projection method [CM00] to ensure that the resulting velocity field fulfills the incompressibility equation $\nabla_S \cdot \boldsymbol{v} = 0$. Conceptually, this is achieved by applying a Hodge decomposition to the velocity field, which splits it into a divergence-free field and a curl-free field (for details see Section 2.2.1).

We determine a $p$ that lets $\frac{1}{\rho}\nabla_S\, p$ reproduce the curl-free part, such that after the subtraction only the divergence-free part is left. At every time step, this ultimately requires calculating the divergence of the current velocity and solving a surface Poisson problem of the form $\nabla_S \cdot \nabla_S p = \nabla_S \cdot \boldsymbol{v}$ to retrieve the desired pressure. Treating this Poisson equation with an artificial time iteration and the explicit closest point method would require a closest point extension after each step, which makes this approach quite costly. Another option would be applying the implicit closest point method [MR09], either to an artificial time iteration or directly to the Poisson problem. This would require solving a large linear system in each time step.

We choose a computationally less expensive approach where we apply the closest point extension only to the right-hand side of the equation, and we replace the left-hand side with the Cartesian Laplacian. Then, we solve the linear system with the conjugate gradients method [KW03]. The solver considers only grid points within the evolution band and uses a Neumann boundary condition for the values outside (i.e., the gradient in normal direction to the boundary is zero). We chose a Neumann condition because it resembles the

effect of the closest point extension within an artificial closest point method time-integration [Gre06]. While this may not enforce incompressibility to a high order of accuracy, in practice it gives very good results if we reuse the pressure from the last time step as the initial guess. After the pressure update, a closest point extension must also be applied to the velocity field itself to prepare it for the next time step. It is worth noting here, that the Neumann boundary condition is only necessary to resemble the behavior of the closest point method at the boundary of the evaluation band. It may not be confused with the typical CFD boundary conditions used to simulate object boundaries or in-/out-flow conditions.

In order to visualize the fluid flow we introduce an additional, final substep in which the advection of a mass-less, colored dye through the velocity field is simulated. The dye is transported through the flow by the same advection operator $\boldsymbol{v} \cdot \nabla_S$ that is used for the velocity. The only difference is that the dye is represented by three scalar fields—one for each $r\,g\,b$ color channel—as such the operator does not include a projection onto the tangent plane.

## 3.7 Rendering

For rendering the simulated fluid effects on the surface we use Direct3D 10, and we make use of CUDA's interoperability functions to access the respective GPU memory resources within both APIs. If the surface is given as a triangle mesh and the simulated attributes are used to modulate its color, the mesh is rendered via polygon rasterization. The coordinates of the triangle vertices in the embedding 3D domain are interpolated by the rasterizer and then a pixel shader is employed to retrieve values from the simulation buffers using trilinear interpolation. Figure 3.8 (left) demonstrates such a rendering with flat shading to show the triangulation, and in Figure 3.9 (top panel) Phong shading was used to realize a smooth look of the surface.

### 3.7.1 Volume Raycasting

If the initial surface is not given as a polygon mesh, for instance if it is given by a point set representation or an implicit surface description, or if the simulated attributes should be used to modulate the geometry of the surface, rasterization cannot be used any more. To overcome this problem, we present a GPU

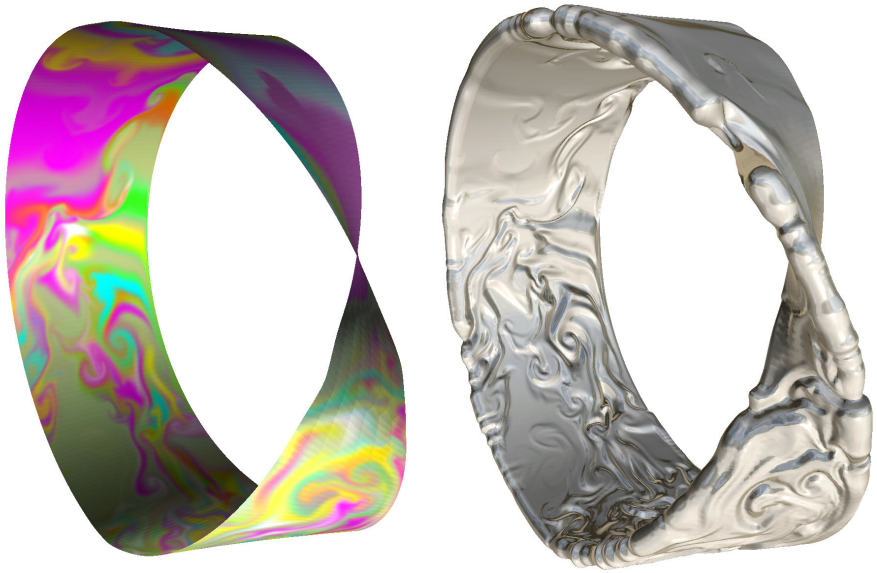

**Figure 3.8**: *Navier-Stokes simulation on a Möbius strip; a non-orientable manifold. Left: Rendering with rasterization and flat shading to emphasize the triangle mesh serving as the original surface description. Right: Same simulation as left, but now rendered via raycasting and using the red ink as a displacement on "both sides" of the surface.*

method that renders the surface directly from the sparse closest point volume representation, regardless of the initial surface representation.

A first approach is to perform exact voxel ray-casting of the closest point volume in a DDA-like fashion [AW87]. The voxels correspond to the closest point cells, and the constant voxel attributes are given by the simulation values at the cell vertices. The rays of sight first traverse the grid of blocks until a marked block is hit. Then, the block's location in the *closest-point-buffer* is retrieved and traversal is continued on the cells inside. Traversal is stopped if a voxel is hit that contains the closest surface point stored at this voxel, and the voxel attribute is used as pixel color.

While the proposed technique can visualize a piecewise (per voxel) constant distribution of attributes across a surface, it does not allow rendering a smooth distribution. This is achieved by interpolating the distances of the cell vertices to their closest surface points and performing an exact intersection test between the ray and its zero crossings in the reconstructed field.

For this purpose we sample along the ray in regular intervals of half the voxel width. When a hit is detected, we refine the result using binary search. Since the closest point method is based solely on the existence of an *unsigned*

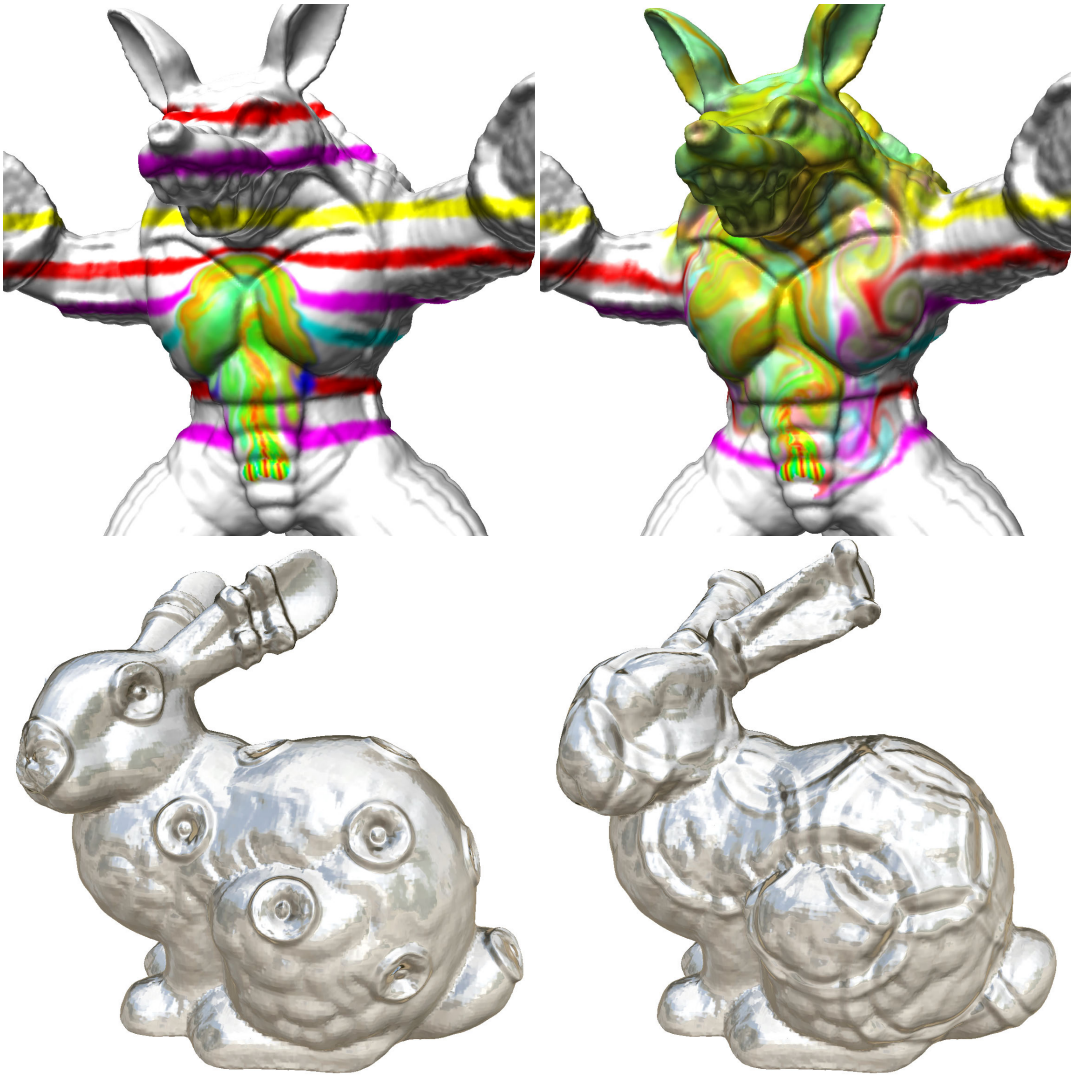

**Figure 3.9**: *Numerical simulation of fluids on complicated surfaces via the closest point method. Top panel: The simulation result is rendered via rasterization and pixel shading. Bottom panel: Raycasting the embedding computational grid allows simulating surface displacements. At a resolution that corresponds to a* $512^3$ *Cartesian grid, simulation and rendering takes less than 160 ms per time step.*

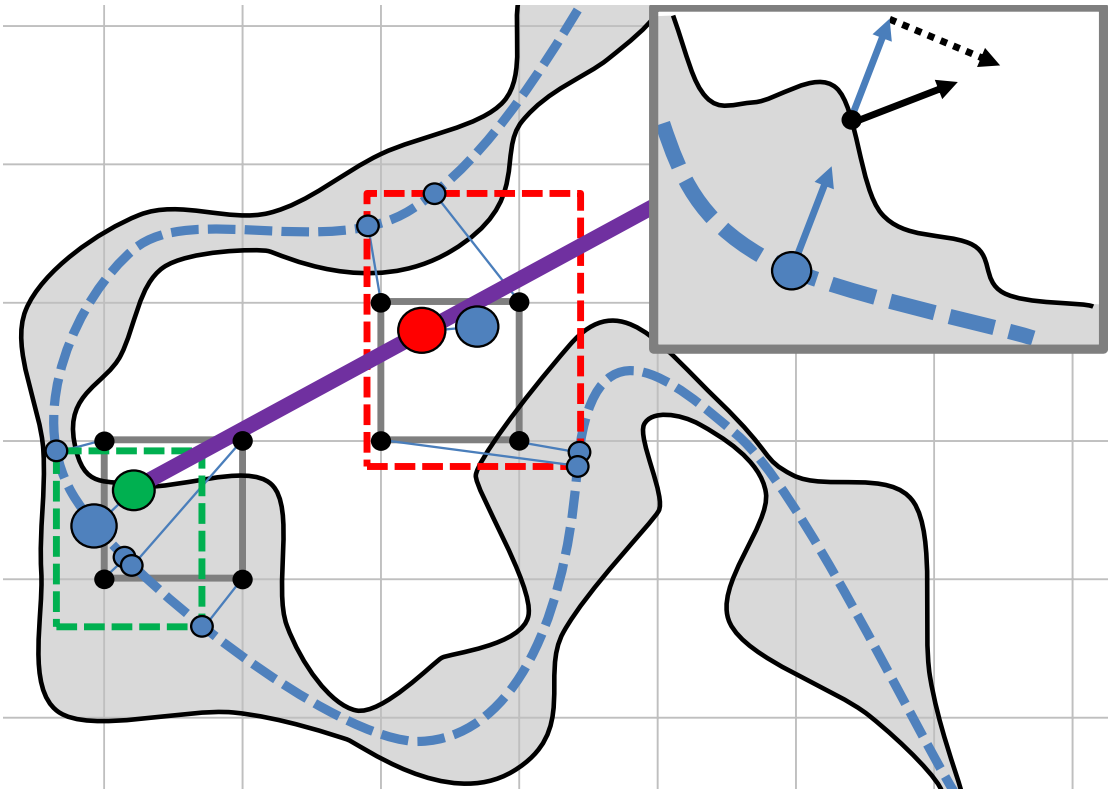

**Figure 3.10**: *Bottom left: Intersection of a ray (purple) with a displaced surface (black). At the red ray sample, the interpolation of closest points on the original surface (blue) results in an off-surface point. At the green ray sample, an intersection with the displaced surface was found, as the distance to the closest point equals the interpolated height value. Top right: The normal of the original surface (blue arrows) combined with the gradient of the height value (dotted arrow) leads to the normal of the displaced surface (black arrow).*

distance transform, the zero crossings cannot be calculated exactly by this means, and the intersection test comes down to finding the points along the rays where the distance between the sample position and the trilinearly interpolated closest point is below a given $\epsilon$. At this intersection point, the simulation attributes are retrieved via trilinear interpolation, too. To simulate smooth shading, the attributes are translated into a material color and per-pixel normals are calculated by normalizing the vector between the intersection point and the closest surface point.

### 3.7.2 Rendering Surface Displacements

In addition to using the closest point representation for rendering the original surface, we can also use it for simulating surface displacements that are given, for instance, by the simulation values at the grid vertices (see Figures 3.8 right, and 3.9 bottom). After interpolating the closest point $cp(\boldsymbol{x}_R)$ at a sample position $\boldsymbol{x}_R$ on a ray, a scalar simulation value $h\big(cp(\boldsymbol{x}_R)\big)$ at this point is interpolated and compared to the distance between $cp(\boldsymbol{x}_R)$ and $\boldsymbol{x}_R$. As Figure 3.10 shows, a hit is detected if the distance value is smaller than the interpolated attribute value. The figure also shows, that the interpolation of closest points from different parts of the surface can lead to off-surface points. To avoid rendering artifacts in this case, we determine the maximum extent of an axis aligned bounding box around the closest points of the interpolation stencil for the current ray sample. If this bounding box is considerably larger than the interpolation cell, we skip the current sample and continue sampling along the ray. The normal of the displaced surface is calculated from the original surface normal and the gradient of the attribute field. Since the gradient is tangent to the surface because of the closest point method, it can be used in turn to disturb the surface normal appropriately. The maximum displacement is supposed not to exceed the width of the computational band. This constraint can be abandoned, however, by using an enlarged displacement band, where the grid vertices outside the computational band store only their closest points but no attributes.

The proposed rendering approach displaces the surface equally on both of its sides. For orientable surfaces it is also possible to distinguish the two halfspaces in order to displace in just one direction or to use negative displacements. For non-orientable surfaces the view-direction can be used to introduce an artificial orientation. Despite being view dependent, this approach has the additional problem that it results in artifacts at the silhouettes, where the orientation flips. We therefore recommend to treat both sides equally in this case.

## 3.8 Performance Analysis

To validate the efficiency of the proposed method for simulating and rendering fluid effects on surfaces, we have performed a number of experiments us-

**Table 3.1**: *Performance statistics for fluid simulation and rendering on the GPU via the closest point method.*

| Mesh | Res. | # Closest Points | Grid Generation GPU Memory | Time | Wave Equation linear | Wave Equation WENO4 | Navier-Stokes linear | Navier-Stokes WENO4 | Rendering Rasterization | Rendering Raycasting | Displ. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Möbius | $64^3$ | 34k / 48k | 1MiB / 4MiB | 6.1ms / 9.5ms | 2.0ms | 2.1ms | 3.7ms | 20ms | 1.3ms | 3.8ms | 20ms |
| (Figure 3.8) | $128^3$ | 77k / 184k | 4MiB / 18MiB | 6.4ms / 10ms | 2.1ms | 4.1ms | 7.5ms | 65ms | 1.3ms | 4.3ms | 23ms |
| 3,385$\Delta$ | $256^3$ | 303k / 721k | 18MiB / 73MiB | 7.4ms / 12ms | 2.3ms | 15ms | 23ms | 237ms | 1.4ms | 5.5ms | 31ms |
| | $512^3$ | 1.2M / 2.9M | 82MiB / 297MiB | 12ms / 21ms | 5.1ms | 54ms | 85ms | 909ms | 1.7ms | 8.4ms | 48ms |
| Bunny | $64^3$ | 41k / 80k | 2MiB / 7MiB | 29ms / 84ms | 2.2ms | 3.0ms | 4.0ms | 28ms | 1.4ms | 3.3ms | 15ms |
| (Figure 3.9) | $128^3$ | 177k / 374k | 9MiB / 35MiB | 32ms / 86ms | 2.4ms | 9.0ms | 13ms | 118ms | 1.4ms | 4.0ms | 24ms |
| 69,451$\Delta$ | $256^3$ | 733k / 1.6M | 41MiB / 159MiB | 34ms / 90ms | 3.6ms | 34ms | 51ms | 529ms | 1.4ms | 5.7ms | 35ms |
| | $512^3$ | 3.0M / 6.8M | 178MiB / 679MiB | 49ms / 117ms | 11ms | 136ms | 222ms | 2,174ms | 1.7ms | 8.6ms | 52ms |
| Armadillo | $64^3$ | 28k / 58k | 1MiB / 5MiB | 117ms / 377ms | 2.0ms | 2.7ms | 3.4ms | 18ms | 2.5ms | 3.0ms | 16ms |
| (Figure 3.9) | $128^3$ | 123k / 258k | 6MiB / 25MiB | 118ms / 380ms | 2.1ms | 6.1ms | 9.5ms | 80ms | 2.5ms | 3.7ms | 20ms |
| 345,944$\Delta$ | $256^3$ | 513k / 1.1M | 29MiB / 110MiB | 120ms / 384ms | 2.9ms | 25ms | 38ms | 370ms | 2.5ms | 4.6ms | 28ms |
| | $512^3$ | 2.0M / 4.8M | 129MiB / 477MiB | 135ms / 404ms | 8.6ms | 100ms | 157ms | 1,667ms | 2.6ms | 5.9ms | 45ms |

ing surface models and simulation grids at different resolutions. In all of our experiments, a block size of $b = 4$ was used. We found this size to give the best performance compared to smaller (tighter fit of the adaptive simulation grid to the surface but increasing number of indirections to access adjacent grid cells) and larger (increasing memory and computation requirements) blocks. All measurements were performed on a 2.4 GHz Core 2 Duo processor and an NVIDIA GeForce GTX 480 graphics card with 1,536 MiB local video memory. A detailed memory and performance statistic of the GPU closest point method for fluid simulation is given in Table 3.1. Here, numbers separated by a slash refer to the simulation using the wave equation with linear interpolation and the Navier-Stokes equations with WENO4 interpolation, respectively. All timings are given in milliseconds and rendering was always performed on a $1,024 \times 1,024$ viewport.

For each model, the first column lists the name and number of triangles, while the second column gives the resolution of the uniform Cartesian grid to which the simulation resolution corresponds. The third column lists the number of closest points within the computational band. The differences are due to differently sized computational bands that are dictated by the respective numerical stencils of the wave equation (bandwidth with linear interpolation: $2.4\Delta\boldsymbol{x}$) and the Navier-Stokes equations (bandwidth with WENO4 interpolation: $5.7\Delta\boldsymbol{x}$). The forth column gives the memory that is required for solving the equations on the GPU via the closest point method. The higher memory consumption of the Navier-Stokes simulation is due to the larger computational band and the additional buffers that are required to store the simulation attributes. The times required to build the closest point representation are given in the fifth column. Here it is important to note that more than 75% of the time is always required by the closest points computation, meaning that the grid construction on the GPU consumes only a small portion of this time. It can further be seen that the GPU memory requirements mainly depend on the resolution of the simulation grid, while on the other hand, the time for constructing the grid is strongly dependent on the number of triangles.

The following four columns list the simulation times using linear and WENO4 interpolation in the closest point extension for the wave equation and the Navier-Stokes equations. As expected, WENO4 interpolation increases the overall simulation times significantly due to its larger computational stencil, and the enlarged computational band thereof. The Navier-Stokes simulation

uses the respective interpolation type in three separate closest point extensions (divergence, velocity, ink) and two advection passes (velocity, ink). It can be observed that solving the pressure Poisson equation is the most expensive operation (~75% of the total time) as long as linear interpolation is used. With WENO4 interpolation, however, these ratios turn into the opposite: Most of the time is now spent on interpolations in the advection (~50%) and extension kernels (~40%).

The last three columns show the rendering times for rasterization, voxel-based surface raycasting, and raycasting with surface displacements. The timings refer to the rendering of a WENO4 wave simulation. In the last column a computational band of width $8\Delta\boldsymbol{x}$ was used to allow simulating large surface displacements. As expected, rasterizing the triangle mesh on the GPU leads to the highest frame rates. Direct surface raycasting is between 1.2 to 5 times slower, but still delivers highly interactive frame rates. Raycasting with smooth displacements, which allows simulating dynamic surface modifications, also delivers interactive frame rates even for the computational grids with the highest resolution. This is quite remarkable since much larger closest point and simulation buffers are used.

### 3.8.1 Surface Deformations

The timing statistics indicate that the creation of an adaptive closest point grid for the CPM is possible at high rates even for large triangular meshes. This is an important step towards a real-time CPM for simulating deformations of the surface itself. In the following we demonstrate mesh smoothing based on the Laplace operator [Tau95] via the GPU CPM as a first example (see Fig. 3.11).

Each smoothing step consists of three operations: Firstly, we create a discrete, adaptive closest point field as described. Secondly, we apply the Laplace-Beltrami operator—discretized with the standard Laplace operator on the embedding Cartesian grid—to the closest point field. At every point on the surface the resulting vector field is directed towards the mean of the point's neighborhood on the surface. Thirdly, we interpolate the discrete field using WENO4 at every vertex position of the mesh and move each vertex into this direction.

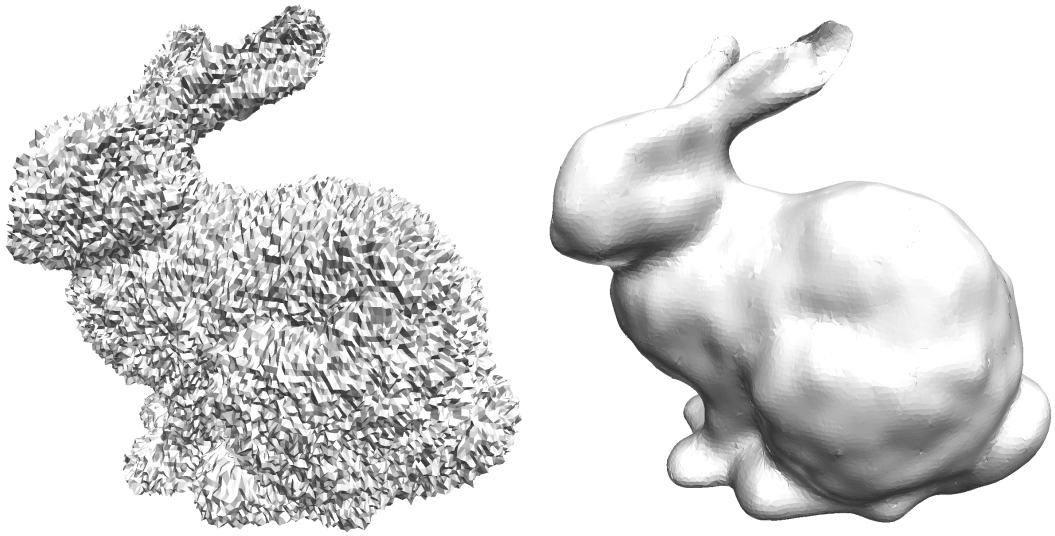

**Figure 3.11**: *Laplacian mesh smoothing using the closest point method. Left: Distorted Bunny model. Right: The model after 15 smoothing steps (64ms each) using a grid resolution of* $256^3$*. Both images show triangle rasterization and flat shading.*

## 3.9 Conclusion

In this chapter, we have presented a real-time method for simulating fluid effects on surfaces using the closest point method. To achieve this, we have developed a fast GPU method for realizing the closest point method and rendering the simulation results as surface colors or displacements. The method can efficiently solve non-linear PDEs on surfaces using a high-resolution embedding grid, and it can calculate and render the resulting fields independently of the resolution and topology of the input geometry.

In the presented examples the closest point method is solely used for solving PDEs on (infinitely thin) surfaces, without enforcing any problem-specific physical conditions. Even though this would be possible in special cases, it has not been considered in this work. Consequently, the effects demonstrated in this work do not show real fluid effects as they would appear in reality, but rather demonstrate the potential of the closest point method for producing visually plausible and compelling fluid effects on surfaces in real-time. It is interesting to note, on the other hand, that the exact same type of PDEs may arise at the boundaries of volumetric simulations, and coupling oddly-shaped boundaries to full volumetric simulations opens an interesting venue for future work.

The proposed methods open a number of future research directions. From a numerical point of view we are most interested in exploring how multigrid schemes can be employed to speed up the simulation on the embedding grid. This requires pursuing research on the construction of a multigrid closest point representation that can accurately approximate the surface at ever coarser scales. By integrating such a construction into approaches that can create a coarse grid hierarchy independently of the complexity of the object's shape [LPR+09, DGW11], good approximation quality and convergence rates can be expected even for complicated surfaces.

# 4

# A Semi-Lagrangian Closest Point Method for Deforming Surfaces

In this chapter we present a Eulerian method for the real-time simulation of intrinsic fluid dynamics on deforming surfaces. To compute and visualize flows and wave propagations along animated triangle meshes at high resolution, we propose a novel semi-Lagrangian closest point method for the solution of PDEs on arbitrarily deforming surfaces and demonstrate the efficient conversation of animated triangle meshes into a time-dependent implicit representation based on closest points. The proposed technique is unconditionally stable with respect to the surface deformation and, in contrast to comparable Lagrangian techniques, its precision does not depend on the level of detail of the triangulation.

## 4.1 Introduction

While most existing techniques for the solution of PDEs on surfaces consider only static surfaces, several authors demonstrated recently that there are even more applications for PDEs on deforming surfaces, especially in the context of fluid simulation. Some physical phenomena, like capillary waves on the fluid interface, inherently demand a special treatment of the surface which goes beyond a mere tracking [TWGT10]. Additionally, a restriction to the surface often makes sense even for effects which can be captured by a 3D simulation, simply because the problem size is reduced considerably. This is an important advantage for the "up-resing" of existing coarse simulation results [KTT13], as

well as for interactive simulations [ATBG08].

Considering a moving and deforming simulation domain, it is worthwhile to classify the available algorithms according to the time dependence of the employed discretization. On one side of the spectrum there are the purely Eulerian methods, based on a sampling that remains fixed in the three-dimensional embedding space. These techniques often employ implicit surface representations and provide a very natural coupling to fully three-dimensional Eulerian simulations [KTT13]. On the other side there are the purely Lagrangian methods, which employ a dynamic sampling that follows the surface movement *and* the evolution of the simulated properties on the surface at the same time. Most existing techniques are hybrids, however, which employ, for example, a Eulerian fluid simulation on a Lagrangian grid following the surface [ATBG08, TWGT10]. The algorithm we propose utilizes semi-Lagrangian discretization schemes and a Lagrangian representation of the surface deformation. We still consider it a Eulerian technique, however, since it carries out the simulation on a purely Eulerian grid. To the best of our knowledge, our algorithm is the first real-time Eulerian method for the solution of PDEs involving intrinsic differential operators on deforming surfaces.

We further develop the techniques presented in Chapter 3 to a semi-Lagrangian closest point method (SLCPM) for the interactive simulation of fluid effects on *deforming* surfaces. Our method focuses on the efficient coupling of the closest point method with existing animations of triangulated surfaces.

Compared to its most closely related interactive alternative [ATBG08], the SLCPM has several advantages. The addition of a Eulerian embedding grid and an implicit closest point surface representation makes the simulation agnostic to the topology, level of detail and other quality features of the input triangulation. This means in particular that it automatically adapts to deformations which alter the size of the triangles or the extrinsic topology of the surface. In contrast to other techniques which apply the closest point method to deforming surfaces [HZQW10, KTT13], our technique integrates the surface deformation mechanism directly into the closest point method. We therefore do not require an additional level set representation of the surface or an external velocity field, and we cut down on the number of interpolation steps to reduce numerical dissipation.

This chapter is structured as follows: In Section 4.2 we review previous work that is related to ours. The beginning of Section 4.3 motivates the integra-

tion of the semi-Lagrangian scheme into the closest point method and gives a high-level overview of our approach. In Section 4.3.1 we recap the basis techniques and explain the aspects relevant to their combination. Section 4.3.2 describes the required implicit representation of the deforming surface. In Sections 4.4 and 4.5 we conclude the chapter with a presentation of the results and an outlook to future work.

## 4.2 Related Work

Several authors have simulated fluid-related phenomena on deforming surfaces. The majority of the presented techniques uses an unstructured tessellation of the surface as a Lagrangian spatial discretization. Bargteil et al. [BGOS06] developed a method for texturing fluid surfaces which contours and reinitializes a zero level set in each time step. They advect the triangle mesh in the velocity field of the basis fluid simulation which thereby transports texture coordinates or the attributes of a reaction-diffusion simulation [Tur91], stored at the vertices. Their simulation is very sensitive to the motion of the surface, however, so that even small perturbations can cause large changes in the resulting surface texture. Angst et al. [ATBG08] discretized the wave equation on an animated character mesh based on a mixed finite volume / finite element method. Their interactive simulation is very robust because of the implicit time integration scheme and the fixed topology of the Lagrangian discretization grid. The computation of an additional well conditioned triangulation at runtime is avoided, hence the discretization does not adapt to deformations affecting the triangle sizes. Thürey et al. [TWGT10] also discretized the wave equation on a Lagrangian mesh in order to simulate capillary waves driven by surface tension. Their simulation employs a finite element method similar to the one of Angst et al. [ATBG08], yet it transforms the surface tessellation into a height field on a lower resolution simulation grid in each time step of the basis simulation. The numerical analysis of all these methods based on Lagrangian surface meshes is quite difficult, because deforming and unstructured computational grids in combination with approximative differential surface characteristics lead to non-trivial discretizations of the intrinsic differential operators [BCOS01, RM08].

An alternative are the embedding techniques discussed in Section 2.3 (page 25). In contrast to the parametrization-based and mesh-based techniques,

these methods usually employ a Eulerian spatial discretization, often in the form of a regular 3D grid [BCOS01, Bur05, Gre06, RM08]. This simplifies the numerical analysis considerably, especially because the discretization is not affected by the deformation of the surface. Like with all Eulerian techniques, the explicit handling of advection requires special care, however, especially when movement and deformation of the surface are considered. Among the embedding techniques, the closest point method [RM08, MR08, MR09, MM12] stands out for its simplicity. It employs the unmodified three-dimensional Cartesian counterparts of the intrinsic surface differential operators in the embedding PDE and it restricts the calculations to a narrow band around the surface without enforcing any non-physical boundary conditions. Hong et al. [HZQW10] apply the closest point method to the level set equation in order to simulate the spreading of fire on an animated surface. Since they treat also the movement of the surface itself as a level set evolution, they handle only advections in directions normal to the surface. Kim et al. [KTT13] suggest to increase the apparent resolution of an already acquired liquid simulation by simulating additional wave propagation on the surface via the closest point method. They use a level-set to reinitialize the closest point function and an extended velocity field, generated by the low-resolution volumetric simulation, to advect the simulation attributes with the deforming surface.

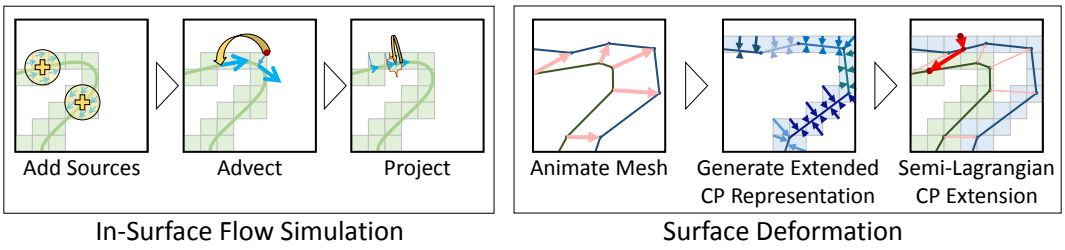


**Figure 4.1**: *An overview of the steps of our flow simulation. The semi-Lagrangian closest point extension step advects the flow field with the deforming surface.*

## 4.3 A Semi-Lagrangian Closest Point Method

In this section we describe an algorithm that extends the simulation techniques presented in Chapter 3 to deformations of the surface, given as an animation of the input triangle mesh.

In existing applications of the explicit closest point method to deforming surfaces, the advection of simulation attributes with the surface is handled in an additional, decoupled step. Either level-set advection [HZQW10] or MacCormack advection in an external velocity field [KTT13] are employed. The first approach ignores tangential surface movements, whereas the second one is computationally too expensive for interactive applications, especially if the velocity field is not already available. We instead propose to integrate this advection step directly into the closest point method itself.

Our algorithm is based on the observation that the explicit closest point method and the backward semi-Lagrangian method exhibit remarkable similarities. The former finds the closest point of a computational node on the surface, whereas the later integrates the position of a node backward in time to find the foot of a characteristic curve in a velocity field. Both methods obtain updated nodal values through interpolation at the found position. We propose to combine the two methods by first finding the closest point on the surface and then integrating this position backward in time along the characteristics defined by the motion and deformation of the surface. Combining the closest point extension with a semi-Lagrangian backtrace in a single step has three main advantages: First, the approach is unconditionally stable with respect to the surface deformation, which is very important when the input animation is out of our control. Second, we reduce the number of spatial interpolations which are computationally expensive and introduce numerical

dissipation (see Section 3.8 on page 72 and [KTT13]). Third, if the backtrace starts at a closest point on a triangulated surface, the foot of the trajectory can be found very efficiently by considering only the movement and deformation of an individual triangle.

Figure 4.1 gives an overview of our algorithm for the interactive flow simulation on deforming surfaces based on these ideas. The steps in the left panel solve the incompressible Navier-Stokes equations

$$\frac{\partial \boldsymbol{v}}{\partial t} = -(\boldsymbol{v} \cdot \nabla_S)\,\boldsymbol{v} - \frac{1}{\rho}\nabla_S\, p + \nu \nabla_S^2\, \boldsymbol{v} + \boldsymbol{f} \quad \text{and}$$

$$\nabla_S \cdot \boldsymbol{v} = 0$$

on the surface. They combine an adaptive multiblock closest point method with a standard splitting technique as described in Section 3.6 (page 58). In the first step, velocity sources are injected by applying Dirichlet boundary conditions in user-defined regions on the surface. We advect these regions together with the surface, using a technique similar to the one employed for the simulation attributes. The second step addresses the convective acceleration term $(\boldsymbol{v} \cdot \nabla_S)\,\boldsymbol{v}$ by performing first-order semi-Lagrangian advection along the surface, followed by a tangential projection of the velocities. We skip the viscosity term $\nu \nabla_S^2\, \boldsymbol{v}$ because we do not intend to simulate highly viscous fluids and because a small amount of numerical dissipation is already introduced by the interpolation steps. The artificial viscosity is reduced considerably if we apply high-order stability-preserving WENO4 interpolation, as remarked in Section 3.6 and [KTT13]. We also do not use the external forces term $\boldsymbol{f}$ in our current examples, in order to demonstrate the pure in-surface evolution of the intrinsic flow. If desired, one could use this term to integrate extrinsic effects such as inertia, as suggested by Angst et al. [ATBG08]. To enforce the continuity equation, we use the pressure term $\frac{1}{\rho}\nabla_S\, p$ of the momentum equation to project the velocity field onto a space of solenoidal functions. This third step utilizes a conjugate gradient method and an artificial Neumann boundary condition to solve a Poisson problem of the form $\nabla \cdot \nabla p\,(\boldsymbol{x}) = \nabla \cdot \tilde{\boldsymbol{v}}\left(cp\,(\boldsymbol{x})\right)$, in which the right-hand side is the closest point extension of the divergence. After the pressure correction, the original technique obtains a closest point extension of the velocity field. We replace this final sub-step with a *semi-Lagrangian* closest point extension which additionally advects the velocity

field together with the deforming surface, as detailed in Section 4.3.1.

As an alternative to the intrinsic flow simulation, we also consider the simulation of intrinsic waves which propagate along a deforming surface membrane. Therefore we replace the velocity field with a height field and instead of the Navier-Stokes equations we solve the surface-PDE version of the classic wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla_S^2 \, u.$$

As described in Section 3.6, an embedding PDE is obtained by replacing the Laplace-Beltrami operator on the right-hand side with the standard Cartesian Laplace operator and an explicit Verlet integration scheme is employed as discretization method. In contrast to the original algorithm, however, we again obtain a *semi-Lagrangian* closest point extension instead of a standard closest point extension at the end of each time integration step.

Before a semi-Lagrangian extension can be applied in one of our simulations, however, we must first deform the triangulated surface and update its implicit representation, as depicted in the right panel of Figure 4.1. Our algorithm considers the animation of the triangle mesh, i.e. the update of the vertex positions, to be an external input on which it does not impose any restrictions: Topological changes of the surface are supported without special treatment and the precision of the simulation is unaffected of strongly varying triangle sizes. The animation may be key-framed, come from an interactive technique such as skinning, or it may be the output of some real-time physics simulation. If the animation technique does not support efficient random access to former time steps, we keep the vertex positions of the last time step in a buffer. The implicit surface representation and its generation from the deformed triangle mesh are discussed in Section 4.3.2.

### 4.3.1 Semi-Lagrangian Closest Point Extension

As described in Section 2.4.3 (page 30), the closest point extension is a central concept in the equivalence principles which form the basis of the closest point method. For a surface $S$ embedded in three-dimensional Euclidean space, the equivalence of gradients principle, for example, states that the intrinsic surface gradient $\nabla_s u$ and the Cartesian gradient $\nabla v$ agree if the volume function

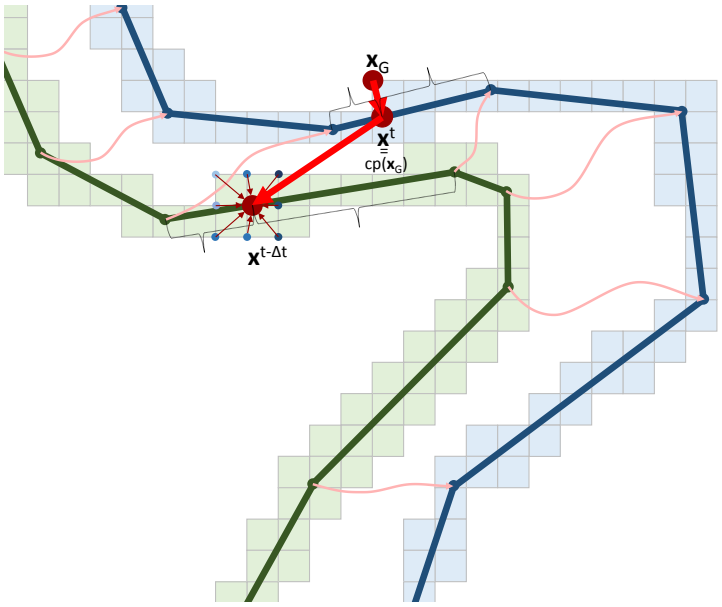


**Figure 4.2**: *The animation has deformed the triangulated surface and moved it to the right. For each grid vertex $\boldsymbol{x}_G$ in the narrow band at time $t$, we determine the closest point on the surface $\boldsymbol{x}^t$ and the position of this point at time $t-\Delta t$. To obtain the closest point extension at $\boldsymbol{x}_G$, we interpolate the values stored in the narrow band of the former time step.*

$v$ is a closest point extension of the surface function $u$:

$$v(\boldsymbol{x}) = u\big(cp(\boldsymbol{x})\big) \qquad \Rightarrow \qquad \nabla_s u(\boldsymbol{x}) = \nabla v(\boldsymbol{x}), \quad \boldsymbol{x} \in S.$$

Like all embedding methods, the explicit closest point method further assumes that the surface function is a restriction of the volume function to the surface. In a discrete integration scheme, it therefore enforces the precondition of the equivalence principles by projecting the volumetric solution to the space of closet point extensions. First the closest surface points of all vertices of a Cartesian grid within a narrow band around the surface are determined. In each integration step, the simulation attributes stored at these grid vertices are then updated with values interpolated at the respective closest points. To ensure consistency and accuracy of the method, a high-order WENO scheme can be employed to interpolate values in the three-dimensional grid.

Semi-Lagrangian schemes are often applied to advection terms in Eulerian fluid dynamics simulations. The traditional technique traces a pathline beginning at a grid vertex backwards in time trough the velocity field. At the foot

of this characteristic curve, the attribute fields of the last time step are sampled using an interpolation technique, to obtain the updated values for the grid vertex. The accuracy of this approach depends on the order of the time-integration employed for the backtrace and the order of the spatial interpolation. Raising only the accuracy of the time integrator can have negative effects, since typically multiple spatial interpolations are required in this case.

Our semi-Lagrangian closest point extension combines the explicit closest point extension with a semi-Lagrangian backtrace along the characteristics defined by the surface motion as depicted in Figure 4.2. In order to ensure a good performance and accuracy of our technique, we directly utilize the triangle animation for the backtrace and employ a stabilized WENO4 scheme for the interpolation (see [MR09] and Section 3.6.3 on page 60). By this means the advection scheme can achieve up to third-order accuracy, depending on the smoothness of the surface and the accuracy of the input animation. For each grid vertex, we first compute the barycentric coordinates of its closest point on the mesh with respect to the closest triangle. The foot of the characteristic is then obtained by applying barycentric interpolation to the vertex positions in the last time step. Since the semi-Lagrangian closest point extension typically needs to be performed several times for multiple closest points per triangle, we accelerate the process by precomputing an implicit surface representation in each time-step as discussed in the next section.

### 4.3.2 Extended Closest Point Representation

To facilitate an efficient GPU implementation of the semi-Lagrangian closest point method, we convert the triangle mesh into an implicit representation after each deformation of the surface, i.e. usually once per time step of the simulation. The implicit surface representation employed in this chapter is an extension of the adaptive closest point grid presented in Section 3.4 (page 51). The embedding space is discretized with a uniform Cartesian grid and the closest points of the grid vertices within a narrow band around the surface are stored in a sparse multi-block grid. For the width of the narrow band we follow the suggestions of Macdonald and Ruuth [MR08]. Additionally to the closest point, our extended implicit representation stores also the index of the closest triangle to enable a coupling to the input animation.

The grid generation and the closest point computation are handled by effi-

cient CUDA algorithms similar to the ones presented in Sections 3.5.1 and 3.5.2. We first determine the coarse-level blocks which overlap the narrow bands and allocate the resources in global GPU memory required to store the closest points, closest triangles and the simulation attributes in the fine-level subgrids. Additionally we determine for each overlapped block the triangles within the narrow band radius. We then iterate over the fine-level vertices of each overlapped block, compute for each vertex the distances to the respective triangles directly and store the index of the closest triangle in global memory. Then we iterate over the fine-level vertices a second time, compute for each of them the closest point on the closest triangle and store it in global memory. While this approach adds another kernel call, a redundant calculation and an indirection over global memory, we found it nevertheless to work faster than the original algorithm because it requires less shared memory to hold the index of the closest triangle (4 byte) instead of the closest point (12 byte).

Together with the time-dependent positions of the triangle vertices, given by the input animation, this implicit surface representation enables the efficient mapping between the grid vertices and the former positions of their closest points, which is required for the semi-Lagrangian closest point extension. As discussed in the last section, we need the barycentric coordinates of the closest points within their respective triangles whenever we perform this mapping. Since the semi-Lagrangian extension needs to be performed for a very large number of grid vertices, the computation of the barycentric coordinates can have a severe impact on the performance of our method. If multiple semi-Lagrangian extensions are required, for example for additional simulation attributes, we could precompute the required data already in the closest point calculation and store it in an additional buffer. This would considerably increase the memory and bandwidth requirements of our method, however. We make instead use of the fact that the number of triangles was always *considerably* below the number of grid vertices in all of our test cases. For each triangle we precompute a barycentric transformation whenever the surface is deformed. This transformation allows us to obtain the barycentric coordinates of multiple closest points on a triangle very efficiently.

To the best of our knowledge, the most efficient way to compute barycentric coordinates for multiple points $\boldsymbol{x}_{cp}$ on a triangle with vertices $\boldsymbol{x}_U, \boldsymbol{x}_V, \boldsymbol{x}_W$ was presented by Schneider [Sch09] (page 121). The idea is to first transform all vector components into a triangle coordinate system with basis vectors

**Algorithm 2** Barycentric Transformation

**Input**
Triangle vertices $\boldsymbol{x}_U, \boldsymbol{x}_V, \boldsymbol{x}_W \in \mathbb{R}^3$.

**Output**
Transformation vectors $\boldsymbol{T}_V, \boldsymbol{T}_W \in \mathbb{R}^4$ for Equation 4.1.

**Procedure**

$$
\begin{array}{lll}
\boldsymbol{t} \leftarrow \boldsymbol{x}_V - \boldsymbol{x}_U, & \boldsymbol{b} \leftarrow \boldsymbol{x}_W - \boldsymbol{x}_U & \\
tu \leftarrow \boldsymbol{t} \cdot \boldsymbol{x}_U, & bu \leftarrow \boldsymbol{b} \cdot \boldsymbol{x}_U & \\
 & & \\
tt \leftarrow \boldsymbol{t} \cdot \boldsymbol{t}, & bb \leftarrow \boldsymbol{b} \cdot \boldsymbol{b}, & tb \leftarrow \boldsymbol{t} \cdot \boldsymbol{b} \\
D \leftarrow tt \cdot bb - tb \cdot tb & & \\
tt \leftarrow tt/D, & bb \leftarrow bb/D, & tb \leftarrow tb/D \\
 & & \\
\boldsymbol{T}_V.xyz \leftarrow \boldsymbol{t} \cdot bb - \boldsymbol{b} \cdot tb, & \boldsymbol{T}_V.w \leftarrow bu \cdot tb - tu \cdot bb & \\
\boldsymbol{T}_W.xyz \leftarrow \boldsymbol{b} \cdot tt - \boldsymbol{t} \cdot tb, & \boldsymbol{T}_W.w \leftarrow tu \cdot tb - bu \cdot tt &
\end{array}
$$

$\boldsymbol{t} := \boldsymbol{x}_V - \boldsymbol{x}_U$ and $\boldsymbol{b} := \boldsymbol{x}_W - \boldsymbol{x}_U$, and then to obtain the barycentric coordinates $\alpha_U, \alpha_V, \alpha_W$ by solving the linear $3 \times 3$ system

$$
\begin{bmatrix}
\boldsymbol{t} \cdot \boldsymbol{x}_U & \boldsymbol{t} \cdot \boldsymbol{x}_V & \boldsymbol{t} \cdot \boldsymbol{x}_W \\
\boldsymbol{b} \cdot \boldsymbol{x}_U & \boldsymbol{b} \cdot \boldsymbol{x}_V & \boldsymbol{b} \cdot \boldsymbol{x}_W \\
1 & 1 & 1
\end{bmatrix}
\cdot
\begin{pmatrix} \alpha_U \\ \alpha_V \\ \alpha_W \end{pmatrix}
=
\begin{pmatrix} \boldsymbol{t} \cdot \boldsymbol{x}_{cp} \\ \boldsymbol{b} \cdot \boldsymbol{x}_{cp} \\ 1 \end{pmatrix}.
$$

Note that the inner products in the above equation transform the covariant vector components from three-dimensional Cartesian coordinates into two-dimensional triangle coordinates.

By taking advantage of matrix inversion and homogeneous coordinates, we precompute two transformation vectors $\boldsymbol{T}_V, \boldsymbol{T}_W \in \mathbb{R}^4$ for each triangle which allow us to obtain the barycentric coordinates of a closest point, given in homogeneous Cartesian coordinates, with only 16 floating point operations as

$$
\alpha_V = \boldsymbol{x}_{cp} \cdot \boldsymbol{T}_V, \quad \alpha_W = \boldsymbol{x}_{cp} \cdot \boldsymbol{T}_W, \quad \alpha_U = 1 - \alpha_V - \alpha_W. \tag{4.1}
$$

The computation of $\boldsymbol{T}_V$ and $\boldsymbol{T}_W$, which requires 61 floating point operations, is described in Algorithm 2.

**Table 4.1**: *Performance statistics for fluid simulation and rendering on the GPU via the semi-Lagrangian closest point method.*

| | Wave Equation | | Navier-Stokes | |
|---|---|---|---|---|
| Resolution | $128^3$ | $320^3$ | $128^3$ | $320^3$ |
| # Closest Points | 79k / 124k | 524k / 860k | 128k / 162k | 884k / 1.2M |
| GPU Memory | 4MiB / 5MiB | 32MiB / 39MiB | 13MiB / 15MiB | 99MiB / 115MiB |
| CPM Solver | 1.1ms / 3.9ms | 2.4ms / 24ms | 8.7ms / 41ms | 54ms / 294ms |
| SLCPM Solver | 15ms / 30ms | 24ms / 75ms | 27ms / 64ms | 84ms / 312ms |
| Raycasting | 14ms / 15ms | 28ms / 30ms | 18ms / 19ms | 31ms / 32ms |
| SLCPM Total | 29ms / 45ms | 52ms / 105ms | 45ms / 83ms | 115ms / 344ms |

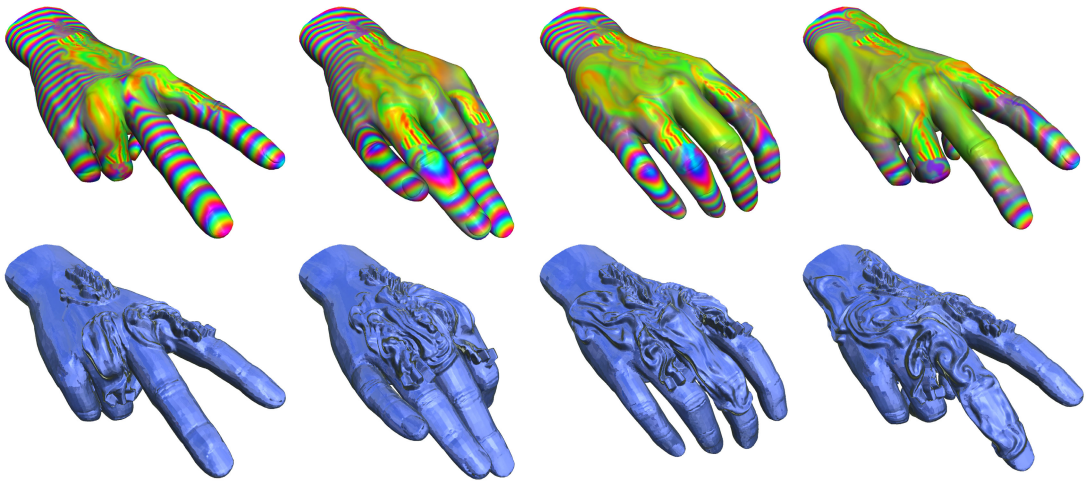

**Figure 4.3**: *Intrinsic incompressible-Navier-Stokes flow simulation on a deforming surface via the semi-Lagrangian closest point method. Top panel: A colored dye is advected in the solution velocity field and visualized via rasterization and pixel shading. Bottom panel: Surface displacements based on an advected height field are simulated via raycasting of the embedding computational grid. At a resolution corresponding to a* $320^3$ *Cartesian grid, simulation and rendering takes less than 120 ms per time step.*

## 4.4 Results and Performance Analysis

The semi-Lagrangian closest point method is able to produce high-quality fluid effects on deforming surfaces as depicted in Figure 4.3. Here, the flow is driven by several attached sources in the form of Neumann boundary conditions, and the appearance and shape of the surface is modulated to visualize the solution. In Figure 4.4 a high-resolution simulation was interactively steered by a user, who painted waves onto a flexing hand and thereby created an visual effect in an exploratory way. The experiment in Figure 4.5 in-

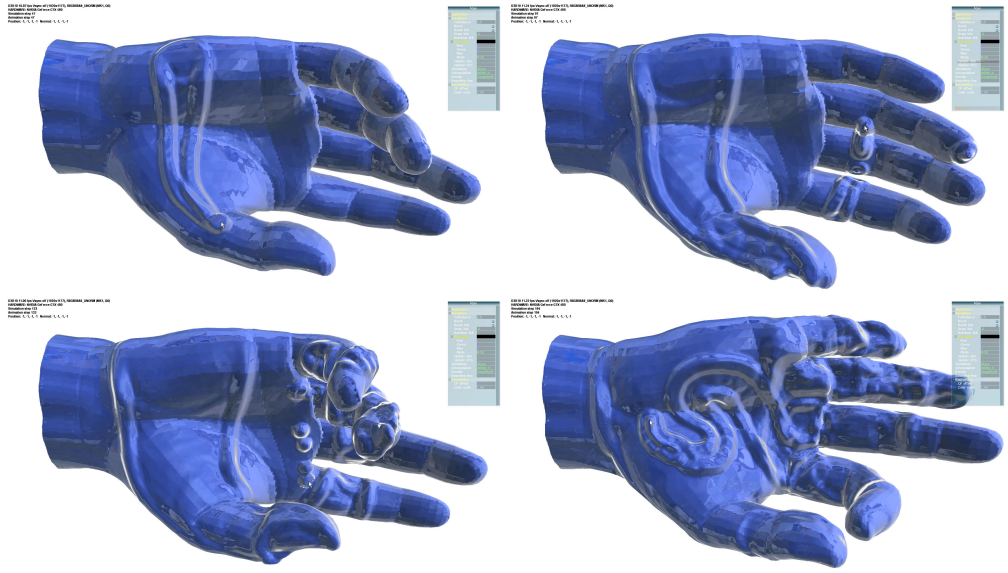

**Figure 4.4**: *Snapshots of an interactive simulation session, in which the user painted waves on a deforming surface. At a resolution corresponding to a* $320^3$ *Cartesian grid and a* $1920 \times 1177$ *viewport, simulation and rendering ran at 8 frames per second on a Geforce GTX 480 graphics card.*

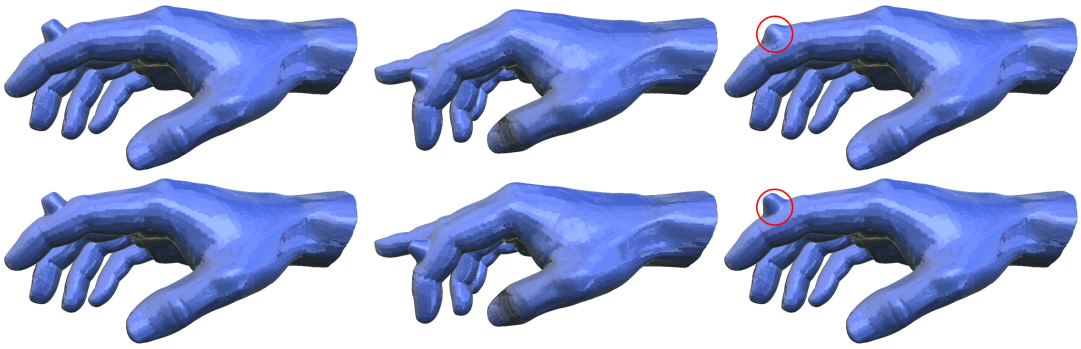

**Figure 4.5**: *The accuracy of the semi-Lagrangian closest point extension depends only on the interpolation scheme. One full animation cycle with disabled PDE solver is shown from left to right. A semi-Lagrangian closest point extension of the height field was computed for each of the 44 animation key-frames. We used linear interpolation in the top panel and stabilized WENO4 interpolation in the bottom panel.*

vestigates the accuracy of the semi-Lagrangian closest point extension. The backtrace along the characteristic is exact up to floating point precision in our scenario, because of the barycentric interpolation of vertex positions on the piecewise-linear surface. The discretization error depends therefore on the spatial interpolation technique. The example clearly shows that higher-order WENO interpolation introduces significantly less numerical dissipation than tri-linear interpolation.

To validate the efficiency of the semi-Lagrangian closest point method, we have performed a number of experiments using simulation grids at different resolutions. In all of our experiments, a key-frame animation of the Laurent hand model was used, consisting of 15855 triangles and 44 animation steps. Note that the surface movements between two consecutive key frames are often an order of magnitude larger than the grid spacing. This could be a problem for advection algorithms with a stability criterion dependent on the Courant-Friedrichs-Lewy condition. All measurements were performed on a 2.4 GHz Core 2 Duo processor and an NVIDIA GeForce GTX 480 graphics card with 1536 MiB local video memory. A detailed memory and performance statistic of the GPU semi-Lagrangian closest point method for the simulation of fluid flow and waves is given in Table 4.1. Here, numbers separated by a slash refer to the simulation using linear, respectively WENO4 interpolation exclusively. All timings are given in milliseconds and rendering was always performed on a $1280 \times 720$ viewport.

The first line gives the resolution of the uniform Cartesian grid to which the simulation resolution corresponds. The second line lists the number of closest points within the computational band. The differences are due to differently sized computational bands that are dictated by the numerical stencils of the respective discretization methods. The third line gives the GPU memory requirements of semi-Lagrangian closest point method. The Navier-Stokes simulation requires more memory due to its larger computational band and the additional buffers required to store the simulation attributes. For comparison, the fourth line lists the simulation times on the static surface, using linear and WENO4 interpolation in the closest point extension. The fifth line gives the respective timings of the semi-Lagrangian closest point method on the animated surface. As expected, the increase in simulation time between 15 and 50 milliseconds definitely contains a constant factor due to the generation of the extended closest point representation. In Chapter 3, we already mentioned

that this step scales mainly in the number of triangles. The WENO4 interpolation therefore remains the most expensive operation at higher resolutions. The last two lines show the rendering times for raycasting with smooth surface displacements, as discussed in Section 3.7.2, and the total time for simulation and rendering.

## 4.5 Conclusion and Further Work

In this chapter, we have presented a real-time Eulerian method for the simulation of fluid flow and wave propagation on deforming triangulated surfaces. Our approach successfully combines the semi-Lagrangian method and the closest point method, and rigorously exploits the synergies between the two. The method can simulate intrinsic fluid effects at a high, constant and uniform resolution, and its precision does not depend on the input geometry.

The Eulerian discretization of the surface sidesteps all problems with dynamic parametrizations in Lagrangian methods. The embedding of the surface into a three-dimensional Euclidean space also results in an extrinsic notion of the surface topology, however. This especially means that the simulation treats a deformation leading to a self-intersection as a connection of the respective surface parts. Depending on the goals of the user, this may be seen as an advantage or disadvantage.

The proposed method opens a number of future research directions. By enabling support for level-of-detail changes of both the input triangulation and the Eulerian computational grid, the algorithm could run with a fixed time budget, a requirement for the application in computer games. We plan also to combine our method with a semi-Lagrangian contouring approach [BGOS06] in order to employ it for the up-resing of interactive, fully three-dimensional fluid simulations, e.g. [CLT07, CM11]. The integration with techniques for free-surface flow would also be an important step towards the real-time simulation of surface tension [TWGT10] and the interactive artistic control of fluid behavior [SY05].

# 5

# Direct Contouring of Implicit Closest Point Surfaces

Constructing a (signed) distance field and contouring its zero level set are two important steps in many surface reconstruction methods. While most high-quality distance transforms compute the distance to the surface as well as the closest point on it, the contouring step typically uses only the distance and omits the closest point information. Our novel closest point contouring algorithm (CPC) uses the full closest point field, and, thus, allows improving existing methods for high-quality triangle mesh reconstruction based on implicit function models: Since we select the vertex positions directly from the set of closest points, all triangle vertices are guaranteed to lie exactly on the zero-contour and no approximations are necessary. By employing recent findings in the context of so-called embedding techniques, we derive a formulation of the mean curvature vector on the closest point representation and use this formulation to properly select the vertices to be triangulated. In combination with a new table-based triangulation scheme this allows us to detect and preserve sharp features, and to avoid small degenerated triangles in smooth areas. Closest point contouring can handle open and non-orientable surfaces, and its data-parallel nature makes it well suited for GPUs.

## 5.1 Introduction

Implicit surfaces which are defined as zero level sets in discrete volumetric distance fields are used as intermediate surface representations in a number of

computer graphics and geometry processing applications, ranging from constructive solid geometry and point cloud surface reconstruction to fluid simulation and physics-based modeling. A distance field can be computed in a number of ways. According to [JBS06] there are two categories of such algorithms: Exact or direct algorithms compute explicitly the Euclidean distance between a vertex of the discrete sampling grid and its corresponding closest point on the continuous surface. Approximate distance transforms perform a direction-dependent uneven propagation of distance measures from vertex to vertex, typically building up distances in increments to speed up computation time. The distance measure can either be scalar-valued, or the distance transform builds upon the propagation of vectors to the closest surface points to achieve superior accuracy. High quality algorithms from both categories can thus be extended easily to also generate a closest point representation of a given surface, i.e., a 3D surface-embedding grid which stores at each grid vertex the closest point on the surface. In Section 2.4.5 (page 33) we list a number of general methods to generate such a discrete closest point function from a number of other surface representations and in Section 3.5.2 (page 56) we show how the closest points within a band around a triangulated surface can be calculated efficiently on the GPU.

For constructing a triangle mesh from a distance field, contouring algorithms like the Marching Cubes (MC) algorithm [LC87] can be employed. The marching cubes algorithm works solely on a scalar distance field and extracts a triangle mesh by computing intersection points between the level-0 surface and the grid edges via linear interpolation of distance values. If the individual directed distances between the grid vertices and the surface along the coordinate axes are known, the Extended Marching Cubes (EMC) algorithm [KBSS01] interpolates separately between these distances to obtain exact intersection points. In contrast to the marching cubes algorithm, the extended marching cubes algorithm as well as the Dual Contouring (DC) algorithm [JLSW02] also extract sharp features by using the normals of the implicit surface to evolve the contour into the interior of grid cells. This requires the solution of a quadric error function to obtain the interior-cell surface points via plane intersections.

## 5.2 Contribution

We propose a new contouring approach that entirely avoids the approximation of interior-cell surface positions, and, instead, directly obtains a high-quality surface triangulation from a closest point representation. The construction of a closest point field is no more complicated than the construction of a high-quality distance field, and the closest points can be used directly as mesh vertices. Since the closest points lie exactly on the surface, there is no need for any approximation of triangle vertices in the interior of grid cells. We will subsequently call our proposed algorithm Closest Point Contouring (CPC).

It is important to note, however, that it is not clear per se which subset of the closest points in the 3D embedding grid should become the vertices of the triangle mesh, and how to connect the vertices in order to extract a high quality triangulation. Of all closest points that are stored at the vertices of the 3D embedding grid, only those should be used which are required to accurately represent the shape of the initial surface. In this work we propose a vertex selection oracle which makes use of the closest point method (see Section2.4). This oracle performs a Laplacian analysis on the embedding closest point grid, and uses the mean curvature vectors at the closest points to guide the selection of mesh vertices according to the surface's shape.

In summary, the main contributions presented in this chapter are:

- An efficient, high-quality contouring method that works solely on a surface embedding 3D closest point grid.
- An oracle for selecting a feature-preserving subset of all closest points, which is based on a Laplacian analysis of the embedding grid.
- A novel table-based algorithm for triangulating the voxels of a 6-separating surface voxelization.
- A GPU implementation of the closest point contouring pipeline, which achieves a reconstruction performance suitable for real-time applications.

The remainder of this chapter is structured as follows: In Section 5.3 we review previous work that is related to ours. Section 5.4 begins with a high-level overview of the contouring algorithm. In Sections 5.4.1 and 5.4.2 we introduce a first strategy to select a set of closest points for triangulation and an

algorithm that triangulates such a set of vertices to obtain a triangle mesh of the contour. We then extend our vertex selection strategy to detect and recover sharp surface features in Section 5.4.3 and discuss the GPU implementation of the closest point contouring algorithm in Section 5.5. In Sections 5.6 and 5.7 we conclude the chapter with a presentation of the results and an outlook to future work.

## 5.3 Related Work

The marching cubes algorithm [LC87] is the most popular contouring algorithm, and it has influenced most of the later work in surface reconstruction in one way or another. The marching cubes algorithm takes as input a scalar field on a hexahedral grid structure and generates a piecewise tri-linear approximation to a selected iso-contour in the scalar field. Given a distance field, the level-0 surface extracted by the marching cubes algorithm is an approximation to the surface to which the distance field was computed. However, because the marching cubes algorithm considers only one single distance value per grid vertex, regardless of the direction along which the closest surface point was found, and approximates the contour in the interior of grid cells, it fails in general to extract sharp surface features accurately.

The extended marching cubes (EMC) algorithm [KBSS01] overcomes the limitations of the marching cubes algorithm by considering the exact intersection points between the level-0 surface and the grid edges as well as the surface normals at these points. The normals indicate the orientation of the contour, so that properly oriented planes can be fit and intersected to find a contour point in the interior of cells. This vertex is then considered in the triangulation to accurately represent sharp features.

The dual contouring (DC) algorithm [JLSW02] further extended on the extended marching cubes algorithm to improve the algorithm output in the case that sharp edges lie in the interior of grid cells. In the spirit of [Gib98], the dual contouring algorithm generates a triangle mesh topologically dual to the one of marching cubes. By a plane fit similar to the one performed in the extended marching cubes algorithm, dual contouring generates one new point for every grid cube that intersects the contour. These points are connected across cube boundaries to form a quadrilateral contour representation. In later years, the dual contouring algorithm was extended in many different directions (see,

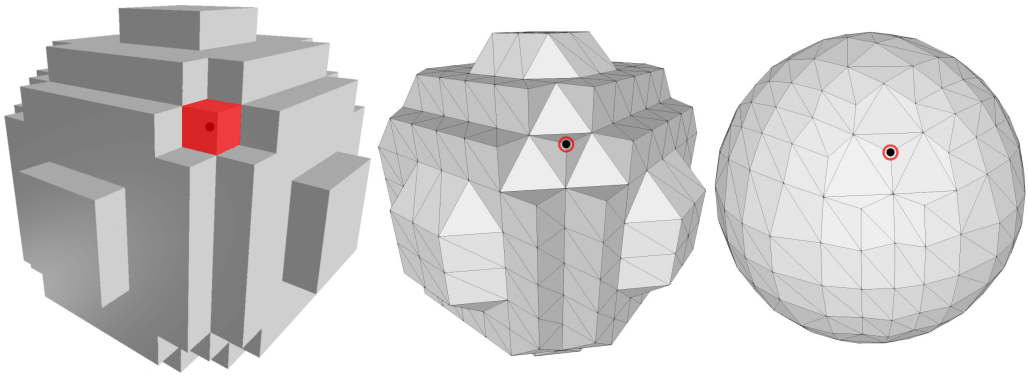

**Figure 5.1**: *Left: In the dual grid, the selection of cells corresponds to a 6-separating surface voxelization. Middle: We generate the topology of a closed triangle mesh from the respective selection of primal grid vertices. Right: Applying this topology to the respective selection of closest points results in a tessellation of the surface.*

e.g., [SW04, Ju06, SJW07, MS10] and references therein).

A different class of contouring algorithms is based on so-called marching fronts. These algorithms start at one or more points on the surface, and they then try to evolve the surface incrementally by adding additional triangles. Since our work does not belong to this class, we point the interested reader to the discussion in [SSS06].

It is also worth noting that several algorithms exist which re-mesh the output of a contouring algorithm to adapt to local features. Edge sharpener [AFRS03] is one prominent example of this class. The survey of Alliez et al. [AUGA05] provides an excellent overview of the re-meshing approaches and lists the most powerful schemes.

## 5.4 The Closest Point Contouring Algorithm

The closest point contouring algorithm is a two pass approach, in which each closest point corresponds to its defining vertex in the primal embedding grid and to the cell in the dual grid which is centered at the primal vertex.

The first pass iterates over the primal grid vertices and selects a subset of their closest points as mesh vertex candidates. We interpret the corresponding selection of cells in the dual grid as a surface voxelization. To obtain a closed triangulation in the second step, this voxelization must be 6-separating [KCY93]. A realization of the first pass for smooth surfaces is discussed in 5.4.1. The contour shown in Figure 5.6 bottom right and in Figure 5.7 bottom left was

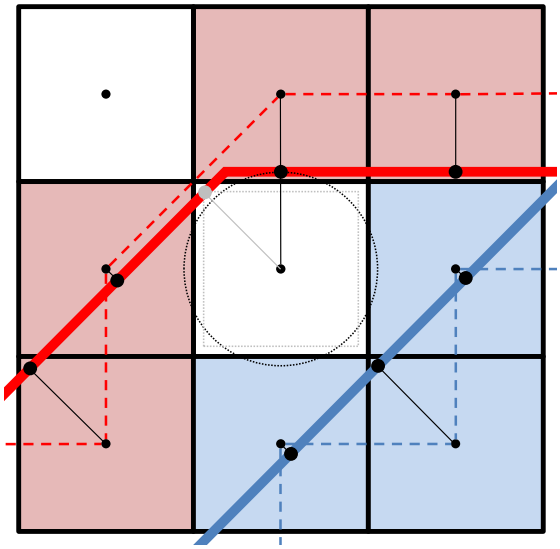

**Figure 5.2**: *Our triangulation-aware vertex selection oracle approximates the intersection of the surface with the cells of the dual grid. The intersection of the red surface with the cell in the center is not found because the Euclidean closest point of the corresponding primal grid vertex is outside the cell. The resulting voxelization is nevertheless still 6-separating, which is necessary to safely deduce the topology of the surface mesh (dashed lines).*

reconstructed with this technique. In Section 5.4.3 we present an advanced vertex selection oracle for surfaces with sharp features, which was used for all other closest point contouring results shown in this chapter.

The second pass iterates over all primal grid cells with three or more selected corner vertices, and, comparable to the marching cubes algorithm, it determines the connectivity of the respective candidate closest points with a lookup into a triangulation table. Contrary to marching cubes, however, closest point contouring does not restrict the triangle vertices to the edges of the grid, but instead uses the exact closest surface points of the grid vertices. The second pass is detailed in Section 5.4.2. Figure 5.1 recaps the basic concept behind the entire algorithm in a visual way.

### 5.4.1 Triangulation-Aware Closest Point Selection

The first pass of our contouring algorithm is an oracle which selects a subset of the closest points as mesh vertex candidates. Assuming a closed, smooth and curved surface, the selection should lead to a uniform vertex placement, which results in neither too large nor in too small or degenerated triangles.

Furthermore, the selection should be based not solely on the closest point locations. Instead it should also consider the connections between the defining vertices in the primal grid, to assist the topology generation in the second pass. After experimenting with different strategies, we found that an almost minimal 6-separating selection of cells in the dual grid leads to a reasonable surface sampling in the corresponding selection of closest points. This type of selection has the additional advantage that the connectivity of the selected primal grid vertices already defines a watertight topology which can be used for the triangle mesh.

A small, but not necessarily minimal, 6-separating closest point selection can be computed by finding the voxels containing the surface. To do so, we consider only the distances between each voxel center $\boldsymbol{x}_v$ (i.e., the position of the primal grid vertex) and its closest point on the surface $\boldsymbol{cp}(\boldsymbol{x}_v)$. 6-separation is guaranteed if we select closest points with a Euclidean distance less than half the voxel diameter. Such a selection, however, contains many false positives, which leads to over-tessellation and degenerated triangles.

To obtain a less conservative selection, we calculate the distances based on the $L^\infty$-norm instead of the $L^2$-norm, and select all closest points with a Chebyshev distance $D_C\left(\boldsymbol{p},\boldsymbol{q}\right) := \max_i\left(|p_i - q_i|\right)$ less than half the grid spacing $h$. This voxel-surface intersection test is exact if the closest point representation is also based on the $L^\infty$-norm. For a Euclidean representation it can miss intersections if the Euclidean closest point is outside the voxel and the Chebyshev closest point is inside, as depicted in Figure 5.2. Yet even in this case the resulting selection is 6-separating, which is why we always prefer the $L^\infty$-norm for distance computations.

### 5.4.2 Table-Based Closest Point Triangulation

The second pass of the algorithm tessellates the surface by creating a triangle-mesh topology for the selected candidate closest points. It processes each cell of the primal grid independently and takes only the edge selection flags from the first pass as input. Like marching cubes, closest point contouring determines the triangle connectivity for a cell by building an 8-bit index from the local indices (depicted in Figure 5.3) of the selected cell vertices. This index identifies one out of 256 possible cube configurations in a precomputed triangulation table. We should note, however, that the triangle vertices (i.e., the

| Configuration | Case | Triangles | Configuration | Case | Triangles |
|---|---|---|---|---|---|
| ----3210 | F | 021 123 | 7--4-21- | 6 | 124 217 147 427 |
| ---4-21- | 0 | 124 | 7--4-210 | 7 | 217 147 427 120 410 240 |
| ---4321- | 1 | 124 213 | 7--43--0 | 4 | 734 430 |
| ---43210 | F | 021 123 | 7--43-1- | 1 | 714 173 |
| --5-3--0 | 0 | 053 | 7--43-10 | 2 | 147 410 173 |
| --5-32-0 | 1 | 305 032 | 7--432-- | 1 | 274 723 |
| --5-3210 | F | 021 123 | 7--432-0 | 2 | 274 723 240 |
| --54--10 | F | 014 415 | 7--4321- | 7 | 124 274 714 213 723 173 |
| --54-21- | 1 | 412 145 | 7--43210 | 5 | 734 430 130 032 |
| --54-210 | F | 014 415 | 7-5--2-0 | 4 | 207 705 |
| --543--0 | 1 | 053 504 | 7-5--21- | 1 | 172 715 |
| --543-10 | F | 014 415 | 7-5--210 | 2 | 172 715 120 |
| --5432-- | 4 | 325 524 | 7-5-3--0 | 1 | 530 357 |
| --5432-0 | 2 | 053 504 032 | 7-5-32-0 | 2 | 305 032 357 |
| --54321- | 2 | 124 213 145 | 7-5-3210 | F | 021 123 |
| --543210 | F | 014 415 021 123 | 7-54--10 | F | 014 415 |
| -6--3--0 | 0 | 603 | 7-54-2-- | 1 | 472 745 |
| -6--3-10 | 1 | 036 301 | 7-54-2-0 | 2 | 427 240 475 |
| -6--3210 | F | 021 123 | 7-54-21- | 7 | 172 742 412 715 475 145 |
| -6-4-2-0 | F | 042 246 | 7-54-210 | 5 | 207 705 105 504 |
| -6-4-21- | 1 | 241 426 | 7-543--0 | 2 | 530 357 504 |
| -6-4-210 | F | 042 246 | 7-543-10 | F | 014 415 |
| -6-43--0 | 1 | 603 064 | 7-5432-- | 2 | 724 273 745 |
| -6-43-1- | 4 | 134 436 | 7-5432-0 | 3 | 053 504 032 357 |
| -6-43-10 | 2 | 036 301 064 | 7-54321- | 5 | 452 253 153 357 |
| -6-432-0 | F | 042 246 | 7-543210 | F | 014 415 021 123 |
| -6-4321- | 2 | 241 426 213 | 76----10 | 4 | 107 706 |
| -6-43210 | F | 042 246 021 123 | 76---21- | 1 | 721 276 |
| -65----0 | 0 | 506 | 76---210 | 2 | 217 120 276 |
| -65---10 | 1 | 506 051 | 76--3--0 | 1 | 360 637 |
| -65--2-0 | 1 | 065 602 | 76--3-10 | 2 | 360 637 301 |
| -65--21- | 4 | 261 165 | 76--3210 | F | 021 123 |
| -65--210 | 2 | 065 602 051 | 76-4--1- | 1 | 471 746 |
| -65-3--- | 0 | 356 | 76-4--10 | 2 | 471 746 410 |
| -65-3--0 | 6 | 603 065 635 305 | 76-4-2-0 | F | 042 246 |
| -65-3-1- | 1 | 356 531 | 76-4-21- | 7 | 721 241 471 276 426 746 |
| -65-3-10 | 7 | 506 036 356 051 301 531 | 76-4-210 | 5 | 107 706 406 602 |
| -65-32-- | 1 | 635 362 | 76-43--0 | 2 | 603 064 637 |
| -65-32-0 | 7 | 065 635 305 602 362 032 | 76-43-1- | 2 | 714 173 746 |
| -65-321- | 2 | 356 531 362 | 76-43-10 | 3 | 603 064 637 301 |
| -65-3210 | 5 | 516 612 012 213 | 76-432-0 | F | 042 246 |
| -654--10 | F | 014 415 | 76-4321- | 5 | 134 436 236 637 |
| -654-2-0 | F | 042 246 | 76-43210 | F | 042 246 021 123 |
| -654-21- | 2 | 412 145 426 | 765----0 | 1 | 650 567 |
| -654-210 | F | 042 246 014 415 | 765---10 | 2 | 506 051 567 |
| -6543--- | 1 | 563 654 | 765--2-0 | 2 | 650 567 602 |
| -6543--0 | 7 | 053 563 603 504 654 064 | 765--21- | 2 | 721 276 715 |
| -6543-1- | 2 | 563 654 531 | 765--210 | 3 | 506 051 567 602 |
| -6543-10 | 5 | 643 341 041 145 | 765-3--0 | 7 | 530 360 650 357 637 567 |
| -65432-- | 2 | 635 362 654 | 765-3-10 | 5 | 670 071 571 173 |
| -65432-0 | 5 | 325 524 024 426 | 765-32-0 | 5 | 570 072 372 276 |
| -654321- | 3 | 356 531 362 654 | 765-3210 | F | 021 123 |
| -6543210 | F | 042 246 014 415 021 123 | 7654--10 | F | 014 415 |
| 7----21- | 0 | 217 | 7654-2-0 | F | 042 246 |
| 7----210 | 1 | 217 120 | 7654-21- | 5 | 261 165 465 567 |
| 7---3210 | F | 021 123 | 7654-210 | F | 042 246 014 415 |
| 7--4--1- | 0 | 714 | 76543--0 | 5 | 043 347 547 746 |
| 7--4--10 | 1 | 147 410 | 76543-10 | F | 014 415 |
| 7--4-2-- | 0 | 274 | 765432-0 | F | 042 246 |
| 7--4-2-0 | 1 | 427 240 | 76543210 | F | 042 246 014 415 021 123 |

**Table 5.1**: *The triangulation table in condensed form.*

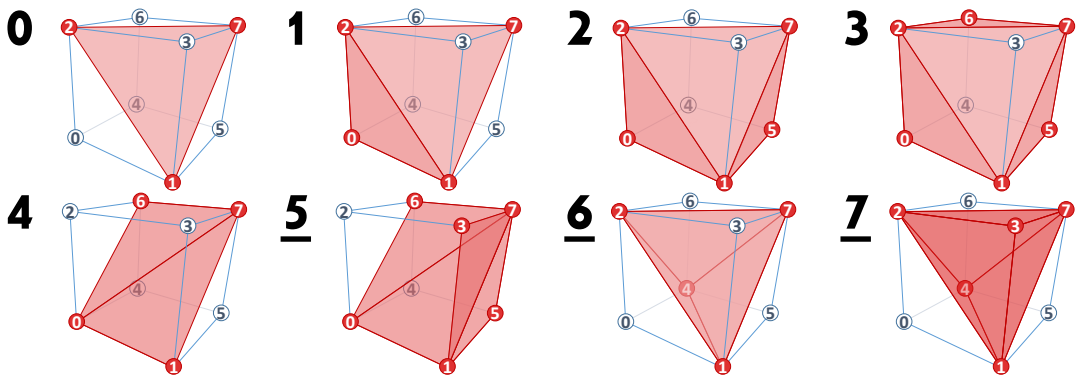


**Figure 5.3**: *Triangulated cubes*

closest points) belong to the vertices of the primal grid and not to its edges or cells. Therefore, the resulting triangulations cannot necessarily be classified as primal or dual regarding to the marching cubes or dual contouring triangulations.

Each entry in the triangulation table is a triangle list containing the local indices of up to 6 triangles. To fill the table, we first consider the cube faces containing the corner with local index 0, to avoid redundant evaluation in adjacent cells. If the closest points of all 4 vertices of a face are selected, we triangulate this face by adding a quad of two triangles to the respective table entry. In a second step, we consider the 8 cases depicted in Figure 5.3 and all of their permutations arising from the 48 cube symmetries (24 rotations and reflection). If a cube configuration matches one of these cases, the respective table entry is filled with the triangulation depicted in the figure (rotated or reflected as required). The triangulation table is presented in condensed form in Table 5.1, which lists only the non-empty triangulations. The cube configuration in the first column consists of the local indices of the selected cell vertices. The numbers in the second column represent the triangulation cases in Figure 5.3, whereas an F marks the cases where one or more faces of the cube are triangulated. The last column lists the local indices of the generated triangles.

For the underlined cases in Figure 5.3, which represent non-manifold triangulations, we set an additional non-manifold bit in the triangulation table. If this bit is detected after the first table lookup, we treat the most distant selected closest point as unselected and compute a new cube configuration. A manifold triangulation is then obtained with a second lookup. The additional lookup can be omitted, if manifoldness is less important than performance.

### 5.4.3 Feature-Sensitive Closest Point Selection

While the vertex selection oracle described in Section 5.4.1 yields convincing results in smooth surface areas, the resulting triangle meshes do not reproduce sharp features accurately. This is a direct result of the uniform sampling strategy, which does not consider the curvature of the surface.

As an extension of the closest point selection pass, we therefore adjust the initial selection from Section 5.4.1 so that closest points on sharp features are preferred. The idea underlying our approach is illustrated in Figure 5.4. Panel A shows the initial selection of voxels from Section 5.4.1 and the resulting closest point contouring reconstruction. In the panels B and C this selection was moved to the 6-adjacent voxels on the inside, respectively outside of the object. We observe that features are preserved where the selection was moved to the locally convex side of the surface. The reason for this effect is that in a certain area on the convex side near an edge or corner all voxel centers are closest to a point on the feature. Panel D shows the result of the feature-sensitive closest point selection. Here the selection was adaptively moved towards the locally convex side, guided by the negative mean curvature vectors (purple arrows) to identify areas of high curvature and their convex side.

The mean curvature vector at a point $\boldsymbol{x}_s$ on the continuous surface $S$ is given by

$$\boldsymbol{H}(\boldsymbol{x}_s) = \nabla_S^2 \, \boldsymbol{id}(\boldsymbol{x}_s), \tag{5.1}$$

where $\nabla_S^2$ denotes the Laplace-Beltrami operator, and $\boldsymbol{id}(\boldsymbol{x}) = \boldsymbol{x}$ is the identity function. $\boldsymbol{H}(\boldsymbol{x}_s)$ is parallel to the surface normal, points to the locally concave side of $S$ (i.e., on convex objects it points inside), and its length equals the absolute value of the mean curvature. The closest point method allows us to define $\boldsymbol{H}$ on the discrete embedding grid. The equivalence of gradients principle (see Section 2.4.1) states that the Laplace-Beltrami operator applied to a function $f$ and the standard Cartesian Laplace operator applied to the *closest point extension* of $f$ agree on the surface:

$$\nabla_S^2 f(\boldsymbol{x}_s) = \nabla^2 f\left(\boldsymbol{cp}(\boldsymbol{x}_s)\right)$$

Since the closest point function $\boldsymbol{cp}(\boldsymbol{x})$ is the closest point extension of the identity function, we can rewrite Equation 5.1 to

$$\boldsymbol{H}(\boldsymbol{x}_s) = \nabla^2 \, \boldsymbol{cp}(\boldsymbol{x}_s). \tag{5.2}$$

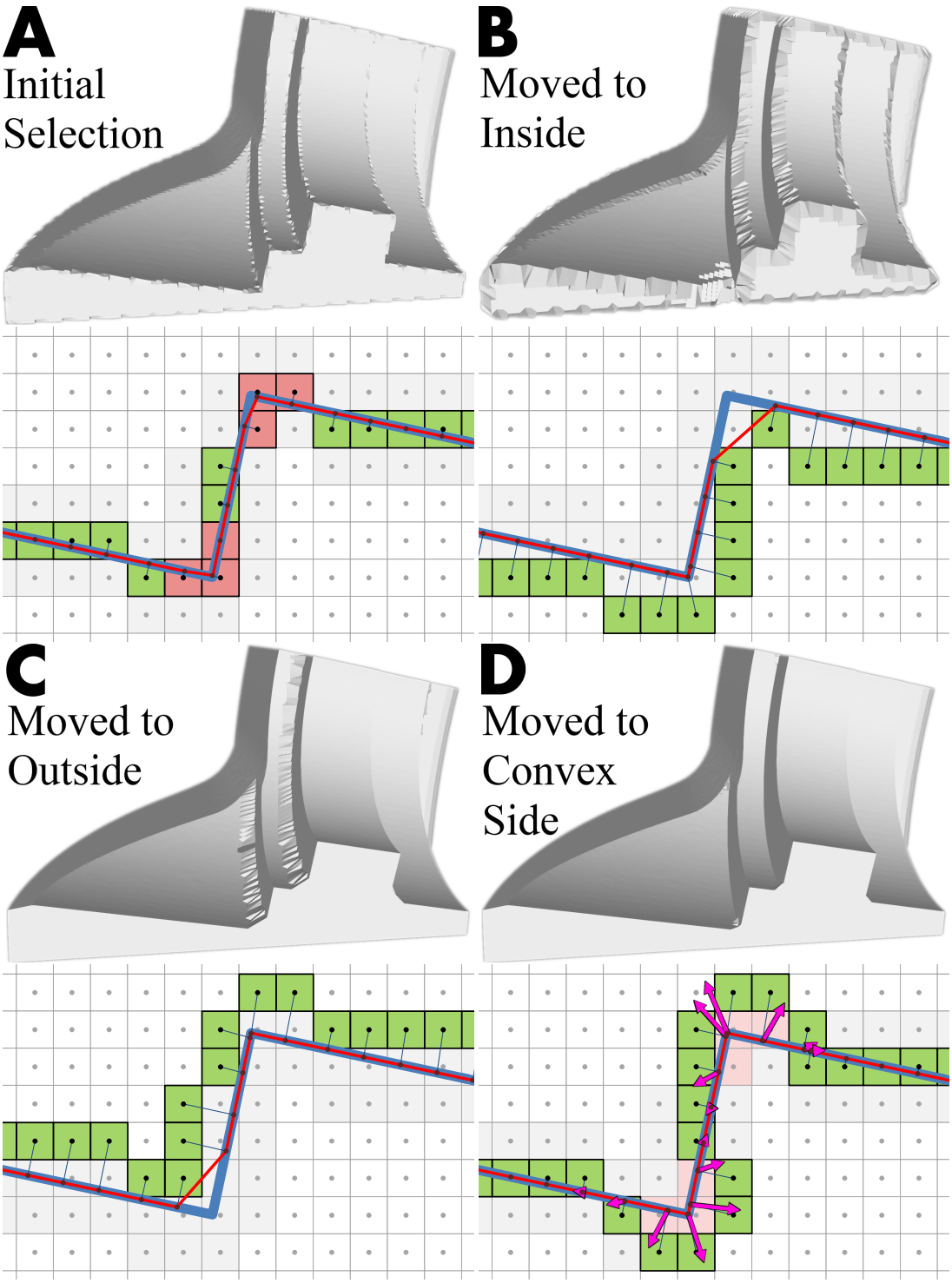


**Figure 5.4**: *Effect of moving the selection to the convex side*

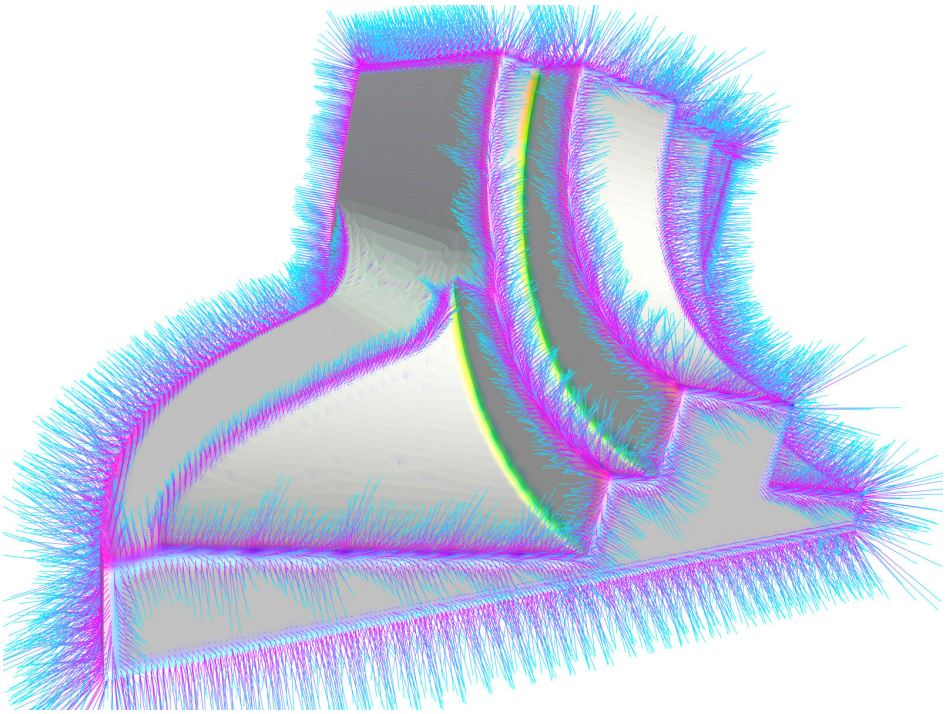

**Figure 5.5**: *The negative mean curvature vectors, visualized as color-coding on the surface and as line primitives to emphasize the locally convex side.*

To discretize this PDE, a standard discrete Laplace operator with a $3 \times 3 \times 3$ stencil is applied to the closest points in the embedding grid. The result $\tilde{\boldsymbol{H}}$ on the discrete grid equals $\boldsymbol{H}$ at points on the surface. Since a grid vertex $v$ does not coincide with a surface point in general, we evaluate $\tilde{\boldsymbol{H}}$ at the closest point of $v$: $\boldsymbol{H}(\boldsymbol{x}_v) = \tilde{\boldsymbol{H}}\left(\boldsymbol{cp}(\boldsymbol{x}_v)\right)$. This discrete closest point extension is implemented as a WENO4 interpolation [MR08]. Figure 5.5 visualizes the negative mean curvature vectors which are obtained with the closest point method in the described way.

To guide the movement of the initial selection to the convex side, we classify the grid vertices based on their mean curvature vectors. The *classification* stage first computes the initial selection and the mean curvature vector $\boldsymbol{H}(\boldsymbol{x}_v)$ in the described ways, and then decides for each grid vertex $v$ if its selection and deselection are allowed. A selection is allowed only if $v$ was never selected before and if it does not lie on the locally concave side. Concavity is assumed if the inner product $\boldsymbol{H}(\boldsymbol{x}_v) \cdot \left(\boldsymbol{cp}(\boldsymbol{x}_v) - \boldsymbol{x}_v\right)$ is larger than a small $\epsilon$. A deselection is allowed only if $v$ is close to a feature and if the resulting selection is still

6-separating. We assume that a feature is close if

$$D_C\left(\boldsymbol{cp}\left(\boldsymbol{x}_v\right),\boldsymbol{x}_v\right)+h/2 \quad < \quad \lambda\left|\boldsymbol{H}(\boldsymbol{x}_v)\right|,$$

with $D_C$ and $h$ being the Chebyshev distance respectively grid spacing as defined in Section 5.4.1, and with $\lambda$ being a user-defined value to control the sensitivity of the feature detection. 6-separation is preserved if $\boldsymbol{H}(\boldsymbol{x}_v)\cdot\boldsymbol{H}(\boldsymbol{x}_{\tilde{v}})>\epsilon$ for all $\tilde{v}$ in the 26-neighborhood, i.e., if the concave side does not change at the adjacent voxels.

Based on the classification we apply two iterations of the *move* stage to the initial selection. In each iteration, we check for each currently selected vertex if its deselection is allowed. If allowed, we deselect the vertex and instead select all of its 6-adjacent neighbors which are selectable.

## 5.5 Closest Point Contouring on the GPU

Our closest point contouring pipeline is entirely GPU-based. It adopts the adaptive closest point surface representation introduced in Section 3.4 (page 51) and the CUDA implementation of the closest point method presented in Section 3.6 (page 58). A primal grid vertex $v$ within the computational bands can therefore be identified either by its Cartesian coordinates or by its unique index within the linear buffers of the adaptive multiblock grid, which store all per-grid-point attributes.

The first pass, the feature-sensitive vertex selection oracle, is implemented in four different CUDA kernels which run in one thread per vertex of the adaptive grid. If required, double buffering is used to avoid possible concurrency hazards. The first two kernels calculate the mean curvature vector $\boldsymbol{H}\left(\boldsymbol{x}_v\right)$ via the closest point method, i.e., they apply a discrete Cartesian Laplace operator to the closest points and then perform a closest point extension. The third kernel generates the initial mesh-vertex selection and the classification, and stores them in a bit mask buffer containing the three flags: *selected*, *selectable* and *deselectable*. The first three kernels belong to the classification stage, which is executed once. The fourth kernel implements the move stage and is executed twice. Each of its threads checks if the respective grid vertex is currently selected and deselectable, in which case it clears all three flags and sets the selected flag of the selectable 6-adjacent neighbor vertices.

The second pass, the table-based triangulation, is implemented in one Direct3D 10 geometry shader. Depending on the application, the triangulation can therefore happen on-the-fly during rendering or the triangle-mesh topology can be written into an index buffer in the stream output stage. The shader has access to an integer buffer which encodes the triangulation table and to the buffers of the adaptive multiblock grid, which contain the closest points and the vertex selection flags. To trigger the triangulation, we set the primitive topology to *point-list* and issue a draw call with as many pseudo-vertices as there are vertices in the adaptive multiblock grid. By this means, we run the geometry shader once for each cell of the primal grid within the computational bands and set the system-generated primitive id to the buffer index of the first cube corner.

In the geometry shader, we determine the Cartesian coordinates and the buffer indices of the eight cube corners, and load the respective vertex selection flags. If less than three cube corners are selected, the execution is stopped without generating any triangles. Otherwise the positions of the selected triangle vertex candidates are loaded from the closest point buffer, which serves as a vertex buffer except that it is not bound to the input assembler stage. If additional triangle-vertex attributes are required, we load them from the respective buffers or compute them on-the-fly. When the vertex candidates are assembled and the usual transformations are applied, we append the buffer indices as last attribute so that the stream-output stage can optionally capture the mesh topology. Using the bit-shift operator, the selection flags are combined into the cube configuration index and the triangle configuration is loaded from the triangulation table. If manifoldness is enforced, we optionally check the non-manifold flag and obtain a new triangle configuration if required. For each triangle in the configuration, we append three assembled vertices to the output stream.

## 5.6 Results and Performance Analysis

Note that all triangle meshes in this chapter were rendered with per-pixel lighting using flat triangle normals, to enable a better judgment of the mesh quality. Figure 5.6 shows that closest point contouring effectively preserves sharp creases and corners (top left), that it generates reasonable triangulations for smooth areas as well as for smaller details (top right), and that it supports open

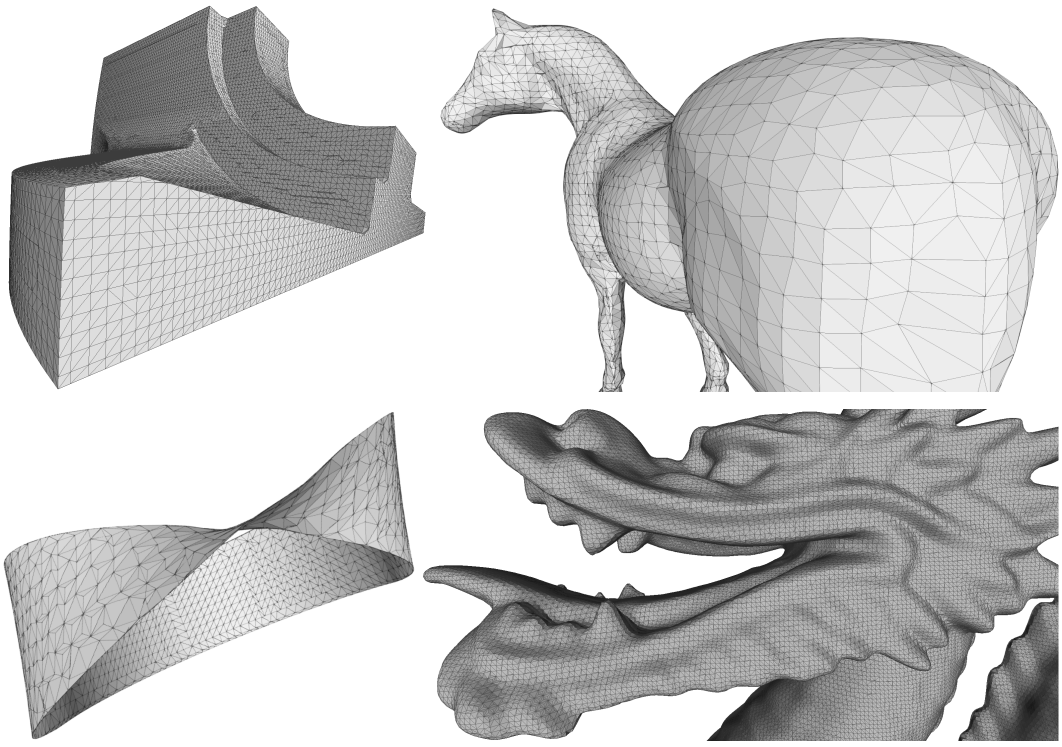

**Figure 5.6**: *Triangle meshes reconstructed from closest point fields.*

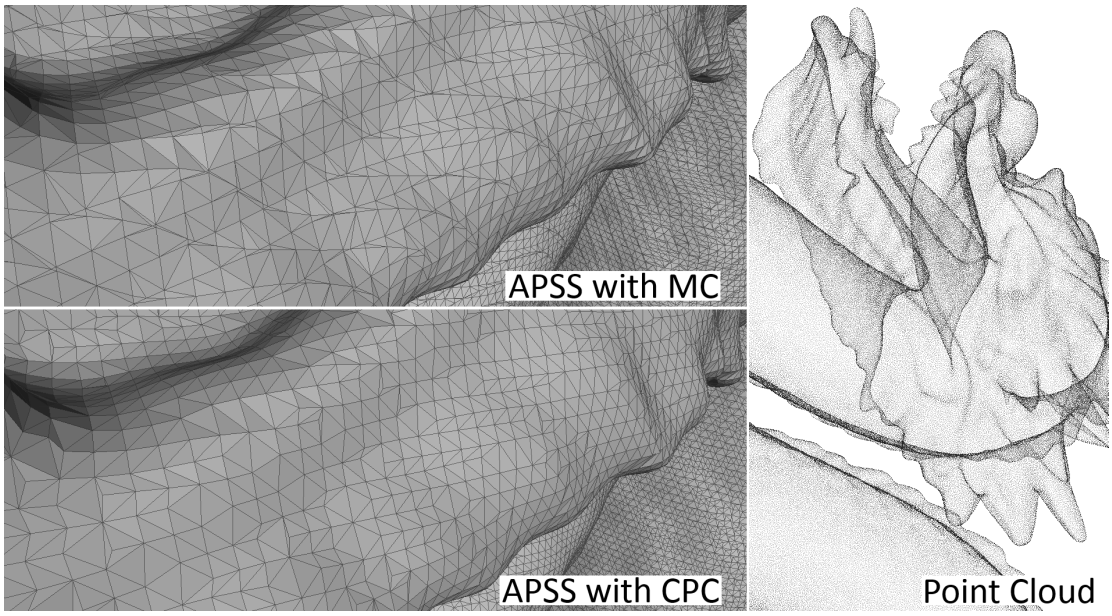


**Figure 5.7**: *Integration with an existing moving-least-squares surface reconstruction tool for point clouds. We replaced the marching cubes contouring step of the algebraic point set surfaces algorithm [GG07] with closest point contouring.*

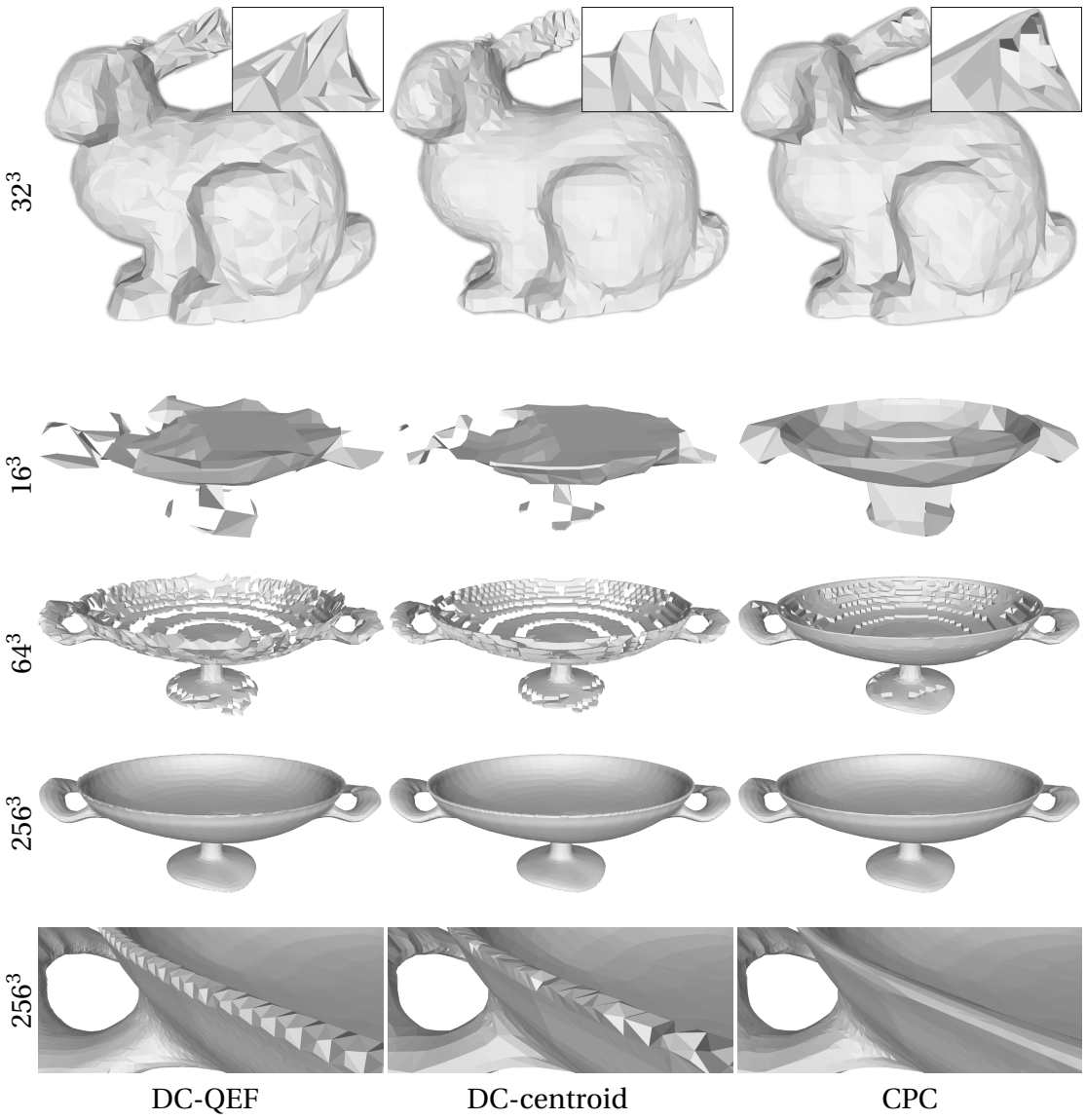


**Figure 5.8**: *Dual contouring compared to closest point contouring*

**Table 5.2**: *Performance statistics for closest point contouring*

| Grid | Generated Triangles | Classification | Move | Triangulation | Total |
|---|---|---|---|---|---|
| $64^3$ | 25k | 6ms | 6ms | 6ms | 18ms |
| $128^3$ | 108k | 21ms | 15ms | 17ms | 53ms |
| $256^3$ | 401k | 82ms | 68ms | 58ms | 208ms |
| $512^3$ | 1.5M | 270ms | 213ms | 221ms | 704ms |

and non-orientable surfaces (bottom left).

For the bottom right image of Figure 5.6, we have integrated closest point contouring into the open-source MeshLab version of the algebraic point set surfaces (APSS) algorithm by Guennebaud and Gross [GG07], which computes a signed distance field over an unstructured point cloud by projecting each grid point onto a moving-least-squares surface. This demonstrates that existing techniques for the generation of a distance-based implicit surface can easily be modified to generate a closest point-based representation. Figure 5.7 shows the respective input point cloud and compares the triangle meshes generated by the original and the modified algorithm, for which we have merged the triangulation-aware vertex selection and the triangulation pass. Such a one-pass closest point contouring can be a viable substitute for marching cubes. It is just as simple to implement, yet it uses exact vertex positions which are not restricted to the grid edges and it avoids the generation of degenerated triangles, resulting in 18% less triangles in the example.

In Figure 5.8 we compare the reconstruction quality of closest point contouring and dual contouring for a noisy input model and varying grid resolutions. Besides the standard dual contouring which approximates vertex positions by minimizing a quadratic error function, we also include a more noise tolerant variant which uses the centroid of the edge-surface intersections. Especially at the silhouettes it is apparent that the contour quality is further improved by using exact closest points.

In Table 5.2, we present reconstruction times for the results shown in the bottom part of Figure 5.8. The resolution of the defining Cartesian grid and the resulting number of generated triangles are listed in the first two columns. The third and fourth column show the times spent in the classification and move stages of the feature-sensitive closest point selection. The last two columns show the timings for the triangulation stage and the total reconstruction time.

## 5.7 Conclusion and Future Work

In summary, we have presented a very efficient contouring method, which rigorously exploits a closest point representation to extract high quality surfaces. The proposed contouring method opens a number of future research directions, especially in the context of the surface PDEs. When the closest point method should be used to dynamically animate a surface, an explicit

representation helps to accurately track the movement of the surface. After a few time-steps a re-initialization of the triangle mesh is required, however, since otherwise the simulation cannot generate new surface details. Further research is necessary to determine if a weighting scheme based on the Laplacian analysis in Section 5.4.3 could reduce the problems at sharp edges mentioned in Section 3.6.3 (page 60).

# 6

# Conclusion and Future Work

In this thesis, we have presented novel methods for the real-time simulation and rendering of intrinsic fluid dynamics on deformable surfaces, and for the efficient conversation between implicit and explicit surface representations. Our techniques enable the creation of high-resolution appearance and shape variation effects in an interactive environment, which facilitates the creative exploration of fluid flows and wave propagations along complex shapes.

While the focus was primarily on interactive visual effects, the same techniques are applicable to a wide range of problems involving intrinsic partial differential operators on deforming surfaces. Our work describes the first complete GPU implementation of the entire pipeline of the closest point method, and proposes an extension of its core principles to deformable surfaces.

In addition to our simulation techniques, we have also presented a very efficient contouring method, which rigorously exploits the closest point representation to extract high quality triangulations. The generated contours preserve sharp features, are free of degenerated triangles in smooth areas, and their vertices are guaranteed to lie exactly on the zero-crossing of the distance function. The contouring algorithm is fully integrated in our interactive environment, in order to enable its application in future simulation techniques.

The proposed methods open a number of future research directions. By integrating contouring support into the semi-Lagrangian closest point method we plan to pursue two main paths: The up-resing of full three-dimensional simulations and the simulation of intrinsic surface evolution.

The term "up-resing" characterizes a currently very popular approach in computer graphics to increase the apparent resolution of a coarse simulation with a computationally less expensive algorithm. Kim et al. [KTT13], for exam-

ple, follow the widespread paradigm to generate additional high-resolution details only at the surface of a three-dimensional object. They employ the closest point method to simulate wave propagation along a fluid interface, generated by a full three dimensional free-surface flow simulation. While their approach targets computationally expensive visual effects in movies, we think that this is also very promising idea for interactive fluid simulation. By combining our method with a semi-Lagrangian surface tracking technique [BGOS06], we could increase the details of a low-resolution surface, for example, generated by the methods of Crane et al. [CLT07] or Chentanez et al. [CM11]. We expect that such a technique would also show potential for the up-resing of non-interactive simulations.

Our timing statistics in Sections 3.8 (page 72), 4.4 (page 90) and 5.6 (page 108) show that the conversion of a large triangle mesh into a high-resolution closest point surface representation, the solution of nonlinear PDEs on deforming surfaces and the conversion of a closest point representation back into a triangle mesh are already possible at high rates. With the expected increase in processing power of future GPU architectures, it will soon be possible to utilize all three methods in the time integration scheme of an interactive simulation. This could open the door for real-time simulation of intrinsic surface evolution. We plan to deform the surface *itself* based on the solution of an intrinsic simulation *on* the surface. Using reaction-diffusion systems [Tur91] and other techniques from procedural texture synthesis, we hope to be able to simulate the growth of naturally looking objects.

# Bibliography


[AFRS03] Marco Attene, Bianca Falcidieno, Jarek Rossignac, and Michela Spagnuolo, *Edge-sharpener: recovering sharp features in triangulations of non-adaptively re-meshed surfaces*, Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing (Aire-la-Ville, Switzerland, Switzerland), SGP '03, Eurographics Association, 2003, pp. 62–69.

[And95] John Anderson, *Computational Fluid Dynamics*, 1 ed., McGraw-Hill Science/Engineering/Math, February 1995.

[ATBG08] Roland Angst, Nils Thürey, Mario Botsch, and Markus Gross, *Robust and efficient wave simulations on deforming meshes*, Computer Graphics Forum (Proc. Pacific Graphics) **27** (2008), 1895–1900.

[AUGA05] Pierre Alliez, Giuliana Ucelli, Craig Gotsman, and Marco Attene, *Recent Advances in Remeshing of Surfaces*, Part of the state-of-the-art report, AIM@SHAPE EU network of excellence, 2005.

[AW87] John Amanatides and Andrew Woo, *A fast voxel traversal algorithm for ray tracing*, Eurographics '87, 1987, pp. 3–10.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods, 2nd edition*, SIAM, Philadelphia, PA, 1994.

[BCOS01] Marcelo Bertalmío, Li-Tien Cheng, Stanley Osher, and Guillermo Sapiro, *Variational problems and partial differential equations on implicit surfaces*, J. Comput. Physics **174** (2001), no. 2, 759–780.

[BGOS06] Adam W. Bargteil, Tolga G. Goktekin, James F. O'brien, and John A. Strain, *A semi-lagrangian contouring method for fluid simulation*, ACM Trans. Graph. **25** (2006), no. 1, 19–38.

[BKW10] Kai Bürger, Jens Krüger, and Rüdiger Westermann, *Sample-based surface coloring*, IEEE Transactions on Visualization and Computer Graphics **16** (2010), no. 5, 763–776.

[Bri08] Robert Bridson, *Fluid Simulation for Computer Graphics*, A K Peters/CRC Press, September 2008.

[Bur05] Martin Burger, *Finite element approximation of elliptic partial differential equations on implicit surfaces*, Cam report 05-46, University of California, Los Angeles, 2005.

[CIR52] Richard Courant, Eugene Isaacson, and Mina Rees, *On the solution of nonlinear hyperbolic differential equations by finite differences*, Communications on Pure and Applied Mathematics **5** (1952), no. 3, 243–255.

[Cla11] David A. Clarke, *A primer on tensor calculus*, `http://www.ap.smu.ca/~dclarke/home/documents/byDAC/tprimer.pdf`, June 2011.

[CLB+09] Ming Chuang, Linjie Luo, Benedict J. Brown, Szymon Rusinkiewicz, and Michael Kazhdan, *Estimating the Laplace-Beltrami operator by restricting 3D functions*, Proceedings of the Eurographics Symposium on Geometry Processing, 2009, pp. 1475–1484.

[CLT07] Keenan Crane, Ignacio Llamas, and Sarah Tariq, *Real time simulation and rendering of 3d fluids*, ch. 30, Addison-Wesley, 2007.

[CM00] A. Chorin and J. Marsden, *A mathematical introduction to fluid mechanics*, 4th ed., ch. 1.3, pp. 36–44, Springer Verlag, 2000 (en).

[CM11] Nuttapong Chentanez and Matthias Müller, *Real-time eulerian water simulation using a restricted tall cell grid*, ACM SIGGRAPH 2011 papers (New York, NY, USA), SIGGRAPH '11, ACM, 2011, pp. 82:1–82:10.

[CRT04] Ulrich Clarenz, Martin Rumpf, and Alexandru Telea, *Surface processing methods for point sets using finite elements*, Computers and Graphics **28** (2004), no. 6, 851–868.

[DCB+04] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng, *Real-time voxelization for complex polygonal models*, Proceedings of the Computer Graphics and Applications, 12th Pacific Conference, PG '04, 2004, pp. 43–50.

[DCGG11] E. Darles, B. Crespin, D. Ghazanfarpour, and J.C. Gonzato, *A survey of ocean simulation and rendering techniques in computer graphics*, Computer Graphics Forum **30** (2011), no. 1, 43–60.

[DE07] G. Dziuk and C.M. Elliott, *Surface finite elements for parabolic equations*, J. Comput. Math. **25** (2007), 385–407.

[DGW11] Christian Dick, Joachim Georgii, and Rudiger Westermann, *A hexahedral multigrid approach for simulating cuts in deformable objects*, IEEE Transactions on Visualization and Computer Graphics **17** (2011), 1663–1675.

[ED06] Elmar Eisemann and Xavier Décoret, *Fast scene voxelization and applications*, ACM SIGGRAPH 2006 Sketches, ACM, 2006.

[FAW10] Roland Fraedrich, Stefan Auer, and Rüdiger Westermann, *Efficient high-quality volume rendering of sph data*, IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2010) **16** (2010), no. 6, to appear.

[FP02] J.H. Ferziger and M. Perić, *Computational methods for fluid dynamics*, Springer, 2002.

[FZKH05] Z. Fan, Y. Zhao, A. Kaufman, and Y. He, *Adapted unstructured LBM for flow simulation on curved surfaces*, Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, SCA '05, ACM, 2005, pp. 245–254.

[GG07] Gaël Guennebaud and Markus Gross, *Algebraic point set surfaces*, Proceedings of SIGGRAPH, ACM Transactions on Graphics, vol. 26, 2007.

[Gib98] Sarah F. F. Gibson, *Using distance maps for accurate surface representation in sampled volumes*, Proceedings of the 1998 IEEE symposium on Volume visualization (New York, NY, USA), VVS '98, ACM, 1998, pp. 23–30.

[Gre06] John B. Greer, *An improvement of a recent Eulerian method for solving PDEs on general geometries*, J. Sci. Comput. **29** (2006), 321–352.

[Har07] Mark Harris, *Parallel prefix sum (scan) with CUDA*, NVIDIA Whitepaper, April 2007.

[HZQW10] Yi Hong, Dengming Zhu, Xianjie Qiu, and Zhaoqi Wang, *Geometry-based control of fire simulation*, The Visual Computer **26** (2010), 1217–1228.

[JBS06] Mark W. Jones, J. Andreas Baerentzen, and Milos Sramek, *3D distance fields: A survey of techniques and applications*, IEEE Transactions on Visualization and Computer Graphics **12** (2006), no. 4, 581–599.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren, *Dual contouring of hermite data*, Proceedings of SIGGRAPH, ACM Transactions on Graphics, vol. 21, 2002.

[Ju06] Tao Ju, *Intersection-free contouring on an octree grid*, Proc. 14th Pacific Conf. Computer Graphics and Applications PG'06, 2006.

[KBSS01] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel, *Feature sensitive surface extraction from volume data*, Proceedings of SIGGRAPH, ACM Transactions on Graphics, vol. 20, 2001.

[KCY93] A. Kaufman, D. Cohen, and R. Yagel, *Volume graphics*, Computer **26** (1993), no. 7, 51–64.

[KTT13] Theodore Kim, Jerry Tessendorf, and Nils Thuerey, *Closest-Point Turbulence for Liquid Surfaces*, ACM Transactions on Graphics **(in press)** (2013), 10.

[Kuz12] Dmitri Kuzmin, *Introduction to cfd*, `http://www.mathematik.uni-dortmund.de/~kuzmin/cfdintro/cfd.html`, January 2012.

[KW03] Jens Krüger and Rüdiger Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Transactions on Graphics (TOG) **22** (2003), no. 3, 908–916.

[LC87] William E. Lorensen and Harvey E. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, Proceedings of SIGGRAPH, ACM Computer Graphics, vol. 21, 1987.

[LFW07] Kui-Yip Lo, Chi-Wing Fu, and Tien-Tsin Wong, *Interactive reaction-diffusion on surface tiles*, Proceedings of Pacific Graphics, 2007, pp. 65–74.

[LGF11] Michael Lentine, JÃşn TÃşmas GrÃľtarsson, and Ronald Fedkiw, *An unconditionally stable fully conservative semi-lagrangian method*, J. Comput. Physics **230** (2011), no. 8, 2857–2879.

[LPR+09] Florian Liehr, Tobias Preusser, Martin Rumpf, Stefan Sauter, and Lars Ole Schwen, *Composite finite elements for 3D image based computing*, Comput. Vis. Sci. **12** (2009), 171–188.

[LWC05] Lok Ming Lui, Yalin Wang, and Tony F. Chan, *Solving PDEs on manifolds with global conformal parametrization*, VLSM, 2005, pp. 307–319.

[Mac08] Colin B. Macdonald, *The closest point method for time-dependent processes on surfaces*, Ph.D. thesis, Simon Fraser University, August 2008.

[MC04] T.G. Myers and J.P.F. Charpin, *A mathematical model for atmospheric ice accretion and water flow on a cold surface*, International Journal of Heat and Mass Transfer **47** (2004), no. 25, 5483 – 5500.

[MCC02] T. G. Myers, J. P. F. Charpin, and S. J. Chapman, *The flow and solidification of a thin fluid film on an arbitrary three-dimensional surface*, Physics of Fluids **14** (2002), no. 8, 2788–2803.

[Mic12] Microsoft, *Desktop app development documentation (Windows)*, `http://msdn.microsoft.com/en-us/library/windows/desktop`, November 2012.

[MM12] Thomas März and Colin B. Macdonald, *Calculus on surfaces with general closest point functions*, SIAM J. Numerical Analysis **50** (2012), no. 6, 3303–3328.

[MR08] Colin B. Macdonald and Steven J. Ruuth, *Level set equations on surfaces via the Closest Point Method*, J. Sci. Comput. **35** (2008), no. 2–3, 219–240, doi:10.1007/s10915-008-9196-6.

[MR09] ———, *The implicit Closest Point Method for the numerical solution of partial differential equations on surfaces*, SIAM J. Sci. Comput. **31** (2009), no. 6, 4330–4350, doi:10.1137/080740003.

[MS10] Josiah Manson and Scott Schaefer, *Isosurfaces over simplicial partitions of multiresolution grids*, Comput. Graph. Forum **29** (2010), no. 2, 377–385.

[NMZ07] Patrick Neill, Ron Metoyer, and Eugene Zhang, *Fluid flow on interacting deformable surfaces*, ACM SIGGRAPH 2007 posters, SIGGRAPH '07, ACM, 2007.

[NNRW09] Oliver Nemitz, Michael Bangt Nielsen, Martin Rumpf, and Ross Whitaker, *Finite element methods on very large, dynamic tubular grid encoded implicit surfaces*, SIAM J. on Scientific Computing **31** (2009), no. 3, 2258–2281.

[NVI12a] NVIDIA, *CUDA C best practices guide*, `http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`, October 2012.

[NVI12b] _____, *CUDA C programming guide*, `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, October 2012.

[NVI12c] _____, *NVIDIA geforce GTX 680: The fastest, most efficient gpu ever built*, `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`, March 2012.

[NVI12d] _____, *NVIDIA's next generation CUDA compute architecture: Kepler GK110*, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, May 2012.

[OMS98] L Olsen, P K Maini, and J A Sherratt, *Spatially varying equilibria of mechanical models: application to dermal wound contraction.*, Math Biosci **147** (1998), no. 1, 113–29.

[Pan11] Jacopo Pantaleoni, *VoxelPipe: A programmable pipeline for 3D voxelization*, Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (New York, NY, USA), HPG '11, ACM, 2011, pp. 99–106.

[QS11] Jing-Mei Qiu and Chi-Wang Shu, *Conservative high order semi-lagrangian finite difference weno methods for advection in incompressible flow*, J. Comput. Physics **230** (2011), no. 4, 863–889.

[RM08] Steven J. Ruuth and Barry Merriman, *A simple embedding method for solving partial differential equations on surfaces*, J. Comput. Phys. **227** (2008), 1943–1961.

[RWP06] Martin Reuter, Franz-Erich Wolter, and Niklas Peinecke, *Laplace-Beltrami spectra as "Shape-DNA" of surfaces and solids*, Computer-Aided Design **38** (2006), no. 4, 342–366.

[SC91] Andrew Staniforth and Jean Côté, *Semi-Lagrangian integration schemes for the atmospheric models - a review*, Monthly Weather Review **119** (1991), 2206–2223.

[Sch09] Jens Schneider, *Efficient methods for the display of highly-detailed models in computer graphics*, Ph.D. thesis, Technise Universität München, May 2009.

[SJW07] Scott Schaefer, Tao Ju, and Joe Warren, *Manifold dual contouring*, IEEE Transactions on Visualization and Computer Graphics **13** (2007), no. 3, 610–619.

[SS10] Michael Schwarz and Hans-Peter Seidel, *Fast parallel surface and solid voxelization on GPUs*, ACM Transactions on Graphics (Proc. SIGGRAPH Asia) **29** (2010), 179:1–179:9.

[SSS06] John Schreiner, Carlos Scheidegger, and Claudio Silva, *High-quality extraction of isosurfaces from regular and irregular grids*, IEEE Transactions on Visualization and Computer Graphics **12** (2006), no. 5, 1205–1212.

[Sta99] Jos Stam, *Stable fluids*, Proceedings of the 26th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128.

[Sta03] ———, *Flows on surfaces of arbitrary topology*, ACM Trans. Graph. **22** (2003), 724–731.

[SW04] Scott Schaefer and Joe Warren, *Dual marching cubes: Primal contouring of dual grids*, Proceedings of the Computer Graphics and Applications, 12th Pacific Conference (Washington, DC, USA), PG '04, IEEE Computer Society, 2004, pp. 70–76.

[SY04] Lin Shi and Yizhou Yu, *Inviscid and incompressible fluid simulation on triangle meshes: Research articles*, Comput. Animat. Virtual Worlds **15** (2004), 173–181.

[SY05] ———, *Taming liquids for rapidly changing targets*, Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation (New York, NY, USA), SCA '05, ACM, 2005, pp. 229–236.

[Tau95] Gabriel Taubin, *A signal processing approach to fair surface design*, Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, SIGGRAPH '95, ACM, 1995, pp. 351–358.

[TMR09] Li (Luke) Tian, Colin B. Macdonald, and Steven J. Ruuth, *Segmentation on surfaces with the Closest Point Method*, Proc. ICIP09, 16th IEEE International Conference on Image Processing (Cairo, Egypt), 2009, pp. 3009–3012.

[TQZY05] Ping Tang, Feng Qiu, Hongdong Zhang, and Yuliang Yang, *Phase separation patterns for diblock copolymers on spherical surfaces: a finite volume method.*, Phys Rev E Stat Nonlin Soft Matter Phys **72** (2005), no. 1 Pt 2, 016710.

[Tur91] Greg Turk, *Generating textures on arbitrary surfaces using reaction-diffusion*, Proceedings of the 18th annual conference on Computer graphics and interactive techniques, SIGGRAPH '91, ACM, 1991, pp. 289–298.

[TWGT10] Nils Thürey, Chris Wojtan, Markus Gross, and Greg Turk, *A multiscale approach to mesh-based surface tension flows*, ACM SIGGRAPH 2010 papers (New York, NY, USA), SIGGRAPH '10, ACM, 2010, pp. 48:1–48:10.

[Ver67] Loup Verlet, *Computer "experiments" on classical fluids. I. thermodynamical properties of Lennard-Jones molecules*, Phys. Rev. **159** (1967), no. 1, 98.

[WMT07] Huamin Wang, Gavin Miller, and Greg Turk, *Solving general shallow wave equations on surfaces*, Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (Aire-la-Ville, Switzerland, Switzerland), SCA '07, Eurographics Association, 2007, pp. 229–238.

[ZCEP07] Long Zhang, Wei Chen, David S. Ebert, and Qunsheng Peng, *Conservative voxelization*, Vis. Comput. **23** (2007), 783–792.