PARTICLE BASED MODELING AND SIMULATION OF NATURAL PHENOMENA

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Serkan Bayraktar
August, 2010

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Bülent Özgüç (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Uğur Güdükbay (Co-Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Veysi İşler

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Asst. Prof. Dr. Tolga K. Çapın

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Asst. Prof. Dr. Sinan Gezici

Approved for the Institute of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Institute

ABSTRACT

PARTICLE BASED MODELING AND SIMULATION OF NATURAL PHENOMENA

Serkan Bayraktar
PhD in Computer Engineering
Supervisors: Prof. Dr. Bülent Özgüç and Assoc. Prof. Dr. Uğur Güdükbay
August, 2010

This thesis is about modeling and simulation of fluids and cloth-like deformable objects by the physically-based simulation paradigm. Simulated objects are modeled with particles and their interaction with each other and the environment is defined by particle-to-particle forces. We propose several improvements over the existing particle simulation techniques. Neighbor search algorithms are crucial for the performance efficiency and robustness of a particle system. We present a sorting-based neighbor search method which operates on a uniform grid, and can be parallelizable. We improve upon the existing fluid surface generation methods so that our method captures surface details better since we consider the relative position of fluid particles to the fluid surface. We investigate several alternatives of particle interaction schema (i.e. Smoothed Particle Hydrodynamics, the Discrete Element Method, and Lennard-Jones potential) for the purpose of defining fluid-fluid, fluid-cloth, fluid-boundary interaction forces. We also propose a practical way to simulate knitwear and its interaction with fluids. We employ capillary pressure–based forces to simulate the absorption of fluid particles by knitwear. We also propose a method to simulate the flow of miscible fluids. Our particle simulation system is implement to exploit parallel computing capabilities of the commodity computers. Specifically, we implemented the proposed methods on multicore CPUs and programmable graphics boards. The experiments show that our method is computationally efficient and produces realistic results.

Keywords: physically-based simulation, particle system, fluid simulation, neighbor search algorithms, cloth animation, free fluid surface rendering, boundary conditions, mass-spring systems, shared memory parallel computing, GPU, CUDA.

ÖZET

DOĞAL NESNELERİN PARÇACIK TABANLI YÖNTEMLER İLE MODELLENMESİ VE BENZETİLMESİ

Serkan Bayraktar
Bilgisayar Mühendisliği, Doktora
Tez Yöneticileri: Prof. Dr. Bülent Özgüç ve Doç. Dr. Uğur Güdükbay
Ağustos, 2010

Bu çalışma sıvı ve kumaş benzeri, biçimi bozulabilen nesnelerin fizik tabanlı yöntemler kullanılarak modellenmesi ve benzetimiyle ilgilidir. Benzetimi yapılan nesneler parçacıklar kullanılarak modellenmektedir ve nesnelerin birbirleriyle ve çevreleri ile olan etkileşimleri parçacıklar arasındaki kuvvetlerle tanımlanmaktadır. Çalışmada, varolan teknikler üzerine bir çok iyileştirme önerilmektedir. Komşu parçacıkların doğru ve hızlı bir şekilde bulunması parçacık tabanlı benzetim sistemleri için çok önemlidir. Bu çalışmada, paralel hesaplamaya uygun, birbiçimli ızgara üzerinde çalışan, sıralama tabanlı komşu bulma yöntemi önerilmektedir. Sıvı yüzeyinin oluşturulması için önerilen yöntem, varolan yöntemlerden daha ayrıntılı bir yüzey oluşturmaktadır. Bunun sebebi parçacıkların sıvı yüzeyine göreceli konumlarının dikkate alınmasıdır. Sıvı-sıvı, sıvı-kumaş ve sıvı-sınır etkileşimlerini tanımlamak üzere hesaplamalı fizikte de kullanılmakta olan bir çok yöntem araştırılmıştır. Ayrıca örgü tipindeki kumaşları ve bunların sıvılarla etkileşimini benzetmek amacıyla kullanışlı bir yöntem de önerilmektedir. Sıvıların örgüler tarafından emilmesi yüzey gerilimlerini kullanan bir yöntemle benzetilmektedir. Önerilen parçacık sistemi birbirine karışabilen sıvıların benzetmesini de yapabilmektedir. Bu çalışmada anlatılan parçacık tabanlı benzetme yöntemleri, günümüzde yaygın hale gelen paralel bilgisayarlarda (çok çekirdekli işlemciler ve grafik işlemciler gibi) çalışabilecek şekilde uygulanmıştır. Deneyler önerdiğimiz yöntemin işlemsel olarak verimli olduğunu ve gerçekçi sonuçlar ürettiğini göstermektedir.

Anahtar sözcükler: fizik tabanlı benzetim, parçacık sistemleri, sıvı benzetimi, komşu arama algoritmaları, kumaş canlandırma, serbest sıvı yüzeyinin görsel giydirilmesi, sınır koşulları, kütle-yay sistemleri, paylaşımlı bellek tabanlı paralel hesaplama, GPU, CUDA.

Acknowledgement

This thesis would not have been possible without the motivation, guidance, understanding, and help of my supervisors Prof. Dr. Bülent Özgüç and Assoc. Prof. Dr. Uğur Güdükbay. I would like to express my most sincere gratitude to them for always encouraging me, even when I lost my faith.

The jury members, Assoc. Prof. Dr. Veysi İşler, Asst. Prof. Dr. Tolga K. Çapın, and Asst. Prof. Dr. Sinan Gezici, have been kind enough to read and comment on the manuscript. I would like to thank them for their time and kind attention. Their help has been very valuable to improve the quality of this thesis.

I would like to express my deepest gratitude to my wife, İrem. She has been standing by me through the most difficult and arduous periods of my PhD study. I admire her patience and prudence both of which I often exploit to the limits. Deep down, I know very well that this has been a strenuous and exhausting period for her, as well.

I cannot find words strong enough to express my wish that my father would see this day. He was the wisest, most insightful person I have come to know. I always feel the hole his loss has drilled into my heart.

I would like to thank my mother for her support and understanding. She has been wise and patient in her own way and I am sure that she is at least as happy as I am for seeing the end of this story.

My brother deserves my thanks and gratefulness no less than any other. I am a very fortunate person to have him as a brother and friend.

There is not enough space here to name each of my friends who has been supporting me throughout my study. The time I spend with them has been an unending source of joy. I am very happy to know them all.

My graduate study had been financially supported by the Computer Engineering Department of Bilkent University, and The Scientific and Technological

Research Council of Turkey, (TÜBİTAK). I would like to thank both institutions for their support.

During the last, and most stressful months of this period, I have been a member of the The Computer Graphics and Multimedia Systems Group, at University of Siegen. I am grateful to Prof. Dr. Andreas Kolb and other group members for their understanding and support.

Last, but not least, I thank to Romeo and Tarçın, because of whom I know more about patience and unconditional love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the main research areas of computer graphics is modeling and simulation of natural phenomena robustly, realistically, and efficiently. Physically-based simulation techniques aim to achieve this goal by employing physical and mathematical methods explaining the natural phenomena. Physically-based animation techniques have the advantage of relying on fully understood theories and widely researched numerical methods that have been used in computational physics and engineering. Thus, these techniques provide unmatched visual reality and control over animation through simulation parameters.

1.1 Motivation

Particle-based simulations can be considered as a subfield of physically-based animation techniques where objects are represented by a set of discrete points in space having several physical properties, such as mass and velocity. Particles are natural choice for simulating natural phenomena since object interactions in physical world are based on molecular interactions. Given enough number of particles, it is theoretically possible to model and simulate most complex objects and their interactions. In modeling particle-based interaction of complex objects, it is, therefore, enough to define interaction of particle pairs.

The motivation of the research presented in this thesis is to understand, improve, and implement particle-based modeling and simulation systems. Our main concern is visual quality and computational efficiency rather than scientific precision since main application area of the methods we develop are video games and entertainment industry. Thus, although we inspire from the methods developed in the computational physics and engineering, we prefer methods producing visually attractive results without introducing much computational burden. The research presented in this thesis is mostly concerned about simulation of fluids and cloth-like deformable objects and their interactions with each other and their environment.

1.2 Contributions

Defining object behavior in terms of inter-particle interactions requires detecting the set of neighbor particles for each simulation step. Each particle pair in the system is both computationally inefficient and unnecessary. This is because of the fact that most of the particle interactions are defined within a limited distance. To resolve particle interactions in a reasonable speed, it is therefore necessary to determine the set of particles close enough to interact with each other. This neighboring set of particles should be determined at each simulation step. One of the main contributions of this work is to develop a method for determining particle neighbors. The method uses sorting on a uniform grid. The proposed method does not make any assumptions about the number of the particles, resolution of the the grid, or particle interaction threshold distance.

The advent of multicore processors and programmable graphics processors (GPUs) to commodity computers revolutionized computational science. Shared memory parallel programming has been experiencing a revival. Particle simulations can take advantage of this development since particles can be handled independently (in parallel) within a simulation step. This makes particle simulations a perfect candidate to be implemented on shared memory parallel processing architectures. We implement proposed particle simulation system on multicore

processors and programmable GPUs and gain considerable performance improvements.

Another contribution of this work is an improvement of the surface generation algorithm for free fluid flows. Unlike previous methods, the proposed algorithm takes the global fluid structure into account so that small details in the surface are captured. We consider the relative position of each fluid particle with respect to the surface and modify the polygonization algorithm accordingly.

Defining the interaction of a particle with a solid unmovable object such as a wall is crucial for realism and stability of fluid simulations. We propose several improvements for modeling boundary interaction particle-based forces for fluid simulations.

Our simulation system can simulate cloth and knitwear and their interaction with fluids. Although similar to cloth simulation, simulating knitwear has some additional challenges such as rendering fuzzy look of the knitwear and more prominent thickness of the material. We also propose a method based on capillary pressure to simulate the absorption of fluid by knitwear.

The contributions presented in this dissertation have been published in several journals and conference proceedings. Below is the list of publications:

- Serkan Bayraktar, Uğur Güdükbay, Bülent Özgüç, "Particle-based Simulation and Visualization of Fluid Flows through Porous Media," *Journal of Visualization*, To appear.

- Serkan Bayraktar, Uğur Güdükbay, Bülent Özgüç, "GPU-Based Neighbor-Search Algorithm for Particle Simulations," *Journal of Graphics, GPU, and Game Tools*, Vol. 14, No. 1, pp. 31-42, AK Peters, Ltd., 2009.

- Serkan Bayraktar, Uğur Güdükbay and Bülent Özgüç, "Practical and Realistic Animation of Cloth," in *Proceedings of the IEEE 3DTV-CONFERENCE: Capture, Transmission and Display of 3D Video*, Kos, Greece, May 2007.

- Serkan Bayraktar, Uğur Güdükbay and Bülent Özgüç, "Sıvı, Kumaş, ve Katı Cisim Etkileşimlerinin Bilgisayar Grafiği İçin Modellenmesi (Modeling Interaction of Fluid, Fabric, and Rigid Objects for Computer Graphics)" (in Turkish), IEEE Sinyal İşleme ve Uygulamaları Kurultayı (SİU 2006), Antalya, Turkey, April 2006.

1.3 List of Symbols

Table 1.1 gives a listing of the mathematical notations used in the thesis.

Notation	**Denoted expression**	**Description**
$\mathbf{r}$ *and* $\mathbf{x}$	coordinate	2D or 3D coordinates of a particle.
$\mathbf{f}$	force	force acting on particles
$\mathbf{v}$	velocity	velocity vector of a particle.
$\mathbf{a}$	acceleration	acceleration vector of a particle.
$\mathbf{n}$	normal	normal vector to fluid or object surface at position of particle
p	pressure	pressure associated with fluid particles
ρ	density	density of a fluid particle
μ	viscosity	viscosity of a fluid particle
V	volume	fluid volume associated with fluid particle
m	mass	mass of a particle
c	color field	color field of a particle in SPH formulation
h	kernel radius	kernel radius of SPH formulation
t	time	
W	kernel	kernel function of SPH formulation
ϕ	porosity	porosity of a rigid or deformable object
k	permeability	permeability of a rigid or deformable object
s	saturation	saturation of a rigid or deformable object

Table 1.1: The listing of mathematical notations used in the thesis. Bold-faced letters are used to represent vectors and regular letters are used to represent scalar quantities. Particles are symbolized with letters i, j, and k.

1.4 The Organization of the Thesis

This thesis is structured as follows: Chapter 2 gives an extended overview of the literature of physically-based simulation of natural phenomena, specifically fluid and cloth simulation. We also give the state of the art in GPU implementation of particle-based simulation methods. Chapter 3 details fundamentals of computational fluid dynamics and theoretical background on particle-based fluid simulation. In Chapter 4, we provide details of the mass-spring method which is used in simulating cloth-like deformable object. We also describe method we propose to model and render knitwear. In Chapter 5, we underline several of the practical issues (e.g. surface generation, neighbor search, surface tension implementation etc.) of implementing a particle-based simulation system.

Chapter 6 explains our implementation of the proposed method on GPUs. Several points should be taken into account when porting a CPU-based algorithm to a GPU and this chapter provides the necessary details. In Chapter 7 we present several of our simulation examples. We underline several important characteristic of each example such as the number of particles, simulation speed, and memory consumption of the simulation. Each example is supplemented with one or several still images from the simulation. Chapter 8 is our conclusion chapter.

Chapter 2

Related Work

2.1 Fluid Simulation

The simulation of fluid bodies has been a research topic for computer graphics community for nearly two decades. Early attempts of fluid animation in computer graphics community are mostly involved with simulating the surface behavior of liquid bodies. These early models define the surface as a parametric function evolves in time.

Perlin [113] propose a simple stochastic model which is used for rough ocean surfaces. In his model, he uses normal perturbation instead of actually modifying the water surface position. He employs several superimposed spherical wave fronts that were distributed randomly. Waves of greater realism were created by using a random spatial frequency. Max [89] uses a procedural model to render fluid surface. His algorithm is based on analytic formulas and he employs traveling sinusoids to simulate waves. Fournier and Reeves [42] use a parametric wave function that was based on the model by Gerstner [45] and Rankine [119]. Their model does not involve mass transport and is derived from the equations for deep water, small amplitudes waves. The system is able to simulate wave refractions and wind effects. O'Brien et al. [110] use a height field based approach to model fluid surface. In an attempt to simulate splashing water behavior, they

also incorporate a particle-based model into their system. Kass and Miller [65] approximate 2D shallow water equations to simulate dynamic height field surface which is in interaction with static ground objects. Their model is able to describe wave refraction, reflection, and net transportation of the liquid. They represent the water surface by a height field and assume that the vertical component of the velocity of the water particles can be ignored. They also assume that the horizontal component of the velocity of the water particles is constant.

A more realistic simulation of fluid phenomena requires solving the partial differential equations (PDE) of motion based on dynamics. For viscous, incompressible, and Newtonian fluids (fluids that have a constant viscosity at all shear rates at a constant temperature and pressure), these equations are called Navier-Stokes equations [43]. Most of the early attempts to solve Navier-Stokes equations for computer graphics purposes employ an Eulerian approach. In Eulerian methods the equations governing the fluid behavior are solved in a (usually) regular grid.

Gamito et al. [44] employ vorticity and velocity field to simulate behavior of turbulent gaseous fluids in two dimensions. They use particles to transport the vorticity and a uniform grid to compute velocities and displacements of particles. Their method is able to simulate turbulent gaseous fluid in a relatively realistic and efficient manner. Chen and Lobo [16] are the first to use Navier-Stokes in graphics literature. They use the pressure from 2D solution of the Navier-Stokes equations to improve height field approach and obtain third dimension. Chen et. al. [17] use the Navier-Stokes equations in order to model animated water surface from the pressure term. They employ a finite-difference solution technique in order to solve the Navier-Stokes equations numerically.

Foster and Metaxas [40, 41] are the first to introduce 3D Eulerian form of the Navier Stokes equations in the computer graphics community. Foster and Metaxas [40] apply the Marker-And-Cell (MAC) approach of Harlow and Welch [52] to simulate water. Their work is able to mimic realistic fluid behaviors such as splashing, pouring, breaking weaves and simple interaction with floating rigid objects that are impossible to simulate by height fields approach.

Stam [128] introduces the so–called "semi–Lagrangian" numerical methods which is unconditionally stable, thus allowing use of large time steps. Stam replaces the finite difference scheme of Foster's model with a semi-Lagrangian method so that larger time steps are possible. Stam also employs the pressure projection method instead of Foster's relaxation scheme to achieve zero divergence. Fedkiw and Foster [39] introduce a new hybrid method to capture small details in fluid's surface. Their method employs the level set approach and massless marker particles that are placed around the fluid's surface. They also replace the forward Euler convection calculations with a semi-Lagrangian approach and use conjugate gradient method to enforce incompressibility.

Enright et al. [33] improve the level–set based surface generation method to ensure mass preservation and photo–realistic fluid effects. Carlson et al. [13, 14] use Eulerian grid based methods to model melting, and two way rigid–fluid interaction. Guendelman et al. [49] use a complex surface traction method implemented in an octree grid [86] so that fluid interaction with thin rigid objects and deformable bodies such as cloth is possible. Song et al. [127] propose the derivative particle method where they implement the non–advection part of the simulation in a conventional Eulerian grid and use a Lagragian scheme for the advection part. Goktekin et al. [47] employ an Eulerian solution of Navier-Stokes equations coupled with the terms to define elastic, and plastic forces to mimic behavior of viscoelastic fluids, such as mucus, clay, toothpaste, etc. They extend the well known staggered grid method to include the terms of strain tensors. They use a fine detail grid for the the level-set method. Feldman et al. [34] simulate the fluid behavior inside of a deforming mesh. They use the Eulerian method in which the fluid velocity is computed with respect to a fixed coordinate system and apply a time-varying discretization of the fluid properties to add the effect of the deforming mesh.

Fedkiw et al. [49] simulate the interaction of smoke and water with thin deformable and rigid shells. Their fluid model is Eulerian while cloth model is Lagrangian. In order to prevent the leaking of fluid across the thin objects, they utilize a ray casting scheme where the space is divided into three regions with respect to the position of the triangles constituting the thin object. Mosher et al. [121]

propose a method to incorporate Eulerian-based fluid and Lagrangian-based rigid body simulations. They base their method on the principle that the momentum should be conserved on the fluid-rigid boundary. They enforce no–slip boundary condition. Kim et al. [68] improve the level-set method to capture the small details of splashing water. They convert the marker particles which escape from the main fluid body into fluid particles. These particles are used as seeds to produce the subcell–level detail. Volume loss is estimated and distributed to water particles. Lasasso et al. [87] employ conventional Eulerian grid-based methods to simulate solids, fluid, and gases. Their fluid model is based on vorticity confinement and particle level-set methods. They can simulate melting and burning of natural material, such as ice cubes and paper.

Wojtan et al. [153] propose a FEM (finite element method) based technique to simulate realistic behavior of highly viscous fluids, and deformable models. Their method behaves well in scenarios where thin strands and sheets appear. Thürey et al. [140] propose a fluid animation control method where small scale details are preserved with control forces which are represented by control particles. Their control method can be used both in Eulerian and Lagrangian fluid simulation paradigms. Brochu et al. [12] develop an Eulerian method to simulate inviscid fluids. They employ semi-Lagrangian advection and an embedded-boundary finite volume pressure projection. Instead of using an explicit surface tracking method, they couple the simulation itself to an existing surface tracking method. This enables them to visualize arbitrary thin features and avoid artifacts arising from the resolution mismatch between the simulation and surface. Wojtan et al. [152] propose a mesh-based surface tracking method. Their method is designed to preserve fine details and adjust to the topology of the fluid body. Thanks to their local convex-hull-based correction method, their method does not require the re-sampling. Thürey et al. [142] present a method to simulate surface tension derived flows by employing triangular mesh-based surface representation. They employ a two layered simulation system where the first layer simulates the surface tension and the second layer simulates sub-grid scale wave details. Their method is able to simulate complex phenomena associated with strong surface tension. Lentine et al. [81] propose an Eulerian fluid simulation technique where large

scale fluids are simulated in coarse grid without losing details. This effectively reduces required computational time for large scale fluid simulation. Long et al. [83] propose a fast method for fluid simulation where they use sine and cosine transforms instead of more expensive Fourier methods. Kang el al. [63] propose a method to simulate both miscible and immiscible fluids. They combine two fluid representation schema, level set functions and volume fractions. Their method is able to simulation several mixing fluid bodies.

Alternative to Eulerian, grid-based methods, modeling and simulation of natural phenomena can be achieved by particle-based modeling and simulation methods. In particle-based methods, simulated mass is represented by a set of particles that carry several physical attributes with them. They interact and effect each other as the simulation evolves.

In one of the earliest works, Reeves [120] uses particles to model fuzzy objects (e.g. fire works). Large amount of particles represent the cloud's volume which is able to move, and change form. Miller et al. [96] utilize pairwise particle interactions to model viscous fluids. They define a term *globule* to refer to an element of the connected particle system. Particularly, they are interested in modeling soft collisions between the globules to avoid rigid stacking problems. Terzopoulos et al. [136] model melting objects with interacting particles which are connected by springs whose constants are modified as the object changes its phase. Tonnesen [143] incorporates a discrete form of heat transfer equation into inter–particle force equations to simulate the change in particle positions due to thermal energy. Premoze et al. [115] use particles to simulate incompressible fluids where they ensure incompressibility by using the Moving Particle Semi–Implicit (MPS) method. The MPS method uses weighted averaging to determine fluid parameters in space positions. Kruger et al. [73] use graphics hardware (GPU) to achieve interactive rates when simulating large particle sets. To render transparent particles correctly, they employ bitonic merge sort which suits the requirements of shader programming well.

Most of the more recent particle-based methods employ the Smoothed Particle Hydrodynamics (SPH) [46, 97] paradigm, which is originally designed to be used

in computational astrophysics. Desbrun et al. [28] are among the first researchers to use SPH in computer graphics context. They model a particle-based simulation system with SPH to model highly deformable objects. They investigate the issues like surface determination, adaptive time integration and fast neighbor search algorithm. Stora et al. [130] use SPH and heat transfer equations to simulate lava flows. They choose a procedural texture-based rendering system to visualize the lava flows. They employ a grid to make fast neighbor detection possible. They exploit the topology of lava spread and construct a grid which is large in vertical range but small in height. Müller et al. [101] utilize SPH version of the Navier–Stokes equations to model incompressible fluid at interactive rates. They simulate surface tension using a force proportional to the curvature at each particle location and pointing into the fluid body along the normal vector of the surface. Müller et al. [103] later improve their system to simulate interaction of fluids with deformable solids. They place virtual particles on solid surfaces to handle fluid-solid interaction on a particle-to-particle base. They also model fluid-fluid interactions using SPH [104].

Hadap and Magnenat–Thalmann [50] couple SPH with strand dynamics to simulate hair–hair, and hair–air interactions. Their formulation is able to model hair strands as continuum while retaining individual structure of each strand. Clavet et al. [23] use SPH to simulate viscoelastic fluids such as toothpaste or mud. To enforce incompressibility and avoid particle clustering, they employ a double density relaxation procedure. An attraction term is added to force computations to simulate particle stickiness. They use marching cubes to tesselate the free fluid surface. Kipfer et al. use [70] GPU based data structure and SPH fluid simulation method to model and render interactive simulation of rivers. To detect particle proximities, they use a linear data structure where a virtual grid is used to hash particles. They assert that their method provides a better performance than an octree based collision detection algorithm. Solenthaler et al. [126] simulate fluid, deformable bodies, and melting and solidification by using SPH and elastic–plastic model. Their system is able to simulate flexible, deformable objects, melting, merging, splitting, and solidification behaviors. To reduce visual artifacts in surface generation, they consider the movement of the center of

mass.

Becker and Teschner [6] propose a variant of SPH method to simulate weakly compressible fluid flows. The SPH method they employ is based on the Tait equation. In addition, they propose a surface tension algorithm to be able to visualize fluid surfaces with high curvature. Cleary et al. [24] utilize SPH to model bubble and froth generation and their coupling with the fluid body. They model bubbles as discrete spherical bodies and couple them with the particle-based fluid simulation system. Lasasso et al. [88] propose a two–way coupled simulation system where dense liquid volumes are simulated using the particle level set and diffuse regions such as mixture of air and sprays are simulated by SPH. Thürey et al. [141] couple shallow water simulation to the SPH method to simulate bubble and foam effects in real-time. Spherical vortices are used to generate flow field around the SPH-based bubbles. They add surface tension to SPH to be able to simulate bubbles on the water surface.

Müller et al. [102] propose a surface generation and rendering technique for rendering the resulting point clouds of SPH method. In their method, they first setup a regular depth map and consider silhouettes. A 2D triangle mesh is constructed by a modified version of the Marching Squares algorithm. The mesh is then transformed back onto world space and rendered by employing several rendering enhancements such as reflections, refractions, and other effects. Laan et al. [75] develop a method for rendering the particle clouds. Their method uses surface depth and thickness. After surface depth is smoothed, a dynamic noise texture is generated on the surface of the fluid. A final step combines the generated texture, smoothed surface depth, the noise texture and the image of the background objects. The whole rendering method is implemented on graphics hardware. Hoetzlein et al. [56] present a rendering technique for stream shaped particle-based fluid flows. Instead of polygonizing the fluid surface by the Marching Cubes algorithm, they wrap fitted and deformed cylinders around the flow streams.

Hong et al. [57] propose a hybrid method to simulate bubble behavior. They combine Eulerian grid-based fluid simulator (to simulate large fluid volumes)

with the SPH method (to simulate bubbles). By using two way coupling, they are able to mimic realistic bubble and foam behavior. Lenaerts et al. [79] use SPH to simulate the behavior of fluids flowing through deformable porous materials. Their system is able to simulate permeability, abortion, and two–way coupling between the fluid and wet material. Because of the macroscopic scale of their pore modeling, the number of computational elements is low. Later, Lenaerts et al. [80] use a very similar technique to simulate mixing of fluids and granular materials where the discrete element method is employed for simulating granular materials. Lee et al. [78] incorporate the SPH and grid-based fluid simulation method to capture flow details even in a coarse grid. They simulate escaped particles with SPH and merged particles with level set method. Their method is able to simulate air bubbles where large bubbles are modeled with level sets and small bubbles are modeled with SPH.

Bell et al. [8] design a particle-based system to simulate granular materials. They model granular materials as collections of non-spherical particles. Their method is able to simulate granular material's interaction with rigid bodies, splashing and avalanches. Becker et al. [7] propose a new predictor-corrector scheme based method to implement one–way and two–way coupling of fluid with rigid bodies. For fluid body simulation they use a corrected SPH and Tait equation. Their system is able to simulate realistic drag and buoyancy effects. Solenthaler et al. [125] extend Smoothed Particle Hydrodynamics method with a prediction-correction schema. They actively update the fluid pressure to obtain a certain density. Their method results in a incompressible fluid where they do not have to solve a Poisson Equation. He et al. [54] propose a SPH-based method for fluid interaction with complex polygon boundaries. They employ adaptive SPH method where they redefine the rule of particle adaptation according to the complexity of the scene. For handling collisions, they propose a voxelization-based collision detection algorithm. Krištof et al. [72] couple Eulerian-based physical erosion approach and SPH to simulate realistic erosion of 3D terrain. They also propose a new donor-acceptor schema for sediment advection.

2.2 Cloth Simulation

The early works on cloth modeling and simulation focus on geometrical methods to mimic cloth behavior such as wrinkles and draping. Even though they are able to simulate some cloth behavior, they are far from creating a complete system of cloth simulation.

Weil et al. [151] employs a geometrical method for cloth modeling. They represent hanging cloth as a grid of points. Simulation is done by fitting catenary curves between hanging or constraint points. Although this method is very fast since it does not involve heavy numerical computations, it can only simulate hanging cloth. As a geometrical method, it is unable to model the properties of real cloth behavior. Agui et al. [2] develop a geometrical method in order to model a sleeve on bending arm. They represent the cloth as a hollow cylinder consisting of a series of circular rings. The difference between the curvatures of the inner and outer parts of the bent sleeve forms folds. This method, although providing satisfactory results, can be applied only to a folding sleeve. Ng et al. [105] aim to create an animation tool that would quickly produce clothed objects. In order to achieve this goal, they associate the cloth layer with the shape of the skin layer. They devise an algorithm to generate folds by using gaps between the cloth and the skin layer to achieve a realistic appearance.

Physically-based cloth modeling and animation methods offer more realism and ease of modeling than geometrically-based modeling methods. In these methods, cloth is represented as triangular or rectangular grid composed of a finite number of mass points. Some of the physically-based methods are the algorithms that calculate the energy of the whole cloth and determine the shape of the cloth by minimizing this energy while some others are force-based. In these methods, forces between points are represented as differential equations and a numerical solution is utilized in order to find the positions of points.

Feynman [37] models cloth as a grid and claim that most realistic shape of the cloth is obtained when the energy state is minimized. He utilizes the theory of elastic plates and uses the steepest descent method to find the energy minima.

Employing this method, Feymann simulates hanging cloths and cloths dropped over a sphere. In their pioneering works, Terzopoulos et. al. [135, 134] define deformable objects as a continuum. Deformable characteristics of the models are calculated by using elasticity and plasticity theories. In this modeling schema, potential energy functionals are used to represent elastic properties of models. Stiffness matrices are employed to store elastic characteristics of the material. They utilize finite element method and energy minimization techniques for numerical solution. The equations of motion are expressed in Lagrange's form which incorporates the mass density, damping density, net external force acting on the deformable body, and potential energy of the elastic deformation.

Sakaguchi et al. [123] develop a technique where the cloth is represented as a grid and use Newton's law of dynamics. Internal force is composed of spring forces, forces due to viscosity, and forces due to plasticity. They use Euler's method as their numerical solver in order to obtain the velocity and position of the cloth. Collision detection speed is improved by using a finite bounding volume hierarchy and collisions are resolved by considering conservation of momentum. Lafleur et al. [76] develop a technique to deal with the collisions occurring between body parts and cloth. In their model, they create a force field around the cloth model to prevent collisions. Thus, when a point enters the force field, it is pushed away by a repulsive force. This model is improved later by Yang et al. [155] to speed up collision detection. To detect self collisions, which are much more expensive than collisions with outside objects in terms of detection, they employ a hierarchy of bounding boxes encapsulating the cloth polygons.

Volino et al. [146] employ Newtonian dynamics and elasticity theory to simulate deformable objects. They model the cloth as a set of triangles. The algorithm calculates interparticle constraints and external constraints, and then detects and resolves collisions by using the principle of the conservation of momentum. The interparticle constraints are due to the elasticity theory of an isotropic surface and external constraints are due to gravity, viscous air damping, and wind. Volino et al. [149] introduce a method to improve collision detection. They exploit the geometrical regularity of the cloth surface and construct a hierarchy to represent this regularity. They improve the collision detection process such that the time to

detect collisions is proportional to not the total number of the cloth polygons, but to the number of collisions. Provot [117] utilize a mass-spring model to simulate cloth. At each time step, after finding the net force on each mass point, Provot uses the Euler's method in order to find the velocity and position of each mass point.

Kunii et al. [74] propose a hybrid model that has two steps. The first step is a physical simulation where the cloth is represented with a mass-spring network. Two kinds of energies are defined: metric and curvature. A gradient descent method is utilized in order to find the energy minima and obtain the shape of the cloth. Then, the singularity theory is used to characterize the resulting wrinkles. The second step is based on a geometrical technique in which surfaces are constructed between the characteristic points to form the wrinkles. After adding wrinkles, they again apply energy minimization to the garment.

Baraff et al. [5] address cloth–to–cloth collision problem. They propose a method based on a history–free, global intersection analysis collision detection. They also supply a solid–to–cloth collision resolution method called collision fly-papering. Bridson et al. [10, 11] propose a collision handling method. Their method can handle self intersections, provide stable folding and wrinkling and consider kinetic and static frictions. They completely separate internal cloth dynamics computation from the collision handling so that the method can be used with any cloth dynamics schema and any numerical integration method. Volino et al. [147] use a viscous damping to enhance stability of the implicit midpoint method, which is simple to implement and provide small time steps. Since the implicit midpoint method is not dependent on the history of simulation, it provides a better robustness on collisions and discontinuous effects.

Choi et al. [22] propose a semi-implicit cloth simulation technique to handle the post-buckling instability. They use a mass-spring network and define two types interaction between neighbor particles, one for stretch and shear resistance and one for flexural and compression resistance. They predict the static post-buckling response with the assumption that the cloth passes the unstable post-buckling stage and reach a stable state. Boxerman et al. [9] develop an adaptive

implicit-explicit method that increases the sparsity of the system. By using this method they are able to simulate decomposing cloth mesh. Oh et al. [111] propose a semi-implicit method where damping forces generated to ensure stability are computed only for internal deformations. They argue that their method does not create excessive damping artifacts. Oh et al. [112] develop a multi–grid method to simulate cloth meshes of large size. To adapt multi–grid method to physically–based cloth simulation, they ensure conservation of cloth's physical properties through the levels. They achieve about 30% speedup by removing redundant matrix–vector multiplications.

Goldenthal et al. [48] address the over–stretching of cloth mess which is an intrinsic problem of mass–spring cloth models. They propose using Constrained Lagrangian Mechanics and a projection filter to avoid over–stretching. Volino et al. [148] aim to simulate nonlinear tensile behavior and large deformations of cloth materials. They use strain–stress curves, elasticity and viscosity computations. They choose to compute forces on mass points to keep run time complexity low. Feng et al. [35] develop a method to achieve real-time realistic cloth simulation with complex deformations. Their method relies on data-driven models to transform low-quality simulated deformations to high-quality dynamic deformations. Wang et al. [150] propose a data-driven method which is implemented on graphics hardware. They aim to achieve interactive speeds for complex cloth simulation with wrinkles. Their method interpolates a precomputed wrinkle database in accordance to coarse cloth simulation. Although their method does not always produce physically correct small details, it captures most of the wrinkle structure correctly and achieves interactive simulation rates. Aguiar et al. [27] present a learning-based approach to cloth simulation on human models. Their method can simulate several types of cloth on human models.

Eberhardt et al. [32, 93] use a particle-based system a simulate behavior of woven and knitted cloth. They model knitwear thread as a chain of bounding points. For force computations they use a linear spring–based approach. Nocent et al. [108] use a spline-based model to simulate cloth plane and project the deformations into yarn control vertices. The stitches are modeled by defining contact constrains. Their main contribution is to present a solution to reduce the

linear equation system size which is increased by adding the contact constraints. Chen et al. [18, 154] exploit the repetitive structure of knitted cloths to simulate and render knitwear. They use spring forces and force field model for a realistic animation. To render knitted cloth, they define lumislice which is a single cross-section of yarn. Lumislice is used to determine the radiance from a yarn cross-section. Kaldor et al. [62] model each yarn as an inextensible, yet otherwise flexible, B-spline tube. Stiff penalty forces and rigid-body velocity filters are used to simulate knitted cloth behavior. They render the knitwear by Chen's lumislice method. Kaldor et al. [61] aim to solve collisions on yarn-based cloth simulations. Their method relies on approximate penalty-based contact forces. They compute an exact collision response at one time step and use a rotated linear force model to approximate response forces of nearby deformations.

2.3 Hardware Accelerated Physically-Based Simulation

Because of its intrinsically parallel nature, particle simulation systems have been one of the first simulation methods to be implemented on GPUs. Harris et al. [53] simulate clouds dynamically on graphics hardware. They use tiled 2D textures to store 3D data to ensure scalability. A single time step of the cloud simulation is spread to several animation frames. Latta [77] presents a technique where a particle system is simulated fully on GPU. Attributes of particles such as position, velocity, and acceleration are stored on 2D textures which are updated by fragment and vertex shaders. The proposed method uses odd-even merge sort to sort particles so that they alpha blend correctly. Their method does not deal with inter-particle collisions. Kipfer et al. [69] present a sorting based inter-particle collision detection system. Their method employs bitonic sort that uses grid cell number as sorting keys. Their system has the weakness of being able to detect a limited number of inter-particle collisions within a specific grid cell. Purcell et al. [118] also present a sorting based algorithm to determine k-nearest neighbors

of a photon (or a particle). They utilize vertex shaders to perform scatter operations, prepare a complex grid map and use stencil buffer for dealing with multiple photons residing in the same cell.

Venetillo et al. [144] implement an auxiliary array on GPU to detect inter-particle collisions. Their algorithm makes several rendering passes to deal with multiple particles mapped into the same grid cell. Kolb et al. [71] use fragment shaders to simulate and render dynamic particle system. Their system is able to handle collision of particles with objects of arbitrary shapes. The outer shape of objects is represented by depth maps that store normal vectors and distance values to handle collisions correctly. Harada et al. [51] present a SPH-based fluid simulation system. Their system uses bucket textures to represent a 3D grid structure and make an efficient neighbor search. One limitation of their system is that it can only handle up to 4 particles within a grid cell. When simulating nearly incompressible flows, the particle density per cell may be higher than the particle density at the rest state.

Hegeman et al. [55] implement a quadtree data structure on GPU to determine inter-particle collisions. To improve tightness of the bounding tree, they re-order particles by using bitonic sort.

Iwasaki et al. [58] propose a splatting-based rendering method to render fluid surfaces completely on GPU. They construct a grid over the simulation space and compute a density value on each grid point by accumulating densities of fluid particles. The iso-surface is extracted and rendered by surfels. Their method can handle refraction, reflection, and caustics.

Chapter 3

Fundamentals of Fluid Dynamics

In order to construct a fluid simulation system, it is necessary for one to be familiar with the fundamental concepts of fluid dynamics. In this chapter, we overview most relevant of these concepts.

The *density* of a fluid, denoted by ρ, is defined as its mass per unit volume. It is defined at each point of the fluid, thus it can be written as

$$\rho \equiv \rho(x, y, z, t), \tag{3.1}$$

where x, y and , z are coordinates of the point and t stands for time. Since density is a scalar quantity, the field defined by Equation 3.1 defines a scalar field. Unlike gases, for fluids variations of pressure and temperature has a very slight effect on density.

Similarly, the *velocity* of a point in a fluid at a given instant is a function of the coordinates of the point and time. That is:

$$\mathbf{v} \equiv \mathbf{v}(x, y, z, t). \tag{3.2}$$

Since velocity is a vector quantity, the field defined in Equation 3.2 is a vector field [43].

Pressure of a fluid is defined as the normal force exerted on a unit area of a surface fully immersed in the fluid. It is created by the collisions of the fluid

molecules into the surface and denoted by p. Another important fluid property is *viscosity*, which is designated by μ and basically determines the *fluidity* of the fluid. Viscosity can be defined as the resistance of a fluid body to deformations due to shear forces. It is the internal friction of fluid which resists movement against a solid surface or other layers of fluid. It can also be thought as resistance of fluid to flowing. Viscosity of a fluid is highly dependent on the temperature of the fluid. Usually, viscosity gets lower as the temperature of a fluid increases.

It is possible to classify fluids with respect to their parameters. An important class of fluids is defined in accordance with viscosity. A fluid that has a constant viscosity at all shear rates at a constant temperature and pressure is called a *Newtonian fluid*. In other words, for Newtonian fluids the shearing stress is linearly related to the rate of shearing strain (angular rate of deformation) [156]. Most common fluids and gases including water, air, and gasoline are Newtonian fluids under normal conditions. In this thesis, all fluids are considered to be Newtonian unless stated otherwise.

Another classification of fluids can be done in terms of the characteristics of the flow. When the effect of the viscosity is assumed to be zero ($\mu = 0$) then the flow is termed an *inviscid flow*. Otherwise (when $\mu \neq 0$) the flow is said to be a *viscous flow*. In reality inviscid flows do not exist and considered only for the sake of simplification of the analysis.

The most important implication of the viscous flow is that the fluid in direct contact with a solid boundary has the same velocity as the boundary itself. The fluid velocity at a stationary solid surface in a moving fluid is zero. Since the bulk fluid is in motion, velocity gradients and shear stresses must be present in the flow. These stresses affect the fluid motion.

Another important concept concerning fluids is *bulk modulus* which describes the *compressibility* of the fluid. Bulk modulus is denoted by K and defined by the following Equation:

$$K = -\frac{\partial p}{\partial V}, \tag{3.3}$$

which relates the change in fluid's volume to change in pressure. The minus

sign designate that the relation is reverse. For all practical purposes, fluids are considered incompressible meaning that their density stays constant as pressure changes. This is not the case for gases which are compressible. The rate that a local change in pressure propagates within the fluid body is called the *acoustic velocity* or the *speed of sound*. It is an important property for defining a specific fluid and it can be expressed as follows:

$$c = \sqrt{\frac{\partial p}{\partial \rho}} \tag{3.4}$$

where p is the pressure and ρ is density.

The motion of a Newtonian, viscous, and incompressible fluid at any point of a flow can be described fully by a set of non-linear equations known as the momentum or Navier-Stokes equations and an equation concerning the conservation of mass.

The Navier-Stokes equations are derived from the Newton's Second Law, which states that the momentum is always conserved. The Navier-Stokes equations account for all possibilities of momentum exchange within the fluid. For an incompressible fluid in three dimensions these equations are as follows:

$$\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}\right) = -\frac{\partial p}{\partial x} + \rho g_x + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \tag{3.5}$$

$$\rho\left(\frac{\partial v}{\partial t} + \frac{\partial vu}{\partial x} + \frac{\partial v^2}{\partial y} + \frac{\partial vw}{\partial z}\right) = -\frac{\partial p}{\partial y} + \rho g_y + \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) \tag{3.6}$$

$$\rho\left(\frac{\partial w}{\partial t} + \frac{\partial wu}{\partial x} + \frac{\partial wv}{\partial y} + \frac{\partial w^2}{\partial z}\right) = -\frac{\partial p}{\partial z} + \rho g_z + \mu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right), \tag{3.7}$$

where u, v, and w are velocities in the x, y, z directions respectively, p is the local pressure, g, gravity, and v is the kinematic viscosity of the fluid. The left hand side of the equations account for changes in velocity due to local fluid acceleration and convection. The right hand side of the equations define acceleration due to the force of gravity, acceleration due to the local pressure gradient, and drag

due to the kinematic viscosity. The Navier-Stokes equations do not account for the conservation of mass. Conservation of mass for a incompressible flow of a Newtonian fluid can be incorporated into the system by the following equation:

$$\nabla.V = 0, \tag{3.8}$$

which is equal to

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0, \tag{3.9}$$

where u, v, and w are as above. Since we have four unknowns (u, v, w, and p) and four equations (equations 3.5, 3.6, 3.7, and 3.9), the problem is well-defined in mathematical terms. However, since the Navier-Stokes equations are nonlinear, second order partial differential equations, the exact mathematical solution does not exist with an exception of a few very simple cases.

When a rigid body is completely or partially submerged in a fluid, the resultant fluid force acting on the body is called the *buoyant force*. According to Archimedes' principle the buoyant force acting on a partially or fully submerged object is equal to the mass of the fluid displaced by the object.

3.1 Computational Fluid Dynamics

Solving the equations of fluid dynamics in order to simulate fluids in complex environments has been focus of research in engineering [60, 38]. These numerical models can collectively describe every aspect of fluid motion effects. However they are not always suitable for computer graphics purposes. This is mostly because of the fact that each technique is derived for a specific class of problem and they are useless in generic situations. Another reason is that these techniques are computationally expensive for interactive applications.

There are two major approaches in computational fluid dynamics to solve fluid flow numerically: the Eulerian approach and the Lagrangian approach. In the *Eulerian method*, the fluid motion is given by completely specifying the properties (pressure, density velocity, viscosity, etc) of fluid flow as a function of space and

time. The information about the flow is obtained from specific and discrete points in space by using these functions.

In order to solve the fluid system defined by Equations 3.5, 3.6, 3.7, and 3.9, a finite representation of the environment is needed. This can be achieved by employing a uniform grid structure. In this representation, the computation domain is modeled by a set of cells aligned with a Cartesian coordinate system. Velocity, and pressure are defined at the center of each cell and supposed to be constant throughout the cell volume [128]. Another option is to define velocity at the boundaries of each cell and calculate the values of velocity and pressure of any point of the environment by linear interpolation [40]. In any case, an explicit finite difference approximation of Navier-Stokes equations are employed to resolve the system in a time-dependent fashion. The velocity and pressure values are updated according to the divergence value computed in each direction. In terms of numerical integration, there are two options: explicit and implicit integration. Implicit integration approach allows stable simulations with large time steps. The resolution of the Eulerian grid determines the degree of trade off between the realism and computational efficiency.

The second approach in computational fluid dynamics is the *Lagrangian method*. In this approach, the simulation space is not subdivided by a grid. Instead, the numerical analysis is done by tracking individual fluid particles as they move and determining how the fluid properties associated with these particles change as a function of time. One of the advantages of the Lagrangian formulation is that fluid properties can be expressed as functions of time only. Moreover, by assuming the constant number of particles and constant per-particle mass, there is no need for an explicit mass conservation equation in the formulation. There are several particle-based meshless (gridless) numerical methods that have been used in CFD. Smoothed Particle Hydrodynamics (SPH) is one of the most popular of these method that is widely adopted by the Computer Graphics community.

3.2 SPH Model

Smoothed Particle Hydrodynamics is a particle-based computational model for simulating fluid flows. In the SPH method, fluid is represented by a set of particles that carry various fluid properties such as mass, velocity, and density. These properties are distributed around the particle according to an interpolation function (kernel function) whose finite support is h (kernel radius). For each point x in simulation space, the value of a fluid property can be computed by interpolating the contributions of fluid particles residing within a spherical region with radius h and centered at $\mathbf{x}$.

3.2.1 Density, Pressure and Viscosity Formulations

According to the SPH, interpolation of a quantity A is defined by the integral interpolant

$$A_I(\mathbf{r}) = \int A(\acute{\mathbf{r}})\, W(\mathbf{r} - \acute{\mathbf{r}}, h)\, d\acute{\mathbf{r}}, \tag{3.10}$$

where $d\acute{r}$ is the volume element, and the function W is the smoothing kernel. h is the core radius of the smoothing kernel [97]. The function W is chosen so that it falls off rapidly with distance. Usually, W is zero when the distance is greater than $2h$. Such kernels that vanish at a finite distance are said to have a compact support. In order to apply the SPH method to fluids, the total mass of the fluid body is distributed to the particles. Particle i has a fixed mass m_i, density ρ_i, and a position $\mathbf{r}_i$. The value of A at particle location i is, then, computed as

$$\int \frac{A(\acute{\mathbf{r}})}{\rho(\acute{\mathbf{r}})}\, \rho(\acute{\mathbf{r}})\, d\acute{\mathbf{r}}. \tag{3.11}$$

This integral is approximated by a summation over the near (closer than h) particles:

$$A_s = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{3.12}$$

For example, the density for the particle i then can be computed as follows

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h), \tag{3.13}$$

where m_j is the mass of particle j. One of the advantages of the SPH formulation is the first and second derivatives of quantities are computed easily since derivatives only effect the kernel function. For example, the first derivative of A_s is

$$\nabla A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h), \tag{3.14}$$

and the second derivative computed as

$$\nabla^2 A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h). \tag{3.15}$$

In its compact form the momentum equation of the the Navier-Stokes equations is as follows:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}.\nabla\mathbf{v}\right) = -\nabla p + \rho\mathbf{g} + \mu\nabla^2\mathbf{v}, \tag{3.16}$$

where p is the scalar pressure field, μ is the viscosity of the fluid, and $\mathbf{g}$ is the vector field of the total force acting of the fluid's body. The term $\mathbf{v}.\nabla\mathbf{v}$ accounts for the advection in which a small amount of fluid is advected by the surrounding fluid's velocity field. The pressure gradient ∇p defines the effect that a part of the fluid's volume is moved from a location with a high pressure to a location with low pressure. The term $\nabla(\mu\nabla\mathbf{v})$ is the momentum diffusion term, and it accounts for the dampening of the fluid's velocity field. Higher the value of the kinematic viscosity μ, the faster the dampening. If we assume for a constant viscosity, the term becomes $\mu\nabla^2\mathbf{v}$.

Using particles instead of an Eulerian approach has several advantages. One of them is that since the particles are advected by fluid's velocity field, there is no need to include the advection term in the Navier-Stokes solution [97]. Another advantage of using particles is that since the number of particles and their individual mass is constant and the lower bound between the particles is enforced, the equation for the conservation of mass is needless. Thus, the Navier-Stokes equations becomes

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t}\right) = -\nabla p + \rho\mathbf{f} + \mu\nabla^2\mathbf{v}. \tag{3.17}$$

If we designate $\mathbf{f}$ as $\mathbf{f} = -\nabla p + \rho\mathbf{f} + \mu\nabla^2\mathbf{v}$, then we can compute the change in the velocity, acceleration, by simply computing the value of $\mathbf{f}$.

Pressure of each particle location is computed by assuming ideal gas behavior where $p = k\rho$, k being the stiffness (or gas) constant. Desbrun et al. [28] point out that, unlike the astrophysical applications, the fluid version of the SPH should include a constant rest density. Thus, the pressure is computed by the equation

$$p = k(\rho - \rho_0), \tag{3.18}$$

where ρ_0 is the rest density. When naively applied, the SPH formulation results the following pressure gradient

$$\nabla p_i = m_i \sum_j m_j \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h), \tag{3.19}$$

where m_i and m_j mass of particles i and j and p_j and ρ_j are the pressure and density of particle j, respectively. However, the force resulting from the pressure gradient is not equal for different particles i and j. This violates the action-reaction principle. Desbrun et al. [28] propose using following symmetric pressure force equation

$$\mathbf{f}_i^{pressure} = -m_i \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{3.20}$$

To solve the same problem, antisymmetry of the pressure gradient, Müller et al. [101] propose using the following force equation which is simpler and faster

$$\mathbf{f}_i^{pressure} = -m_i \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{3.21}$$

The damping effect due to viscosity can be computed by applying the SPH rule to the viscosity term. The same asymmetry problem as in the pressure case exists in viscosity case. Müller et al. [101] solve this problem by considering velocity differences between the particle i and neighbor particles. Thus, the force due to viscosity can be found by the Equation:

$$\mathbf{f}_i^{viscosity} = m_i \mu_i \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{2\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{3.22}$$

where $\mathbf{v}_i$ and $\mathbf{v}_j$ are velocity of i and j respectively, and μ_i is viscosity coefficient of particle i. The SPH momentum equation for the particle i is then:

$$\frac{\mathbf{v}_i}{dt} = \mathbf{f}_i^{pressure} + \mathbf{f}_i^{viscous} + \mathbf{f}_i^{external}, \tag{3.23}$$

where $\mathbf{f}_i^{external}$ denotes the external forces such as gravity acting on the particle i, forces exerted on the particle by the boundaries etc.

3.2.2 Surface Tension

Surface tension is an important phenomenon especially in simulation fluid behavior in small scale. Forces due to surface tension appear in fluid boundaries, specifically on air-to-fluid boundaries. The surface tension forces are due to the unbalanced cohesive forces acting on the fluid molecules on the boundaries. Unlike the interior fluid molecules that are surrounded by other molecules, the attraction forces acting on the boundary molecules are not balanced [156]. The result is a hypothetical membrane like structure where tensile forces acting on every point of the surface along any line.

Morris [99] formulates the surface tension by using SPH by computing particle normals. For surface tension to be computed accurately, it is crucial to calculate surface curvature, and normal vector. The following Equation is used to compute the particle normals:

$$\mathbf{n_i} = \sum_j \frac{m_j}{\rho_j} (c_i - c_j) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h), \tag{3.24}$$

where ρ is particle density. c_i is called color or color field and can be computed as follows:

$$c_i = \sum_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{3.25}$$

The divergence of particle normals can be computed by,

$$(\nabla \hat{\mathbf{n}})_i = \sum_j \frac{m_j}{\rho_j} (\hat{\mathbf{n}}_i - \hat{\mathbf{n}}_j) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{3.26}$$

where $\hat{\mathbf{n}}$ can be defined by the following Equation:

$$\hat{\mathbf{n}} = \begin{cases} \mathbf{n}/|\mathbf{n}| & \text{if } |\mathbf{n}| > \epsilon \\ 0 & otherwise. \end{cases} \tag{3.27}$$

The surface tension force acting on a particle i is then defined as the following Equation:

$$\mathbf{f}_i^{surface} = -\frac{\sigma}{\rho_i} (\nabla \hat{\mathbf{n}})_i \, \mathbf{n}_i \tag{3.28}$$

Considering the surface tension force, Equation 3.23 becomes

$$\frac{\mathbf{v}_i}{dt} = \mathbf{f}_i^{pressure} + \mathbf{f}_i^{viscous} + \mathbf{f}_i^{surface} + \mathbf{f}_i^{external}. \tag{3.29}$$

3.2.3 Capillary Action

The microscale flow characteristic of fluid within a porous material is different than characteristics of a free fluid flow. In the latter, macroscale forces due to gravity, viscosity, pressure are dominant whereas in the former forces due to the porosity are more prominent. In fluid dynamics, the behavior of fluid within a very thin tube or a porous material is called *capillary action* and the force resulting in this behavior is *capillary force*. Absorption of liquids by the pore of a sponge, upward transfer of water within plant bodies, or uphill movement of fluid inside a thin tube are examples of capillary action. Capillary action occurs because the attraction force between the fluid and rigid object molecules is greater than the inner fluid cohesion force (which is the cause of surface tension).

Two main defining characteristics of a porous material are its porosity (denoted by ϕ) and permeability (denoted by s) [79]. Permeability is the ability of a porous material to transmit fluids. Permeability is mainly controlled by the size and interconnectivity of pores within the material. For isotropic materials, permeability can be represented by a scalar (denoted by k).

Porosity is the fraction of material's void volume to its total volume and is a scalar value between 0 and 1. It represents the porous material's fluid absorption capacity.

Saturation can be defined as the ratio of total fluid volume the porous medium

can hold to the current absorbed fluid volume. Hence, one can define saturation by:

$$s = \frac{m_{fluid}}{\phi \, V \, \rho_{fluid}}, \tag{3.30}$$

where V is the total volume of the material, ρ_{fluid} is the fluid density and m_{fluid} is fluid mass.

The main force acting on a fluid particle inside a porous material is due to the capillary pressure [19]. Capillary pressure is the pressure difference in multiphase flows that occurs across the interface. The capillary pressure can directly be expressed by porous material's saturation [67] and can be defined as:

$$p_{capillary}(s) = \frac{\sigma J(s)}{\sqrt{k/\phi}}, \tag{3.31}$$

where s is saturation, k is permeability of the material, ϕ is porosity, and σ a coefficient to control the pressure. $J(s)$ is Leverett function, and it mainly depends of the morphology of the porous material. We use the following Leverett function [82] to define hydrophilic materials:

$$J(s) = 1.417(1-s) - 2.120(1-s)^2 + 1.263(1-s)^3. \tag{3.32}$$

3.2.4 Kernel Functions

Choosing kernel functions is very important since they define how the particles, which represent fluid body in a discrete fashion, affect the space around them. One should consider the requirements of accuracy, stability, smoothness and computational efficiency in choosing SPH kernels. The kernel function should have a compact support. It should, also be at least singly differentiable.

In the computational physics literature, the spline Gaussian kernel is widely used

$$W_h(\mathbf{r}) = \frac{1}{\pi h} \begin{cases} 1 - \frac{3}{2}(\frac{r}{h})^2 + \frac{3}{4}(\frac{r}{h})^3 & \text{if } 0 \le |\mathbf{r}| \le h \\ \frac{1}{4}(2 - \frac{r}{h})^3 & \text{if } h \le |\mathbf{r}| \le 2h \\ 0 & \text{if } |\mathbf{r}| \ge 2h. \end{cases} \tag{3.33}$$

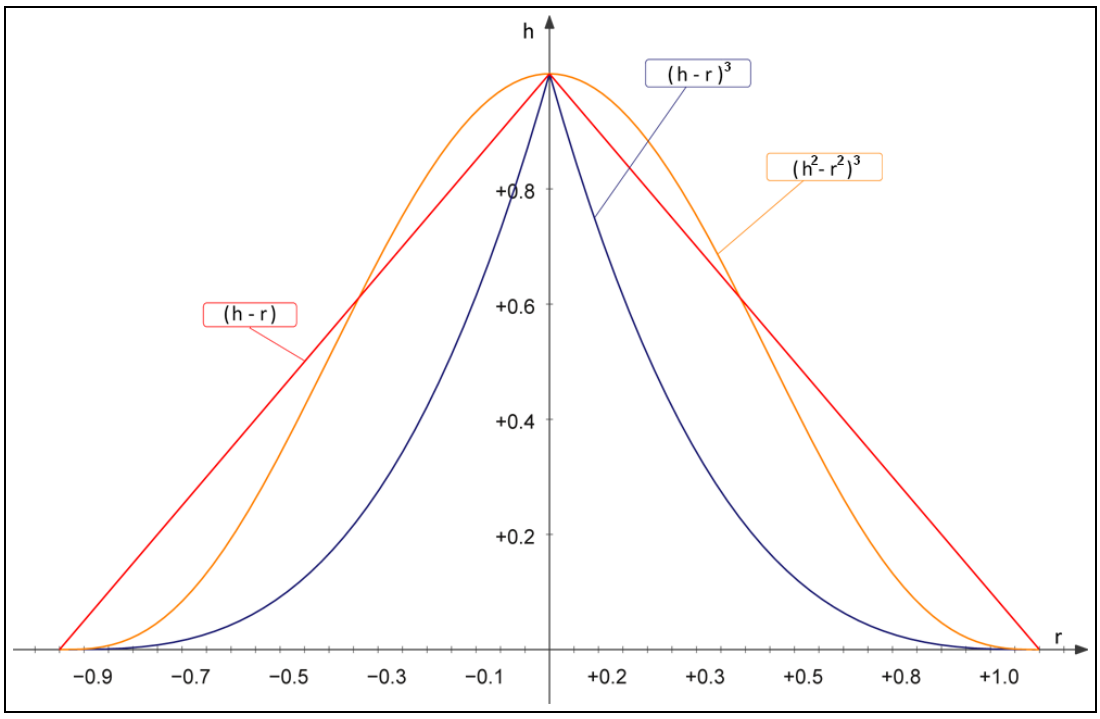


Figure 3.1: Plot of the several kernels. Kernel choice depends on the simulated material's characteristics, computational performance, and required accuracy of the simulation.

The choice of kernel functions is mainly dependent on the characteristics of the object being modeled. Desbrun et al. [28] model deformable objects by using the SPH. Thus, their application cannot have non-constant density and clusters of particles. They claim that since the gradient of the spline kernel (which is used in pressure forces computation) vanishes as r approaches to zero, that is as particles get closer, it is not suitable for their application. They choose to use the kernel

$$W_h(\mathbf{r}) = \frac{15}{\pi(4h)^3} \begin{cases} (2 - \frac{r}{h})^3 & \text{if } 0 \leq |\mathbf{r}| \leq 2h \\ 0 & \text{if } |\mathbf{r}| \geq 2h, \end{cases} \tag{3.34}$$

which, they claim, produces forces very similar to the Lennard-Jones forces.

Müller et al. [103] use three different kernel functions in order to simulate viscous fluid behavior. For computing pressure based forces they choose to use

the spiky kernel:

$$W_h(\mathbf{r}) = \frac{15}{\pi h^6} \begin{cases} (h-r)^3 & \text{if } 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{if } |\mathbf{r}| \geq h, \end{cases} \tag{3.35}$$

since it does not cause particle clustering and it has vanishing first and second derivatives at the boundary. The gradient of the spiky kernel is:

$$\nabla W_h(\mathbf{r}) = -\frac{45}{\pi h^6} \begin{cases} (h-r)^2 \, \mathbf{r} & \text{if } 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{if } |\mathbf{r}| \geq h. \end{cases} \tag{3.36}$$

For viscosity, they use another kernel since they claim that a standard kernel might cause numerical instabilities as they get negative values and might cause relative velocities increase. The kernel they employ for viscosity is as follows:

$$W_h(\mathbf{r}) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & \text{if } 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{if } |\mathbf{r}| \geq h, \end{cases} \tag{3.37}$$

whose Laplacian is,

$$\nabla^2 W_h(\mathbf{r}) = \frac{45}{\pi h^6} \begin{cases} (h-r) & \text{if } 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{if } |\mathbf{r}| \geq h, \end{cases} \tag{3.38}$$

and is used in viscosity computation. It is always positive, and it and its gradient vanishes at the boundary. For density computations, they use a simple kernel:

$$W_h(\mathbf{r}) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & \text{if } 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{if } |\mathbf{r}| \geq h. \end{cases} \tag{3.39}$$

In our implementation, we use the kernel defined in Equation 3.39 for density computations since it gives smoother values as particles get closer and it does not require taking the square root of particle distance. For computing pressure, we prefer using the gradient of the Spiky kernel (Equation 3.36), which prevents particle clustering. We choose the Laplacian of the kernel defined in Equation 3.37 for viscosity computations.

Chapter 4

Cloth and Knitwear Simulation

Mass-spring model is one of the most popular methods for modeling and simulating cloth-like objects. Being a particle-based method, mass-spring model is appropriate for defining simulated object's interaction with other objects, fine-tuning behavior of the object in terms of inter-particle forces, and parallelizing the simulation.

4.1 Mass-Spring Model

A mass-spring network consists of mass points connected by massless damped springs. By this model, it is assumed that the mass of the body is concentrated at specific points rather than it is scattered along the body. One of the constraints on the reality of the model, then, is the density of the mass points of the mesh. Forces acting on the mass points can be classified as external and internal forces. Internal forces are spring forces that mass points exert on each other through damped springs, and external forces are environmental forces such as gravity, viscous drag, impulse based forces, and user defined forces such as mouse drag. The mesh is simulated through time by calculating position of the mass points after a specified time step.

4.1.1 Modeling the Cloth Mesh

In its simplest form, a mass-spring mesh is constructed by three kinds of springs, as illustrated in Figure 4.1. Structural (linear) springs connect each mass point to its four immediate neighbors (upper, lower, left, and right). Shear (diagonal) springs connect each mass point to its immediate diagonal neighbors, and flexion (bending) springs connect each mass point to every other mass point. Structural springs are constrained by the stretching and compression forces. Diagonal springs are constrained by the shear stresses, whereas bending springs are used to limit bending of the structure, since they are constrained by the flexion stresses [117]. Setting spring constants of these spring types independently enables us to mimic different types cloth and rubbery objects.

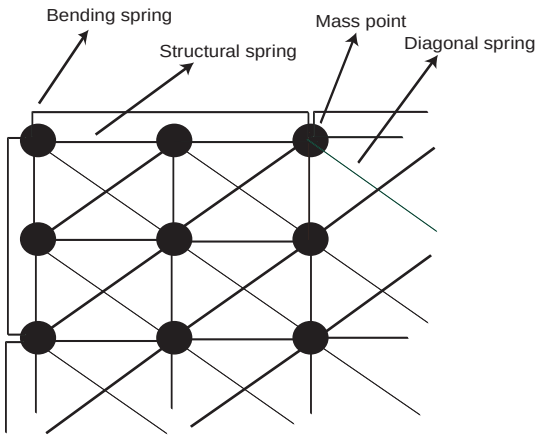


Figure 4.1: A sample mass-spring mesh

4.1.2 Defining the Forces

We use the second Newtonian law of dynamics in order to determine position of a mass point i at a particular time:

$$\mathbf{f}_i^{total} = m_i \, \mathbf{a}_i, \tag{4.1}$$

where m_i is the mass of the point, $\mathbf{a}_i$ is the acceleration caused by the net force $\mathbf{f}_i$ acting on the mass point. The net total force is computed by considering the internal forces, namely the forces mass points exert on each other through springs, and external forces such as gravity, viscous air drag, wind etc. The force acting on a mass point i by its neighbor j through a damping spring can be defined by the Hooke's Law:

$$\mathbf{f}_{i,j}^{spring} = \left((-k_s\left(|\mathbf{s}_{i,j}| - |\mathbf{s}_{i,j}^{rest}|\right) + d_s\,|\mathbf{v}_{i,j}|\right)\right) \frac{\mathbf{s}_{i,j}}{|\mathbf{s}_{i,j}|}, \tag{4.2}$$

where

- $\mathbf{s}_{i,j} = \mathbf{r}_i - \mathbf{r}_j$, where $\mathbf{r}_i$ and $\mathbf{r}_j$ are position vectors of mass points i and j, respectively,
- $|\mathbf{s}_{i,j}^{rest}|$ is the rest length of the spring,
- k_s and d_s are the spring, and damping constants of the spring, respectively,
- $\mathbf{v}_{i,j}$ is the projection of the relative velocity of mass point i and j onto the vector $\mathbf{s}_{i,j}$.

The spring force is symmetric up to its direction for the mass points connected by the spring. That is:

$$\mathbf{f}_{i,j}^{spring} = -\mathbf{f}_{j,i}^{spring}. \tag{4.3}$$

Besides the spring forces, external forces act on mass points. These forces are computed and accumulated for each of the mass points. Equation 4.4 gives the force caused by gravity on a mass point i:

$$\mathbf{f}_i^{gravity} = m_i\,g, \tag{4.4}$$

where g is the global acceleration of gravity.

Viscous air drag is another external force to act on the mass points. It is designed as a uniform force which has the effect of dissipation of kinetic energy of mass points. This force can be calculated by using Equation 4.5:

$$\mathbf{f}_i^{drag} = -c_{drag}\,\mathbf{v}_i, \tag{4.5}$$

where c_{drag} is the coefficient of air drag and $\mathbf{v}_i$ is the velocity vector of the mass point. Adding moderate amount of viscous drag enhances realistic look and numerical stability of the system, but much of this force gives unrealistically oily look.

Moving air (or fluid) also exerts a force on the mass points, which can be computed by Equation 4.6:

$$\mathbf{f}_i^{wind} = c_{wind}(\mathbf{n}_i \,.\, (\mathbf{v}_{wind} - \mathbf{v}_i))\mathbf{n}_i, \tag{4.6}$$

where c_{wind} is a coefficient to express the amplitude of wind, $\mathbf{v}_{wind}$ is the velocity vector of wind, $\mathbf{v}_i$ is the velocity of the mass point i, and $\mathbf{n}_i$ is the normalized normal vector to the surface of the mass-spring mesh at the mass point m_i's position.

4.2 Modeling Knitwear

Modeling and simulating knitwear introduce some additional challenges. One of the challenges is to model, and simulate the thread structure of knitwear. We choose to model the thread structure by defining the thread stitches by a repeating pattern of bonding points. The pattern is defined in a uniform grid which sit upon an underlying spring-mass network. The coordinates of bonding points are interpolated from the coordinates of mass points. This structure is illustrated in Figure 4.2.

Another challenge in modeling knitwear is to simulate the knitwear's thickness. This thickness of knitwear is important for a realistic simulation especially in the presence of perpendicular forces. Thus, a realistic model should have a volumetric representation of the knitwear structure. This can be achieved without sacrificing the simplicity and speed of the mass-spring method. The model depicted in Figure 4.3 illustrates how this can be done. In this model, there are three layers of mass-spring meshes that are connected to each other with volumetric springs. Each of these three layers are modeled by shear, stretch and bending springs as explained in Section 4.1.1. This three layered structure provides a

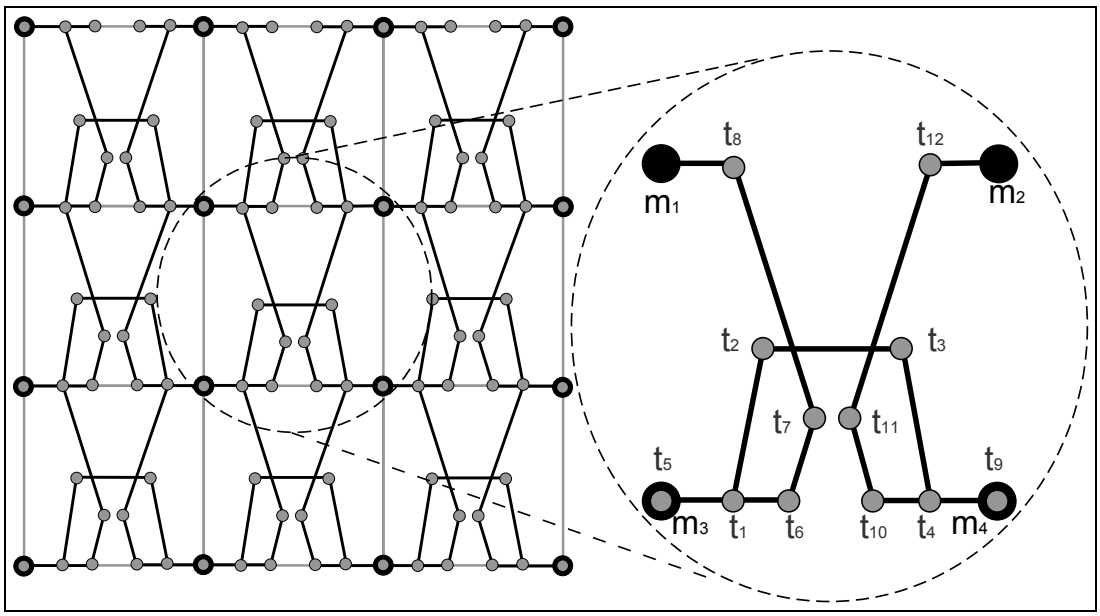


Figure 4.2: Bonding points (gray dots) and mass–points (black dots).

dynamic thickness so that knitwear reacts to perpendicular forces in a realistic and adjustable way.

The repeating pattern of bonding points is achieved by a set of interpolation formula. The formulae define the set of mass points contributing each of bonding point coordinate computation. By selecting mass points from the different spring-mass layers, it is possible to achieve a realistic 3D visual effect for thread stitches.

Mass points are advected dynamically as a result of external forces (e.g., gravity, wind, collision, user forces) and internal forces (e.g., stretching, bending, and shearing forces). The bonding points of the knitwear model are updated (interpolated) accordingly (see Figure 4.2). In Figure 4.2, the gray and black dots are bonding points and mass points, respectively, and the gray lines represent springs. The interpolation equations for the positions of stitch control points is similar to those described in [31], except the fact that we exploit layered structure of the mass-spring mesh to model thickness. Equation 4.7 describes the interpolation of the positions of bonding points for the thread structure depicted in Figure 4.2. In Equation 4.7, m_is and m'_is are mass points that belong to layers 1 and 3, respectively.

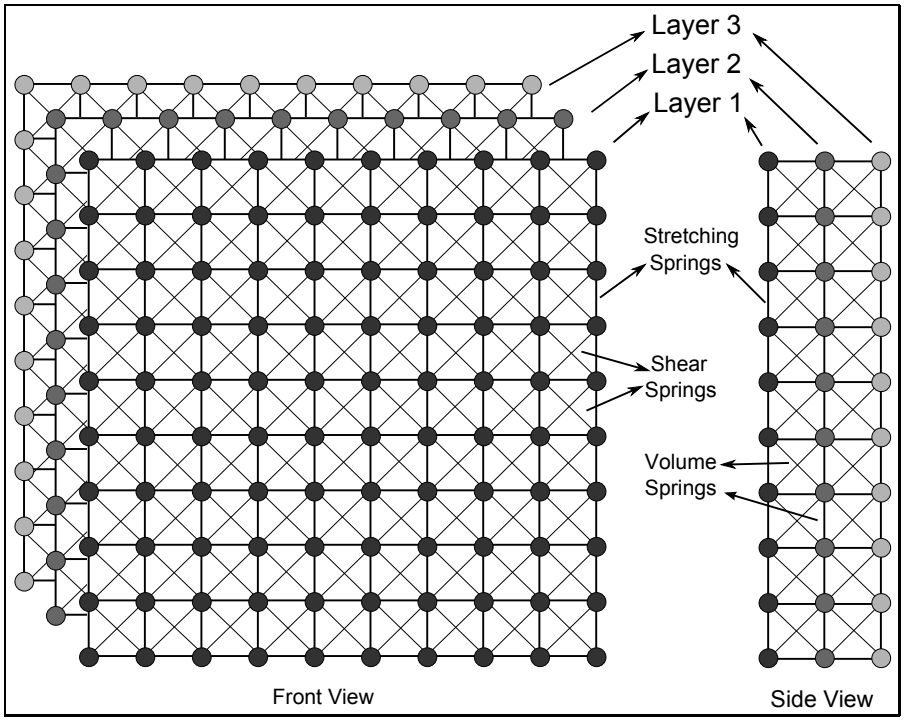


Figure 4.3: Mass-spring structure of our knitwear model. To simulate the thickness of knitwear, the layers are connected by volume preserving springs.

$$
\begin{aligned}
t_1 &= \tfrac{6}{7}m_3 + \tfrac{1}{7}m_4 \\
t_2 &= \tfrac{5.5}{21}m'_1 + \tfrac{1.5}{21}m'_2 + \tfrac{11}{21}m'_3 + \tfrac{3}{21}m'_4 \\
t_3 &= \tfrac{1.5}{21}m'_1 + \tfrac{5.5}{21}m'_2 + \tfrac{3}{21}m'_3 + \tfrac{11}{21}m'_4 \\
t_4 &= \tfrac{1}{7}m_3 + \tfrac{6}{7}m_4 \\
t_5 &= m'_3 \\
t_6 &= \tfrac{5}{7}m'_3 + \tfrac{2}{7}m'_4 \\
t_7 &= \tfrac{4.5}{28}m_1 + \tfrac{2.5}{28}m_2 + \tfrac{13.5}{28}m_3 + \tfrac{7.5}{28}m_4 \\
t_8 &= \tfrac{6}{7}m_1 + \tfrac{1}{7}m_2 \\
t_9 &= m'_4 \\
t_{10} &= \tfrac{2}{7}m'_3 + \tfrac{5}{7}m'_4 \\
t_{11} &= \tfrac{2.5}{28}m_1 + \tfrac{4.5}{28}m_2 + \tfrac{7.5}{28}m_3 + \tfrac{13.5}{28}m_4 \\
t_{12} &= \tfrac{1}{7}m_1 + \tfrac{6}{7}m_2
\end{aligned}
\tag{4.7}
$$

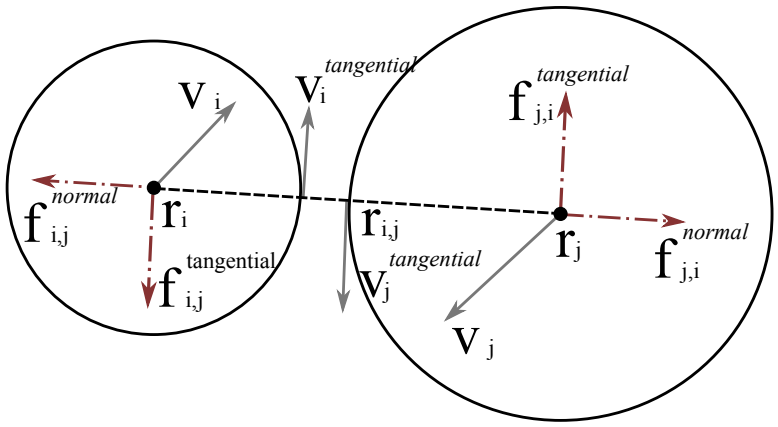


Figure 4.4: The normal and tangential components of the contact force acting on particles p_i and p_j.

4.3 Handling Self Collisions

We handle self–collision of the mass-spring model in a particle–to–particle basis, where penalty forces are applied between the neighboring particles. These penalty forces are computed by the discrete element method (DEM). Particle proximities are detected by a grid based neighbor search algorithm where mass points connected by springs are ignored.

Discrete Element Method (DEM) is a popular numerical method for defining inter–particle forces especially in granular materials. The contact force $\mathbf{f}_{i,j}^{DEM}$ between two particles i and j can be computed as the sum of tangential and normal components:

$$\mathbf{f}_{i,j}^{DEM} = \mathbf{f}_{i,j}^{normal} + \mathbf{f}_{i,j}^{tangential}, \tag{4.8}$$

as shown in Figure 4.4.

The normal force component $\mathbf{f}_{i,j}^{normal}$ is repulsive and acts on particle centers in the direction of the vector $\mathbf{r}_{i,j} = \mathbf{r}_i - \mathbf{r}_j$, where $\mathbf{r}_i$ and $\mathbf{r}_j$ are the centers of particles i and j, respectively. $\mathbf{f}_{i,j}^{normal}$ is computed according to Equation 4.9:

$$\mathbf{f}_{i,j}^{normal} = k \, \frac{\mathbf{r}_{i,j}}{|\mathbf{r}_{i,j}|} \, u_{i,j} - c \, (\mathbf{v}_i^{normal} - \mathbf{v}_j^{normal}), \tag{4.9}$$

where k is the compression stiffness coefficient, c is the coefficient of viscous damping, $\mathbf{v}_i^{normal}$ is velocity component of particle i in the direction of $\mathbf{r}_{i,j}$, and $u_{i,j} = |\mathbf{r}_{i,j}| - (radius_i + radius_j)$ is the particle overlap.

$\mathbf{f}_{i,j}^{tangential}$ is the tangential shear force and it defines the resistance to the movement that the particles exert on each other in tangential direction. It is computed according to Equation 4.10:

$$\mathbf{f}_{i,j}^{tangential} = \mu\ (k\ u_{i,j}) \frac{\mathbf{v}_i^{tangential} - \mathbf{v}_j^{tangential}}{|\mathbf{v}_i^{tangential} - \mathbf{v}_j^{tangential}|}, \tag{4.10}$$

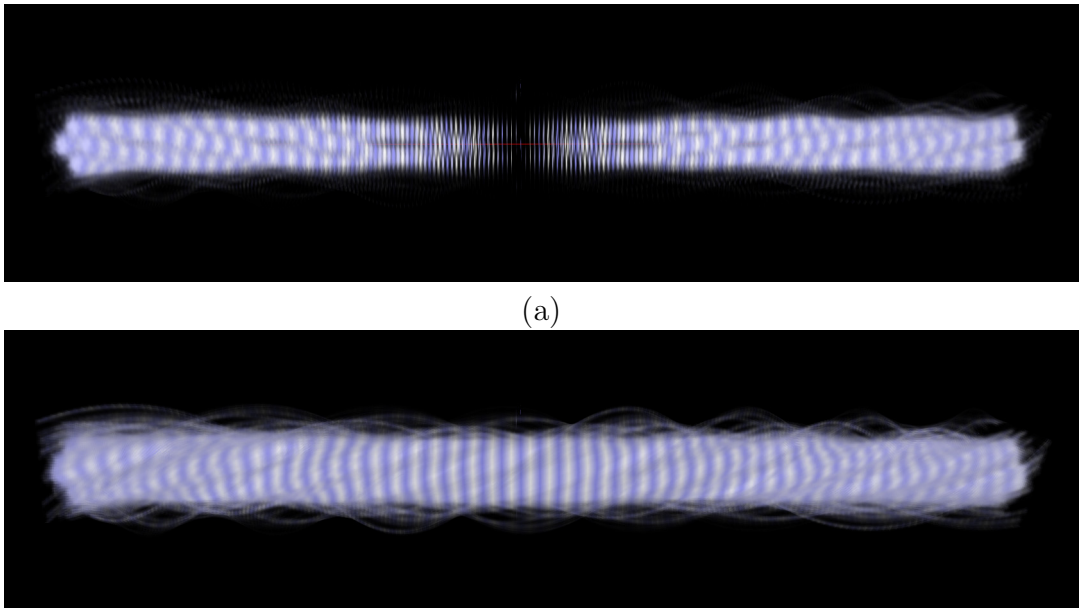
where k and $u_{i,j}$ are as defined above, $\mathbf{v}_i^{tangential}$ and $\mathbf{v}_i^{tangential}$ are the tangential velocities of particles i and j, respectively, and μ is the friction coefficient. This formulation of tangential shear force scales with the magnitude of the normal force, thus, ensures the stability.

Particle based self collision handling is straight forward to implement, easy to incorporate to the particle system, and easily parallelizable. Although visual artifacts may occur in course meshes, it produces visually satisfying results in fine enough mass-spring meshes.

4.4 Knitwear Rendering

The rendering of the knitwear strand is important because it needs to synthesize the microstructure of the knitwear, which consists of a huge number of thin fibers and has a fuzzy look. We use the lumislice primitive presented in [154] as a 2D texture, which shows the distribution of the yarn fibers in a cross–section of the yarn.

We render an array of quads textured with 2D lumislice texture to obtain a volumetric rendering using the alpha-blending capabilities of the graphics hardware. The problem with this approach is that the alpha-blended 1-dimensional array of textured quads do not handle view angles that are non–perpendicular to rendered quads. This artifact is illustrated in Figure 4.5(a). To overcome



(a)

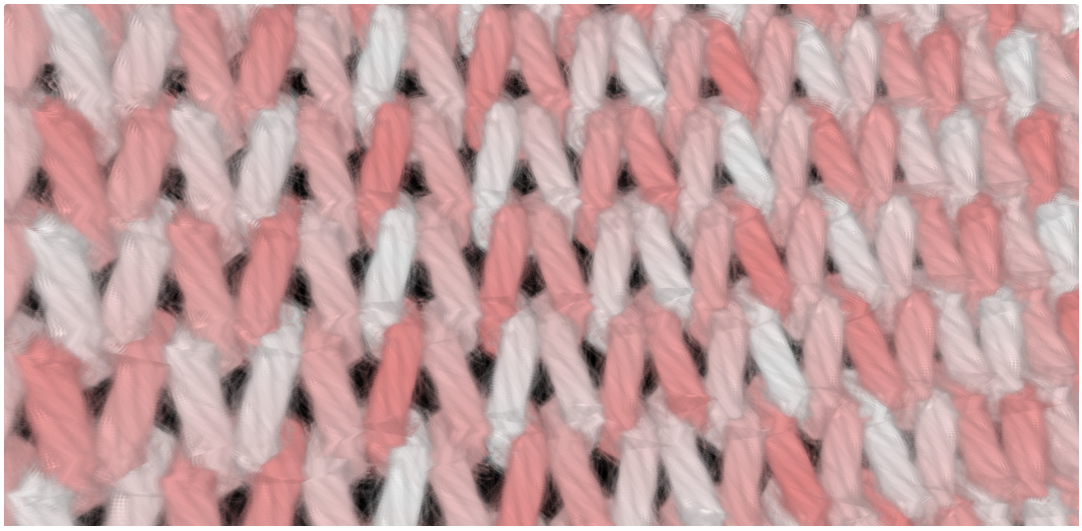
(b)

Figure 4.5: (a) The problem related with 1-dimensional array of 2-dimensional quads is shown. This artifact occurs when the quads lie parallel to the viewing direction. (b) The problem is solved by using 3-D grid of voxels as explained in the text.

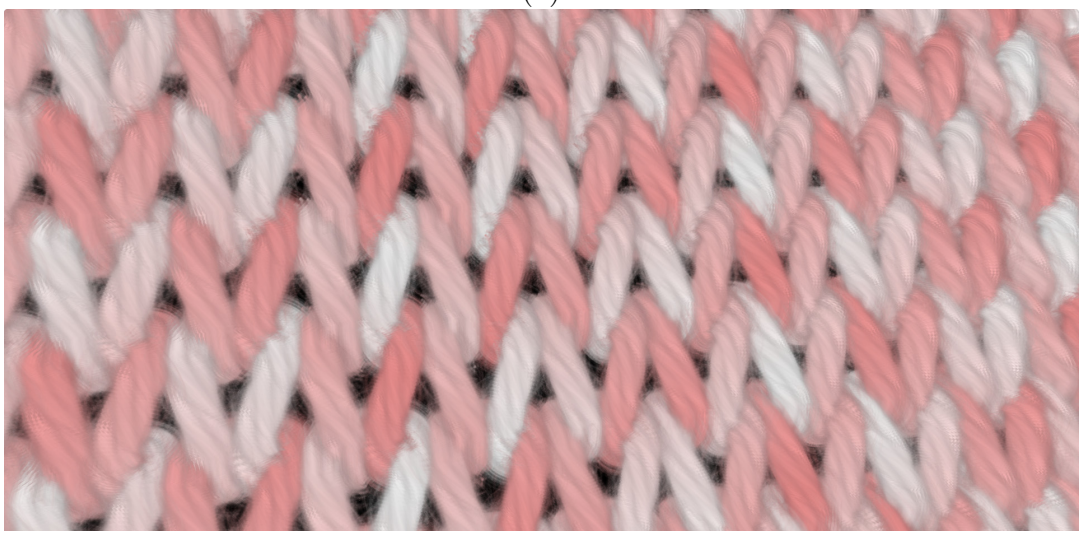
this problem, we create a 3-dimensional grid of voxels for volumetric rendering, instead of the 1-dimensional array of textured quads. In this approach, there is a 1-dimensional array of quads along each coordinate axis of the local coordinate system. Thus, even if one or two of the 1-dimensional quad arrays are parallel to the viewing direction, it is guaranteed that the last one is not. Figure 4.5(b) illustrates the result. The 3D voxel grid structure is illustrated in Figure 4.7.

One limitation of the hardware-accelerated alpha-blending is that the quads must be rendered in back-to-front order. This is a requirement because of how blending works in hardware: the rendering pipeline must know the pixel at the back when rendering the pixel at the front to be able to compute the correct color of the final pixel. This sorting operation has to be done whenever the viewer orientation or model position/orientation changes. To sort the quads, we employ bitonic sort on GPU.

After sorting, we twist the grid of quads along the axis defined by the normal



(a)



(b)

Figure 4.6: (a) The artifacts because of the discontinuities in the overlaps at the segment joints. (b) The artifacts are alleviated by fitting a Catmull-Rom spline on the bonding points.

of each quad in order to increase the visual quality of the strand. The 3D grid of voxels are aligned along the path defined by stitch bonding points. When the voxels are positioned on the straight line between the bonding points, visual artifacts appear because of the overlaps at the joints as illustrates in Figure 4.6(a). To overcome this problem, we fit a Catmull-Rom spline on the bonding points and this spline is used to determine the path of each thread of knitwear. This

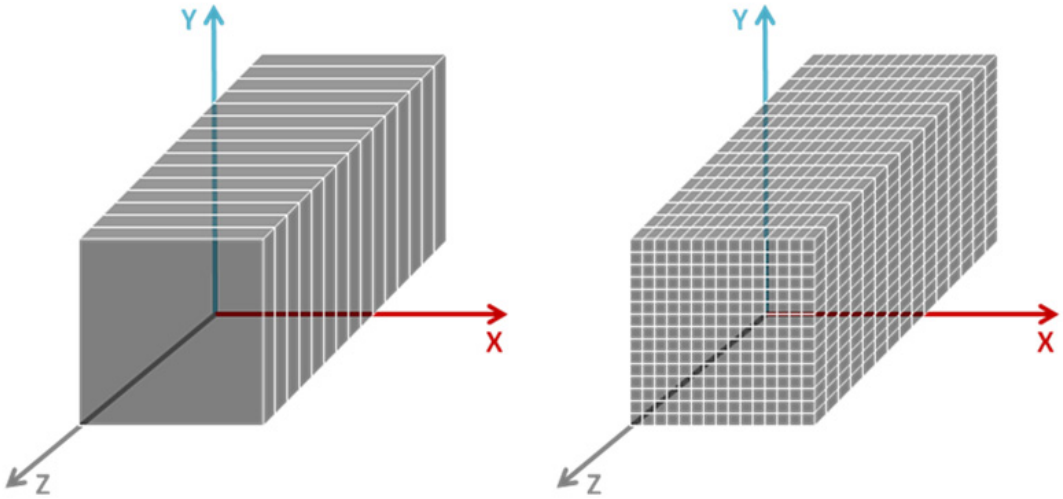


Figure 4.7: 1-dimensional array of quads (on the left) versus 3-dimensional grid of voxels (on the right) are shown.

provides a smoother path for the quads and consequently results in a much better visual quality. In this improved approach, instead of rendering a different yarn segment corresponding to each line segment, we render the whole yarn thread at once by placing the quads with regular intervals on the spline. When we render the threads in this way, no discontinuity results in a single thread, and thus, the image quality is much higher, as shown in Figure 4.6(b).

Soft shadows are indispensable for a realistic, high quality rendering of a knitwear model as they help stitches stand out, especially when the cloth is folded or there are many levels of cloth on top of each other. To achieve high rendering quality of soft shadows, we employ cascaded shadow mapping technique [30], which divides the frustum into a number of segments, making it possible to use different texture resolutions and obtaining high level antialiasing.

Chapter 5

Particle System Implementation

In this chapter, we discuss several practical issues of a particle-based simulation system. Implementing a fast, robust, and realistic particle system depends on addressing these issues and formulating effective solutions. Each of the Sections in this chapter presents an important aspect of implementing a particle system and presents a practical and robust solution.

Section 5.1 presents the details of how to define boundary conditions for a particle based fluid simulation system. Boundary conditions are essential for defining fluid body's interaction with its environment. Section 5.2 gives a detailed description of our fluid surface generation method and explains the improvements we propose to generate a free fluid surface with a higher visual quality. Section 5.3 underlines the implementation of the surface tension and capillary forces whose theory is explained in Section 3.2.2 and Section 3.2.3, respectively. In Section 5.5, we explain the neighbor search algorithm we employ to detect particle proximities. Section 5.6 underlines the alternatives of numerical integration methods.

5.1 Boundary Conditions

The choice of boundary forces is crucial to simulate fluid behavior at the boundaries of a rigid, unmovable (walls, pipes, etc.) objects or as it flows through channels, filters, or porous materials. Flow characteristics in these cases are usually that of Laminar flows and generally dominated by viscous and friction forces rather than inertial forces. Thus, it is important to model fluid-boundary interactions to achieve realistic results. Fluid-boundary interaction forces can be considered in two categories.

Non-penetration condition requires that fluid particles do not penetrate into the boundary region. Also known as Neumann boundary condition [20], Non-penetration condition can be expressed mathematically as follows:

$$\frac{\partial \mathbf{v}}{\partial \mathbf{n}} = 0, \tag{5.1}$$

where $\mathbf{v}$ is fluid velocity and $\mathbf{n}$ is the boundary normal vector. In computational physics, *periodic boundary condition* may be used instead of non-penetration condition. In periodic boundary conditions, materials (particles) leaving the simulation space from a boundary appears at the opposite boundary. That is, a particle leaving the boundary of the simulation space from the front wall reappears at the back door in the following simulation step.

Slip conditions consider the force exerted on the fluid particles in tangential direction to the boundary surface. In the case of free-slip condition, no force is exerted on fluid particles as they move tangential to the boundary whereas in the case of no-slip condition a tangential frictional force is applied so that tangential velocity is zero. No-slip condition is usually simulated by modeling viscous drag applied by boundary particles.

Typically, non-penetration and slip boundary conditions in particle based simulation systems are enforced by inserting stationary boundary particles which exert repulsive and frictional forces on the fluid particles [97]. Usually, the boundary particles are placed to create a layer (or several layers depending on the application) of particles. Figure 5.1 illustrates such a setting where a layer of

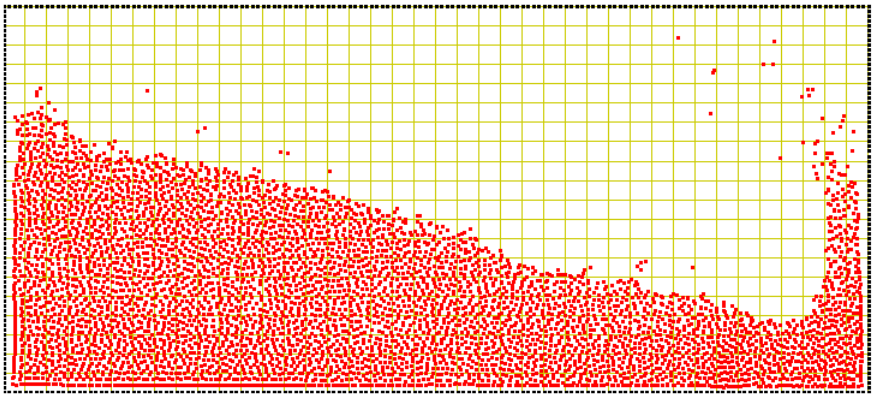


Figure 5.1: A frame of a 2D particle-based fluid simulation where a layer of stationary boundary particles (drawn in black) are placed to enforce boundary conditions on the fluid particles(drawn in red).

boundary particles are placed to enforce boundary conditions. One can choose one of the several fluid particle-to-boundary particle interaction schema to define non-penetration and slip conditions.

5.1.1 Enforcing Boundary Conditions using SPH-Based Forces

One way to define fluid particle-to-boundary particle interaction forces is to compute SPH-based pressure and viscosity forces between fluid and boundary particles. This can be achieved by computing density and velocity values for boundary particles. The density of a boundary particle can be computed by Equation 3.13. In this computation, it is possible that only fluid particles contribute to the density computation. However, as mentioned by Morris et al. [100], when the boundary particles do not contribute to the density of fluid particles, pressure stays constant as fluid particles and boundary particles diverge. Thus, to generate a pressure based restoring force, Morris et al. propose that boundary particles contribute to the density of fluid particles. Based on the computed density, a repulsive pressure force (Equation 3.21) is computed to enforce no-penetration condition.

To compute a frictional tangential force for slip conditions, a viscosity-based force can be employed. This requires calculating velocities of boundary particles. Boundary particle velocities are computed by interpolating the velocities of neighboring fluid particles. The velocity $\mathbf{v}_j$ of boundary particle j is interpolated from the velocities of the neighboring fluid particles by the following equation:

$$\mathbf{v}_j = \sum_i m_i \, \mathbf{v}_i \, W_{ij}, \tag{5.2}$$

where W_{ij} is the kernel defined by Equation 3.34, and m_i and $\mathbf{v}_i$ are the mass and velocity of neighboring fluid particle i, respectively. The viscous drag exerted on fluid particles by boundary particles is computed by Equation 3.22. It should be noted that the boundary particles are stationary and their evolved density and velocity values are used only in viscous drag computation. One disadvantage of implementing boundary conditions by SPH-based pressure and viscosity forces is that doing so introduces an additional computational burden since density for each boundary particle should be computed at each time step prior to boundary force computations.

5.1.2 Enforcing Boundary Conditions by Lennard-Jones Potential or the Discrete Element Method

Another choice for implementing non-penetration condition is known as the Lennard-Jones potential [3] and widely used in molecular dynamics. The form of the Lennard-Jones potential is as follows:

$$f(d) = \begin{cases} \frac{D}{d}((\frac{d_0}{d})^{\alpha_0} - (\frac{d_0}{d})^{\alpha_1}) & \text{if } 0 \leq d \leq d_0 \\ 0 & \text{otherwise,} \end{cases} \tag{5.3}$$

where d is the distance between the particles and d_0 is chosen to be the kernel radius h. The usual choices of α_0 and α_1 are 12 and 6, respectively [98]. Notice that in this formulation the Lennard-Jones potential is solely repulsive.

Boundary conditions can also be enforced by the discrete element method

(DEM) which is described in Section 4.3 within the context of self-collision handling in mass-spring meshes. With DEM method one can fine-tune repulsive normal force and tangential frictional force by adjusting the corresponding coefficients. The advantage of the Lennard-Jones potential or the discrete element method over a SPH based boundary force schema is that it is computationally more efficient since it only depends on the distance between the particles.

5.1.3 Implementing Adhesive Boundary Forces

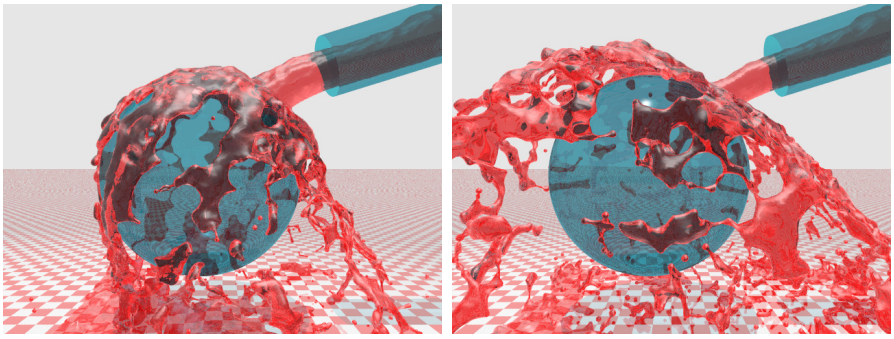
In addition to ensuring non–penetration and no–slip conditions, boundary particles contribute to fluid particles' pressure–based force computation. This creates an adhesive force that prevents fluid particles from leaving the solid boundary freely. We evaluate the pressure value of each solid boundary particle and apply the force computed by the following equation on each fluid particle i for creating a realistic and easily controllable adhesion–like effect.

$$\mathbf{f}_i^{adhesive} = \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W_{ij}, \tag{5.4}$$

where j is the neighboring boundary particle of particle i, m_j is the mass, p_j is the pressure, ρ_j is the density of particle j, and W_{ij} is as defined by Equation 3.35. Figure 5.2 shows the still images from two simulations where, in the left frame, proposed pressure–based adhesion force is active, and in the right, there is no adhesion force acting on the fluid particles.

5.2 Fluid Surface Generation and Rendering

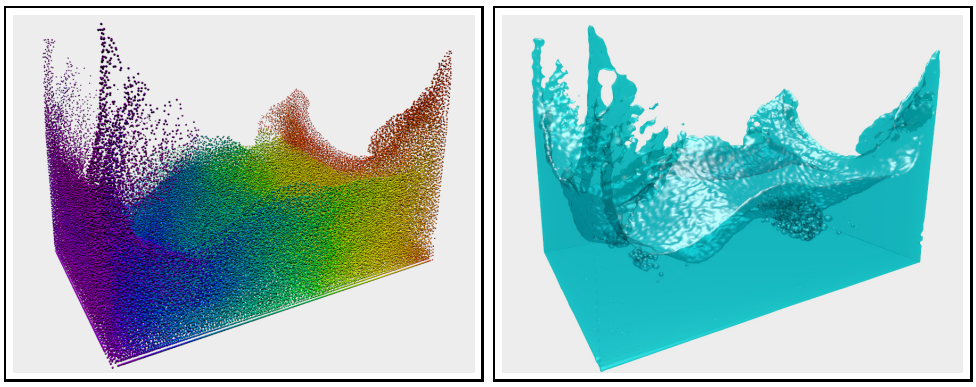
For particle based fluid simulation methods, it is a challenging task to extract a fluid surface since particles do not carry any explicit information about their spatial arrangement and connectivity, see Figure 5.3. A 3D particle based simulation system should provide a robust and physically correct surface generation system for rendering. The generated surface should deliver details in regions like



(a) (b)

Figure 5.2: Fluid pours down on a sphere with (a) adhesion effect and (b) no–adhesion effect.

thin fluid fronts and drops and frame-to-frame coherence. The typical solution is to compute a polygonized isosurface from particle positions for rendering. One of the frequently used algorithms for polygonizing an isosurface is the Marching Cubes [85].



(a) (b)

Figure 5.3: (a) A particle-based simulation, (b) a fluid surface is generated and rendered.

The Marching Cubes algorithm traces a uniform grid iteratively and achieves tessellation based on the scalar values computed on the corners of the grid cells. Originally, the algorithm considers 15 unique cube configurations for each grid cell. The algorithm has been improved by resolving some ambiguous cases [21, 107, 84].

Within the context of a particle-based fluid simulation, the scalar value used by the Marching Cubes is computed by measuring the distances of the neighboring

fluid particles to the cube corners. Thus, for a grid cube corner x, the function $\phi(x)$ computes the scalar value:

$$\phi(x) = \sqrt{\sum_i \left(1 - \left(\frac{d_i}{h}\right)^2\right)^3}, \tag{5.5}$$

where h is the threshold distance, i iterates over the fluid particle neighbors of x, and $d_i = |\mathbf{x} - \mathbf{r}_i|$, where i is the fluid particle with $d_i \leq h$.

The resulting surface captures the main features of the fluid body but it has a thickening effect in detailed surface regions such as waves and water fronts and a bumpy look in flat regions. Adams et al. [1] address this problem by employing a weighted function for the isosurface computation. Their method uses a higher particle density in detailed regions improving the visual quality of the free fluid surface. We propose a modification to Equation 5.5 such that it differentiates particles according to their relative positions to the free fluid surface by assigning a value. This value is computed for each particle according to its proximity to the free fluid surface. We refer to this value as the *surface value* and calculate it for each particle using particle normal vectors, as described in [129].

To compute the normal vector for each particle i, we first determine the centroid c_i of the sphere with radius h centered at the location of particle i. That is:

$$\mathbf{c}_i = \frac{\sum_j \mathbf{r}_j}{k_i}, \tag{5.6}$$

where particle j is a neighbor of particle i satisfying $|\mathbf{r}_j - \mathbf{r}_j| \leq h$, k_i is the number of such neighbors of particle i, and h is the SPH kernel radius. Then the normal vector of particle i is defined to be:

$$\mathbf{n}_i = \mathbf{r}_i - \mathbf{c}_i, \tag{5.7}$$

where $\mathbf{r}_i$ is the position of the particle i. The length of the normal vector, $|\mathbf{n}_i|$, indicates relative proximity of the particle to the fluid surface. The magnitude of the normal vectors of the particles that are closer to the surface are larger than those of particles located deeper the fluid body. Figure 5.5 illustrates a situation where two particles having normal vectors of different lengths because of their relative distances to the fluid surface.

We compute the surface values for each fluid particle i by the following equation:

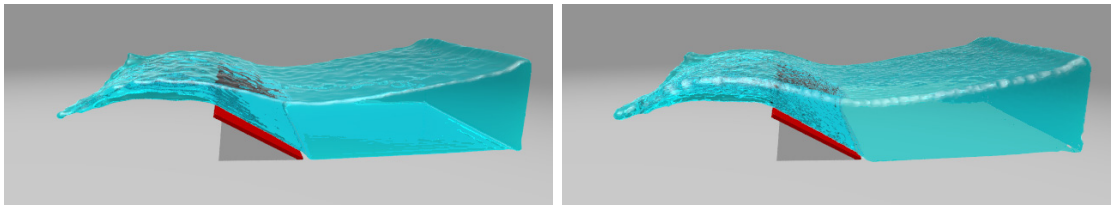
$$s_i = 1 - \left(\frac{\sum_j |\mathbf{n}_j|}{k_i \; h} + \frac{\Re}{2} \right), \tag{5.8}$$

where $\mathbf{n}_j$ is the normal vector of the neighboring fluid particle j, and k_i is the number of neighbors of particle i. $\Re$ is defined to be the maximum of $\Re_i$'s where

$$\Re_i = \frac{\sum_j |\mathbf{n}_j|}{k_i \; h}. \tag{5.9}$$

Equation 5.8 ensures that particles closer to the fluid surface have smaller surface values, thus contribute to the isosurface less than non–surface particles. After including s_i to the function $\phi(\mathbf{x})$, Equation 5.5 becomes:

$$\phi(\mathbf{x}) = \sqrt{\sum_i \left(1 - \left(\frac{d_i \; s_i}{h} \right)^2 \right)^3}. \tag{5.10}$$



(a) (b)

Figure 5.4: Fluid dam breaks into a rigid object. In (a) the surface is generated by including the surface value as proposed and in (b) no surface value is included during the surface generation.

By including the surface value s_i, the generated fluid surface displays better quality on flat surfaces and captures finer details in splashes and filaments since fluid particles close to the surface contribute less to the isovalue. Figure 5.4 illustrates effectiveness of our modified surface generation algorithm. Figure includes two frames of the same scene. Figure 5.4 (a) is from the simulation where the surface value is included during the surface generation. Figure 5.4 (b) is from the simulation where surface value is not considered. It is clear from Figure 5.4 that including the surface value enhances the visual quality of the simulation.

We experimented several values for the grid resolution for our Marching Cubes algorithm. Finer grids produces higher quality surface while introducing a computational burden. In our system, we found that the grid resolution which is

equal to $\frac{1}{6}h$, where h is the kernel radius as defined before, is the optimum. With this resolution each grid corner (the ones that are surrounded by the fluid body) has around 80 neighboring fluid particles.

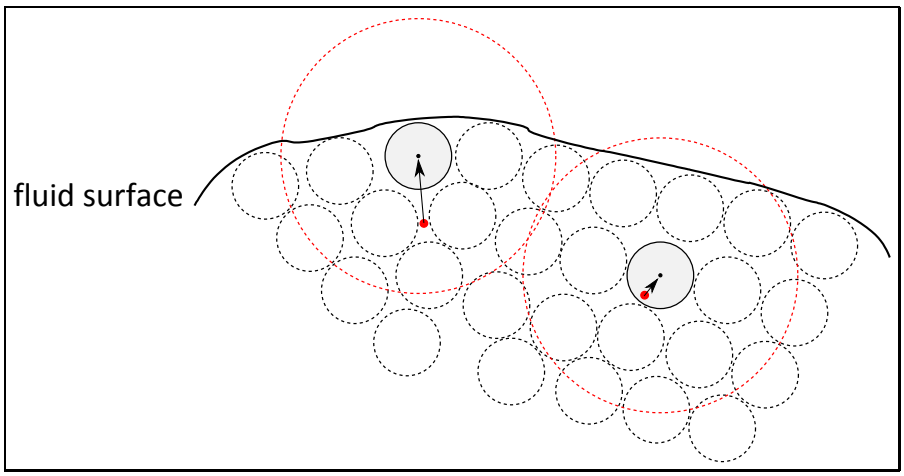


Figure 5.5: Computing particle normals. The particle closer to the surface has a longer normal vector than the particle residing inside the fluid body. The red dashed circle and red dot illustrate the neighborhoods of particles and the centroids of neighboring particles, respectively. The black arrow pointing from the centroid to particle center is the particle's normal vector.

We use The Persistence of Vision Ray tracer (POV-Ray) [92] for offline rendering of tessellated fluid surface. A faster but probably less exact alternative would be isosurface ray casting algorithm implemented on the graphics hardware [106, 133, 124].

5.3 Implementing Surface Tension and Capillary Forces

An accurate computation of the force due to surface tension is crucial both for robustness and visual realism of a fluid simulation. Section 3.2.2 presents a theoretical account of the surface tension force and a method to compute it. We implement the surface tension based on the particle normals that are computed by the Equation 5.7 as explained in Section 5.1. The surface tension force is

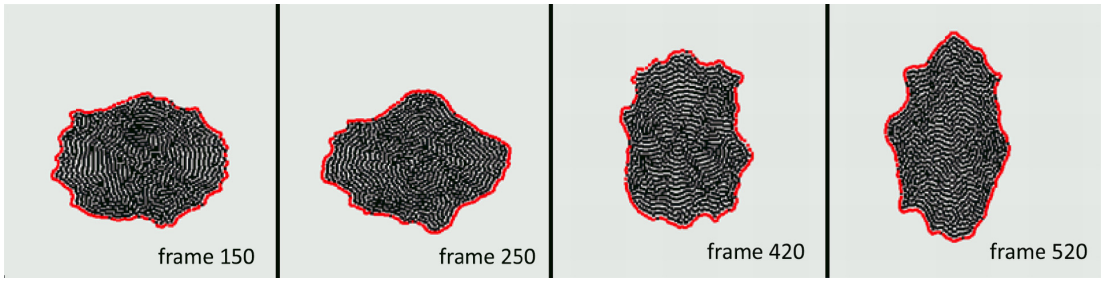


Figure 5.6: 2D SPH simulation where no gravity and environmental viscous drag are present. Because of the surface tension fluid body ossilates between drop-like shapes along two axes.

computed by Equation 3.28 and applied to particles. Figure 5.6 illustrates the result of the surface tension force in a 2D fluid simulation. The red particles are detected by the system as the surface particles while black particles reside deeper in the fluid body. Detection of surface particles is based on comparing the magnitude of the particle normal vectors, as explained in Section 5.1. In the absence of gravity and the environmental viscous drag, fluid body takes a drop-like shape and ossilates along the axes, which is an expected behavior under these conditions.

Section 3.2.3 gives the theoretical background of capillary action. To implement porosity and capillary forces in our particle-based simulation system, we associate each particle i of a porous object with porosity ϕ_i and void volume V_i [79]. Then, the total capillary force acting on the neighboring fluid particle j can be computed by:

$$\mathbf{f}_j^{capillary} = \sum_i p_i^{capillary} \nabla W(\mathbf{r}_{ij}, h), \tag{5.11}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ and $\mathbf{r}_i$ and $\mathbf{r}_j$ are the positions of particles i and j, respectively. h is the kernel radius and $p_i^{capillary}$ is the capillary pressure associated with particle i. $p_i^{capillary}$ is computed as explained in Section 3.2.3.

The porous object's weight increases because of the absorbed fluid mass and this can be simulated by adding neighboring fluid particles' mass to porous object particles. This addition is achieved by the following Equation:

$$m_i = m_i + \sum_j m_j W, \tag{5.12}$$

where m_i is the mass of the porous object particle, m_j is mass of neighboring fluid particle, and W is the smoothing kernel as defined by Equation 3.39. The saturation of the porpous object is modified according to Equation 3.30.

5.4 Simulating Miscible and Immiscible Fluids

Miscibility is a property of fluids to mix at any proportions, resulting in a homogeneous solutions. The term immiscibility, on the other hand, is used for the fluids that do not mix and retain a detectable boundary. Degree of miscibility can be quantified by dispersion, denoted by D, which stands for the mass transfer from highly concentrated regions to less concentrated regions. The convection-diffusion equation models the transfer of solute as follows [158]:

$$\frac{dC(t)}{dt} = D(\nabla^2 C(t)), \tag{5.13}$$

where C is concentration. For practical purposes, we can assume that the fluid density is linearly related to the concentration of solute. For SPH, this linear relation can be expressed by the following equation for each fluid particle i [132]:

$$m_i = m_i^f + \alpha\, C_i, \tag{5.14}$$

where C_i is concentration associated with particle i, m_i^f is fluid mass of i, and α is a constant. Equation 5.13 can be adopted to SPH as follows [157]:

$$\frac{dC_i}{dt} = \sum_j \frac{(D_i n_i + D_j n_j)(C_i - C_j)}{n_i n_j (\mathbf{r_i} - \mathbf{r_j})^2} (\mathbf{r_i} - \mathbf{r_j}) \nabla W(\mathbf{r_i} - \mathbf{r_j}, h), \tag{5.15}$$

where D_i is dispersion associated with i, C_i and n_i are concentration and number density of particle i, respectively. Number density is computed as in:

$$n_i = \sum_j W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{5.16}$$

At each simulation step, particle masses and concentrations are updated by Equations 5.14 and 5.15, respectively.

Figure 5.7 illustrates the effect of different dispersion coefficients. In the simulations, there are two fluid bodies with different concentration (and same

dispersion coefficients), pink fluid having $C = 1.0$ and white having $C = 0.1$. Because of the higher concentration, pink fluid tends to flow down by the effect of gravity. In the top row of the figure, $D = 0$, which results in immiscible fluid flow. Pink fluid does not mix with the white fluid and settles down at the bottom of the containers. In the middle row, $D = 0.2$, so that fluid bodies mix with each other, white fluid getting pinkish. In the bottom row, the dispersion coefficient is higher resulting in a faster mixing.

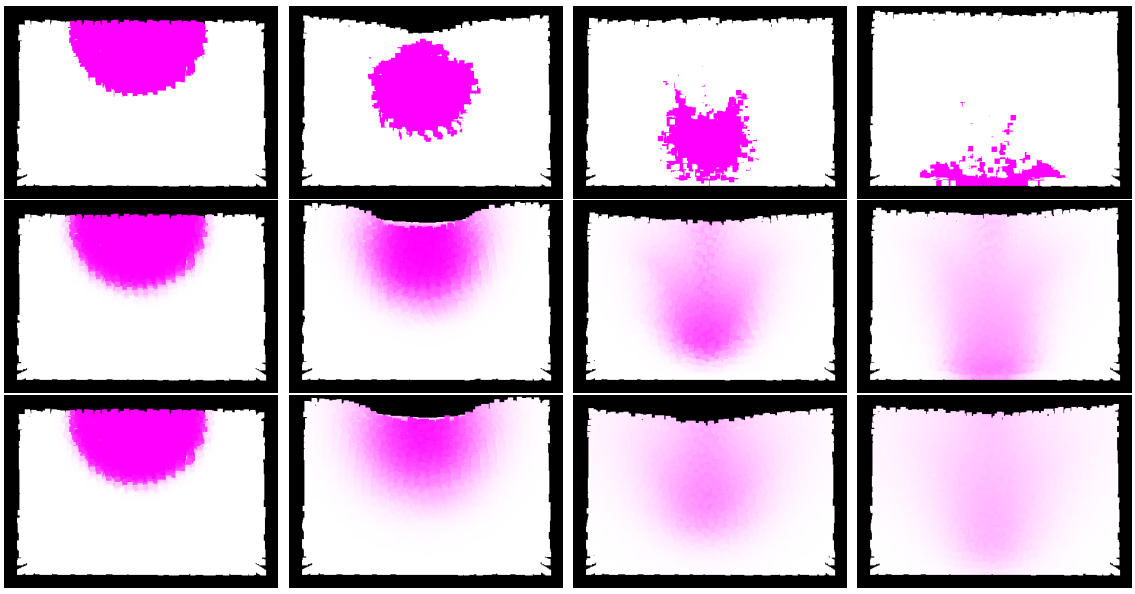


Figure 5.7: Simulation of miscible fluid flows with different dispersion coefficients, D. In the top row, $D = 0$, in the middle row, $D = 0.2$, and in the bottom row, $D = 0.45$.

5.5 Neighbor Search

In most of the particle based simulation methods, particles interact only with a small set of particles at each simulation step. The naive approach of considering the whole particle set for interparticle forces computation results in $O(n^2)$ run time complexity. It is therefore important to employ precise and fast neighbor search algorithm to improve simulation performance. Even then, the neighbor search usually dominates the run time of a particle based simulation system.

Several methods have been proposed to optimize neighbor search in particle based simulation systems. Most of these methods employ some kind of space subdivision approach. Space subdivision methods usually employ a uniform grid [137, 70] and discretize the simulation space to improve the performance of contact detection. Bounding volumes [138], and Binary Space Partitioning (BSP) trees [94] are among other methods of improving contact detection performance. Grid based approaches as used in SPH based particle simulation systems employ a uniform grid with voxel size of $2h$. We propose a neighbor search algorithm that uses a uniform grid to subdivide the simulation space and speed up neighbor detection step. Particles are sorted with respect to their discretized grid locations. Following sections give the details of our sorting-based neighbor search algorithm.

5.5.1 Search Algorithm

Algorithm 1 outlines the proposed neighbor search algorithm. The first step of the neighbor search algorithm is to discretize particle coordinates with respect to the grid to obtain integral positions (r_{ix}, r_{iy}, r_{iz}) of each particle i. The vortex size of the uniform grid is chosen to be $2h$, h being the kernel radius of the SPH algorithm. These grid coordinates are then converted to 1D coordinates by Equation 5.17 and stored in a 1D array *particleCoordinate*:

$$r_{ix} + r_{iy} * gridWidth + r_{iz} * gridWidth * gridHeight. \tag{5.17}$$

The *particleCoordinate* array is sorted first with respect to vortex ids and and then particle ids by employing the Radix Sort algorithm. In another pass, we determine the first and last position of each vortex within the sorted *particleCoordinate* array and store these pointers in the *vortexPointer* array (Figure 5.8). We, then, scan the *particleCoordinate* array and determine potential neighbors of each particle. In order to achieve this, constant offsets are employed for each of the 26 neighbor vortices of particle's host vortex. For example, $(i-1)^{th}$ vortex is the left neighbor of i^{th} vortex and $i + grid\_width^{th}$ vortex is the upper neighbor of the i^{th} vortex.

```
1  for each time step do
2      compute the array particleCoordinate;
3      radix sort the array particleCoordinate
4      scan particleCoordinate and compute vertexPointer array
5      forall the fluid particle p_i do
6          for each of 27 neighbor grid cells k do
7              for j ←vertexPointer[k] to vertexPointer[k+1] do
8                  while i < particleCoordinate[j] do
9                      check whether particle i and particleCoordinate[j] are
                       neighbors
10                 end
11             end
12         endfor
13     end
14 end
```

Algorithm 1: The proposed neighbor search algorithm

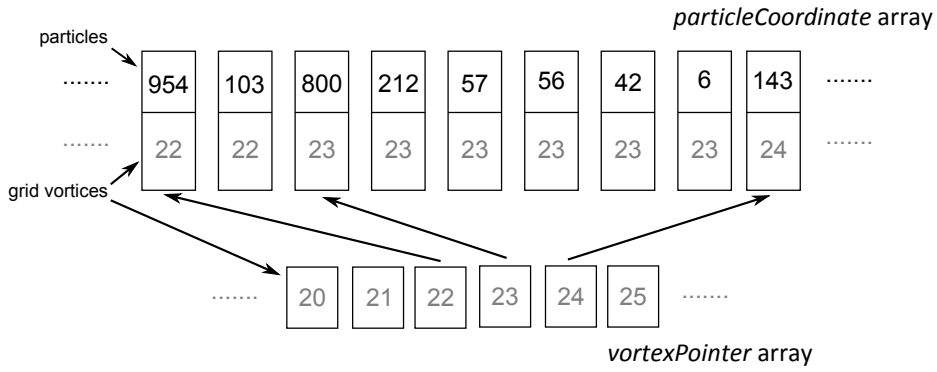


Figure 5.8: The sorted *particleCoordinate* array and *vortexPointer* array

Figure 5.9 illustrates the algorithm where potential neighbors of particle 57 are being searched (within a virtual grid of 30 vortices width, with particle distribution as seen in Figure 5.10). The algorithm goes through the particles of vortex 23 (particle 57's own vortex), of vortex 24 (the right neighboring vortex), and of vortex 53 (the upper neighboring vortex). Start pointers of the vortices in the *particleCoordinate* array is read from the *vortexPointer* array. To consider each potential neighbor pair exactly just once, we scan particles in ascending order and stop when we reach a particle with smaller id. In the example we stop at particle 42 and particle 34.

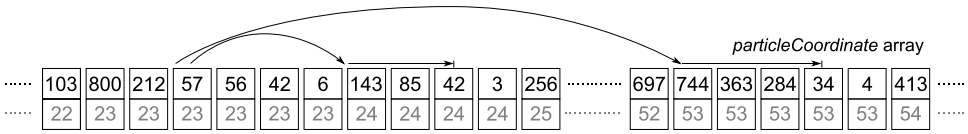


Figure 5.9: Potential neighbors of particle 57 is being searched.

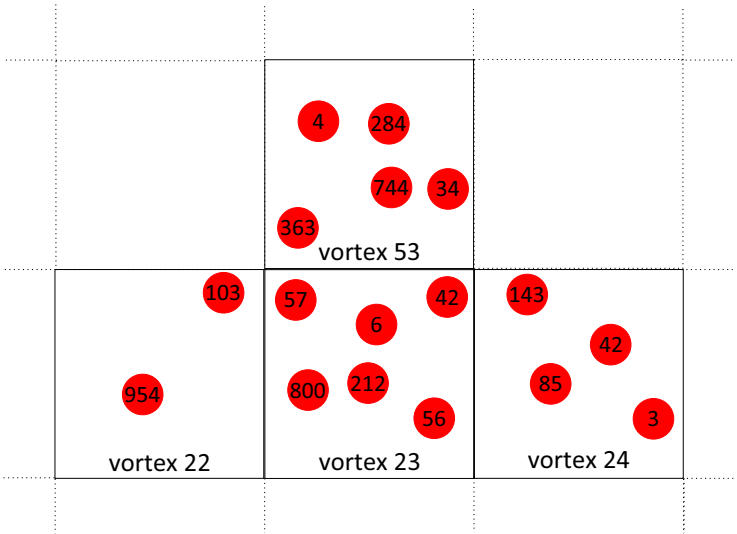


Figure 5.10: Sample grid configuration.

5.6 Numerical Integration

In particle-based simulation systems particles' positions are updated according to the Newton's second law of motion. In mathematical notation, the position of the particle i is computed by the following Equation:

$$\mathbf{x}_i = m_i \, \ddot{\mathbf{x}}_i, \tag{5.18}$$

where $\mathbf{x}_i$ and m_i are the position and mass of the particle i, respectively. $\ddot{\mathbf{x}}_i$ denotes the second time derivative of $\mathbf{x}_i$ (i.e. acceleration) in Newton's notation. This expression is an ordinary differential equation that is solved numerically. Numerical ordinary differential equations is a vast topic in itself and discussing it in its full details is out of the scope of this thesis. We will introduce some concepts about the numerical methods of solving particle movement numerically and discuss some of the alternatives.

Within the context of physically-based simulation methods, numerical ordinary differential equations can be classified into two main categories: implicit methods and explicit methods. Implicit methods are more suitable for physically based simulations than explicit methods since explicit methods consider each particle independently which causes errors especially in mass-spring simulations where particles are coupled by strong force constraints. On the other hand, implicit methods are much more complex to model and implement. This is the especially case in case of parallel programming. In the following sections, we discuss some explicit and implicit integration methods.

5.6.1 Explicit Methods

The simplest method for updating the position and velocity of a mass point is the Euler's method. The general expression of the Euler's method is:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t), \tag{5.19}$$

where Δt is the step size and $\dot{\mathbf{x}}$ the first time derivative of $\mathbf{x}$. Thus, by the Euler's method integrating for the particles' updated position is fairly simple. Equations 5.20, 5.21, and 5.22 express the Euler's method:

$$\mathbf{a}(t + \Delta t) = \frac{1}{m}\, \mathbf{f}(t), \tag{5.20}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t\, \mathbf{a}(t + \Delta t), \tag{5.21}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \mathbf{v}(t + \Delta t), \tag{5.22}$$

where $\mathbf{a}$, $\mathbf{v}$ and $\mathbf{x}$ are particle acceleration, velocity, and position.

Although it is very simple to implement, the Euler's method is not efficient nor accurate especially for stiff equations. This makes the Euler method unsuitable for particle based simulations.

As an alternative to the Euler's method, one can choose to use the fourth-order Runge-Kutta method, which, gives more accurate results with larger time steps. The generic expression of the fourth-order Runge-Kutta method is as follows [116]:

$$k_1 = f(x_0, t_0)$$

$$k_2 = \Delta t\ f(x_0 + \frac{k_1}{2}, t_0 + \frac{\Delta t}{2})$$
$$k_3 = \Delta t\ f(x_0 + \frac{k_2}{2}, t_0 + \frac{\Delta t}{2})$$
$$k_4 = \Delta t\ f(x_0 + k_3, t_0 + \Delta t)$$
$$x(t_0 + \Delta t) = x_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4$$

Although it allows using larger time steps than Euler integration, Runge-Kutta method introduces a high computational overhead because of the fact that each time step requires 4 force computations. This overhead is especially prominent when force computations involve recomputing neighbor particle sets and density computations as in the case of fluid simulations of the discrete element method.

As it is clear from the formula, the fourth-order Runge-Kutta method uses a constant step size. That is, time step should explicitly be determined so that it is not that large to blow up the calculations or that small to slow down the simulation unnecessarily. Theoretically efficiency of the method can be improved by employing adaptive step size control. A numerical solver with adaptive step size control tries to achieve some predetermined accuracy by using as large time steps as possible. This is done by using small time steps where the steepness of the function is high and large time steps where the steepness is low. A possible method for step size can be so called *step doubling*. In this method the algorithm takes each step twice, once as a full step, then, independently, as two half steps. This method requires 11 function evaluations. An alternative method found by Fehlberg uses a fifth-order method in order to estimate truncation error [116]. The fact that this method requires 6 function evaluations may cause worse run times than the fourth-order constant step sized Runge-Kutta.

Another method of implicit numerical integration method is the Verlet algorithm which has been used especially in molecular dynamics [145]. The Verlet algorithm is more accurate than the Euler's method and easier to implement than the fourth-order Runge-Kutta method. The position update according to Verlet integration is computed by:

$$\mathbf{x}(t + \Delta t) = 2\ \mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \mathbf{a}\ \Delta t^2, \tag{5.23}$$

where $\mathbf{x}(t+\Delta t)$ is the updated position, $\mathbf{x}(t)$ is the current position, $\mathbf{x}(t-\Delta t)$ is the position from the previous time step, and $\mathbf{a}$ is the acceleration vector. Since its fomulation does not involve the velocity term, the Verlet method makes it very easy to establish distance constraints between particles [59] which can be very usable to construct a basic physically based simulation system. On the other hand, not computing the velocity explicitly can be problematic since velocity is generally used in several of the force computations such as viscous and frictional forces. This can be alleviated by explicitly computing the velocity by using the position:

$$\mathbf{v}(t+\Delta t) = \frac{\mathbf{x}(t+\Delta t) - \mathbf{x}(t-\Delta t)}{2\Delta t}. \tag{5.24}$$

Instead, the velocity Verlet algorithm [3, 131], which has the same accuracy as the Verlet method, can be used. It is also called the Leapfrog method and is a variation of the common Verlet algorithm. Its error in position and velocity is $O(\Delta t^3)$. The velocity Verlet algorithm updates velocity and positions according to the following equations,

$$\begin{aligned} \mathbf{x}(t+\Delta t) &= \mathbf{x}(t) + \mathbf{v}(t)\Delta t + (1/2)\,\mathbf{a}(t)\Delta t^2, \\ \mathbf{v}(t+\Delta t/2) &= \mathbf{v}(t) + (1/2)\,\mathbf{a}(t)\Delta t, \\ \mathbf{a}(t+\Delta t) &= (1/m)\,\mathbf{f}, \\ \mathbf{v}(t+\Delta t) &= \mathbf{v}(t+\Delta t/2) + (1/2)\,\mathbf{a}(t+\Delta t)\Delta t, \end{aligned} \tag{5.25}$$

where m is the mass, $\mathbf{x}$ is the position, $\mathbf{v}$ is the velocity, $\mathbf{a}$ is the acceleration of the particle and $\mathbf{f}$ is the total force acting on the particle.

Explicit numerical solutions for differential equations should satify the Courant-Friedrichs-Lewy condition (CFL condition) [26]. The CFL condition constraints time step by some computable quantity.

In the SPH-based fluid simulations one of the constraints on time step is due to the magnitude of particle accelerations [97],

$$\Delta t \leq \alpha \ \min_i \sqrt{\frac{h}{|\mathbf{a}_i|}}, \quad 0 \leq \alpha \leq 1.0, \tag{5.26}$$

where h is the smoothing kernel radius and α is the CFL coefficient. Another constraint is due to particle velocities $\mathbf{v}$,

$$\Delta t \leq \alpha \ \min_i \frac{h}{|\mathbf{v}_i|}, \quad 0 \leq \alpha \leq 1.0, \tag{5.27}$$

In mass-spring systems, the time step is limited by the stiffness of the system defined by spring coefficients. One upper bound for time step is then can be defined as follows [66]:

$$\Delta t \leq 2 \ \frac{m_{min}}{K}, \tag{5.28}$$

where m_{min} is the minimum mass of particles and K is defined as:

$$K = \sum_i k_i, \tag{5.29}$$

for each spring constant k_i. This upper bound is an upper bound for the worst case where each of the spring is connected in parallel between to mass particles. This cannot be the case in a cloth animation where the mass-spring mesh is constructed in such a way that two mass points are connected with only one spring (see Section 4.1.1). Then K in Equation 5.28 can be defined as:

$$K = \max_i k_i. \tag{5.30}$$

5.6.1.1 Implicit Methods

A simple application of implicit integration method is the implicit Euler Method. Recall that the explicit Euler method is in the following form:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{f}(t)\frac{\Delta t}{m} \tag{5.31}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \Delta t)dt \tag{5.32}$$

where $\mathbf{v}(t)$ and $\mathbf{v}(t + \Delta t)$ are the particle velocities at time t and $t + \Delta t$, respectively. Likewise $\mathbf{x}(t)$ and $\mathbf{x}(t + \Delta t)$ are the positions and $\mathbf{f}(t)$ and $\mathbf{f}(t + \Delta t)$ are the forces at time t and $t + \Delta t$, respectively. It should be noted that the forces from the previous time step contribute to the positions at the next time step. This fact introduces the aformentioned Courant condition [116]. According to this criterion the integration time step is inversely proportional to the square

root of the stiffness. In this case we have to keep the time step small enough to prevent the system from blowing up. Otherwise, a large time step can induce huge changes in position. An alternative in this case is to use an implicit integration method. Implicit integration methods can provide us time steps large enough for an interactive simulation, if the large linear systems they produce are approximated by the mentioned, or another similar, way.

The implicit Euler method uses the forces at time $t + \Delta t$ instead of the forces at time t. Thus the equations of the Euler method are as follows:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{f}(t + \Delta t)\frac{\Delta t}{m} \tag{5.33}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \Delta t)\Delta t \tag{5.34}$$

In this case the position in the next time step corresponds to the forces of the next time step. In theory, with any value of the time step, we calculate the positions coherent with the forces. Thus, the numerical solver cannot give rise to any instabilities. Problem with this method is that it involves the term $\mathbf{f}_i^{t+\Delta t}$. Fortunately $\mathbf{f}_i^{t+\Delta t}$ can be approximated by the following equation:

$$\mathbf{f}(t + \Delta t) = \mathbf{f}(t) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\Delta \mathbf{x}(t + \Delta t) \tag{5.35}$$

where $\mathbf{f}(t)$ stands for the internal energy of the system. The expression $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is a negated Hessian matrix and denoted by H [29]. Since

$$\Delta \mathbf{x}(t + \Delta t) = \mathbf{x}(t + \Delta t) - \mathbf{x}(t) = (\mathbf{v}(t) + \Delta \mathbf{v}(t + \Delta t))\Delta t \tag{5.36}$$

the equation 5.33 can be expressed by the following equation:

$$(I - \frac{\Delta t^2}{m}H)\Delta \mathbf{v}(t + \Delta t) = (\mathbf{f}(t) + \Delta t H \mathbf{v}(t))\frac{\Delta t}{m} \tag{5.37}$$

Since $\Delta \mathbf{v}(t + \Delta t) = \mathbf{v}(t + \Delta t) - \mathbf{v}(t)$, we are to find $\mathbf{v}(t + \Delta t)$. In Equation 5.37, $\Delta t\ H\ \mathbf{v}(t)$ represents the artificial viscosity forces added to create a dissipative force in mass-spring or particle systems [95]. It can be calculated by the following equation:

$$\Delta t\ H\ \mathbf{v}(t) = \Delta t \sum_{(i,j)\in E} k_{i,j}(\mathbf{v}_j(t) - \mathbf{v}_i(t)) \tag{5.38}$$

where E is the set of the spring edges in the mass-spring system. This artificial viscosity is proportional to the time step and the stiffness of the system which are responsible for the instability of the integration process. Thus, adding this artificial viscosity introduces additional stability.

In order to update the system the expression $I-\frac{\Delta t^2}{m}H$ in Equation 5.37, which is an $O(n\times n)$ sized matrix where n is the number of mass points, should be solved at each time step. In [4] Baraff et. al. employ a modified conjugate-gradient method in order to ease this computational burden. Another method which is argued to be faster is proposed by Desbrun et. al [29] where the Hessian matrix H is approximated in the following manner. The entry of the Hessian matrix $H_{i,j}$, where i is the row number and j is the column number, is approximated as $H_{i,j} = k_{i,j}$, and $H_{i,i} = -\sum_{j\neq i} k_{i,j}$, where $k_{i,j}$ is the stiffness of the spring connecting the mass points i and j. $k_{i,j}$ is 0 when the mass points i and j are not linked. By this method the matrix $(I - \frac{\Delta t^2}{m}H)^{-1}$ remains constant during the animation alleviating the computational burden. The inverse matrix is, then, precomputed and used as a filter during the animation. The equation summarizing the method proposed by Desbrun et. al. can be written as:

$$\Delta\mathbf{v}(t+\Delta t) = (I - \frac{\Delta t^2}{m}H)^{-1}\frac{\widetilde{\mathbf{f}}\Delta t}{m} \tag{5.39}$$

where $\widetilde{\mathbf{f}}$ is the sum of the spring forces and the viscosity forces. Although using the precomputed filter eases the computational burden, it has several disadvantages. One of them is that it is not possible to use the adaptive time-step approach. Moreover, it is impossible to change mass or stiffness and the matrix $(I - \frac{\Delta t^2}{m}H)^{-1}$ is not always a sparse matrix.

Young et. al. [64] propose another approximation method in which they assume a uniform spring constant for the sake of simplicity. They, first, rewrite Equation 5.37 in the following manner:

$$(I - \frac{\Delta t^2 H_{ii}}{m_i})\Delta\mathbf{v}_i - \frac{\Delta t^2}{m_i}\sum_{(i,j)\in E}(H_{i,j}\Delta\mathbf{v}_j) = \frac{\widetilde{\mathbf{f}}\Delta t}{m_i} \tag{5.40}$$

In order to update the velocity change of the ith mass point they only consider the linked mass points, because when ith and jth mass points are not connected,

$H_{i,j}$ is 0. Since they assume a uniform spring constant all over the mass-spring network, the Hessian matrix can be rewritten as $H_{i,j} = k$ and $H_{i,i} = kn_i$ where k is the spring constant and n_i is the number of the springs connected to the ith mass point. Thus, the update equation 5.40 can be written as:

$$\frac{m_i + \Delta t^2 k n_i}{m_i} \Delta \mathbf{v}(t + \Delta t) = \frac{\widetilde{\mathbf{f}}_i^t \Delta t}{m_i} + \frac{\Delta t^2 k \sum_{(i,j)\in E} \Delta \mathbf{v}_j(t + \Delta t)}{m_i} \tag{5.41}$$

By using Equation 5.41, we can express $\Delta v_i(t + \Delta t)$ as follows:

$$\Delta \mathbf{v}_i(t + h) = \frac{\widetilde{\mathbf{f}}_i(t)h + k\Delta t^2 \sum_{(i,j)\in E} \Delta \mathbf{v}_j(t + \Delta t)}{m_i + k\Delta t^2 n_i} \tag{5.42}$$

which includes another unknown expression $\Delta \mathbf{v}_j(t + \Delta t)$ which is the velocity change of the jth mass points linked to the ith mass point. Young et. al. worked out this problem by expressing $\Delta \mathbf{v}_j(t + \Delta t)$ as follows:

$$\Delta \mathbf{v}_i(t + \Delta t) = \frac{\widetilde{\mathbf{f}}_j(t)\Delta t + \Delta t^2 \sum_{(j,l)\in E} k_{jl} \Delta \mathbf{v}_l(t + \Delta t)}{m_j + \Delta t^2 \sum_{(j,l)\in E} k_{jl}} \tag{5.43}$$

and by dropping the term $\Delta t^2 \sum_{(j,l)\in E} k_{jl} \Delta \mathbf{v}_l(t + \Delta t)$ to get the following approximation:

$$\Delta \mathbf{v}_j(t + \Delta t) \simeq \frac{\widetilde{\mathbf{f}}_j(t)\Delta t}{m_j + \Delta t^2 \sum_{(j,l)\in E} k_{jl}} \tag{5.44}$$

After this approximation the formula for $\Delta v_i t + \Delta t$ becomes:

$$\Delta \mathbf{v}_i(t + \Delta t) = \frac{\widetilde{\mathbf{f}}_i(t)\Delta t + \Delta t^2 k \sum_{(i,j)\in E} \frac{\widetilde{\mathbf{f}}_j(t)\Delta t}{m_j + \Delta t^2 k n_j}}{m_i + \Delta t^2 k n_i} \tag{5.45}$$

By using Equation 5.45, we can calculate $\mathbf{v}_i(t + \Delta t)$ and then, $\mathbf{x}_i(t + \Delta t)$ by employing Equation 5.34.

Although implicit numerical integration methods are intrinsicly stable, they are not easy to implement. It is even more diffucult to implement implicit methods in GPGPU or another parallel architecture. It is also not trivial to incorparate position or force constraint with the implicit methods. Thus, an explicit integration method with a high order of error such as the velocity Verlet is an optimal solution.

Chapter 6

GPU-based Particle Simulation

Previous chapters present theoretical and practical backgrounds of particle-based simulation of fluids and cloth like deformable objects and their interaction. In this Chapter, we detail the implementation of a particle-based simulation system on common purpose *Graphics Processing Units* (GPUs). GPUs provide cheap, accessible and easy to use parallel computing environment for personal computers. Exploiting the parallel computing capabilities of GPUs and parallel nature of particle-based simulation systems, performance of such applications can be improved greatly.

6.1 Introduction

In the last decade, common purpose graphics boards have experienced a very rapid improvement in terms of processing power and programmability. Pushed by the demand of gaming industry for faster, stronger and cheaper boards, GPU producers have been developing more powerful GPUs. Thus, the increase of GPU performance has been much more rapid than that of CPUs have been experiencing as illustrated in Figure 6.1. This growing computational power of GPUs has attracted the attention of not only game developers but also software developers who want to exploit computational power of graphics boards for numerical

computations. To provide better development tools to researchers, GPU producers have been developing better programming interfaces. Early examples of numerical computations on GPUs have used the graphical programming interfaces (shading languages), which are essentially aimed at graphics programming applications. Using shading languages for numerical computations is usually named *General-Purpose computing on Graphics Processing Units* (GPGPU) [90].

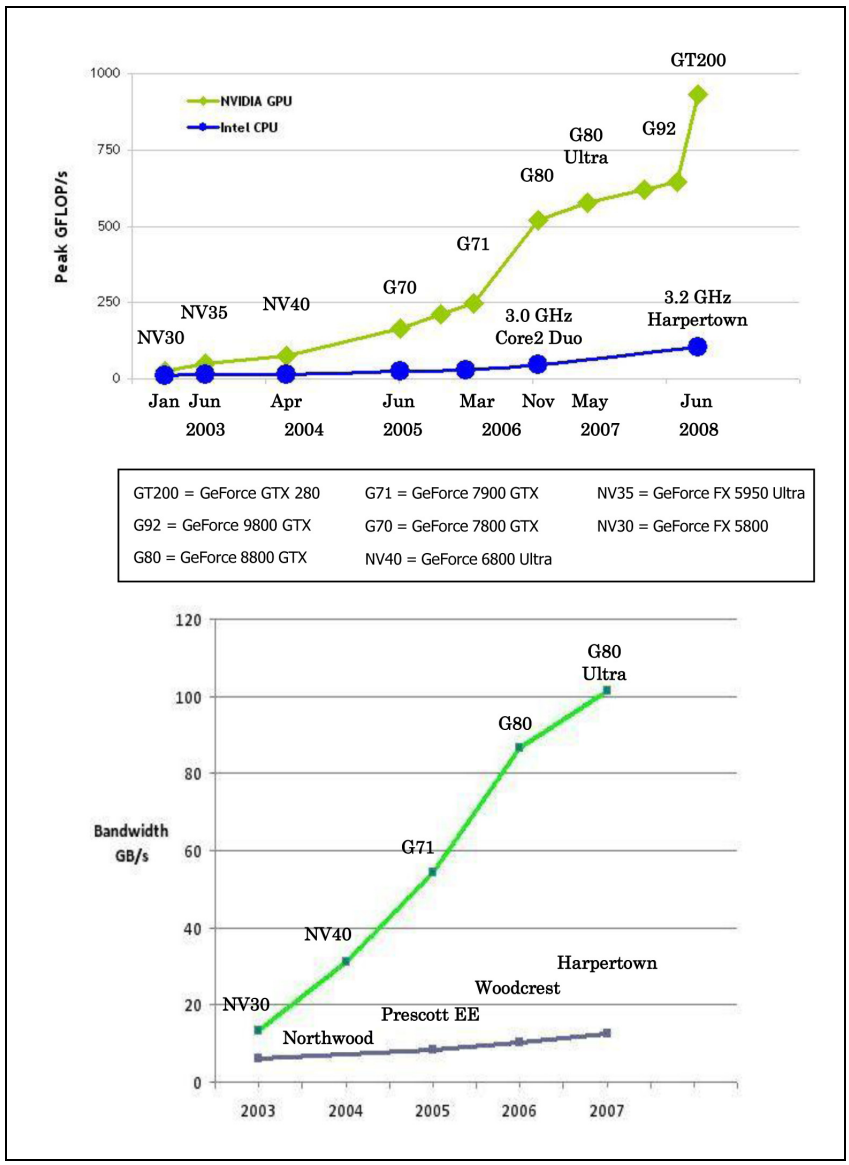


Figure 6.1: Floating-Point operations per second and memory bandwidth for the CPU and GPU [109].

GPUs are designed to execute similar operations on several data members in

parallel. Using GPUs is especially beneficial for the tasks where a relatively small amount of operations are applied to a large amount data. In GPUs, multiple vertices, geometric primitives and fragments are processed in parallel through a pipeline, which is called the Graphics Hardware Pipeline [36]. The graphics pipeline is divided into several logical stages and their corresponding hardware components (shaders), as illustrated in Figure 6.2. Through the development of GPUs several of these components were made programmable.

The shader responsible of transforming the incoming vertices is called *the Vertex Shader*. The vertex shader can perform a series of mathematical operations on vertices and in modern GPUs it can read data from textures. This stage is also called the vertex transformation. The next stage in the graphics pipeline is executed by *the Geometry Shader*. The programmable geometry shader is a relatively new addition to GPGPU and it can generate new graphics primitives additional to those that are sent to the pipeline initially. The third stage of the graphics pipeline is executed by *the fragment* or *pixel shader*, which operates on every pixel. The fragment shader executes mathematical operations on each of the pixels and it can lookup from and write into textures. Section 6.2 presents a particle simulation system implemented by using a shading language.

Nvidia released CUDA™(Compute Unified Device Architecture) in November 2006, which is a parallel computing architecture. It is designed to run on Nvidia GPUs from the G8X onwards and uses an extension to the C language although some other high level languages are supported as well. CUDA is different from the shading languages both in terms of its motivation and design. By definition, shading languages follow the graphics pipeline in their programming style and syntax. This introduces a high learning curve for non-graphics programmers. CUDA, on the other hand, introduces a set of C programming language extensions and constructs that are easier to learn. CUDA provides programmers with a versatile and powerful, both low and high level, APIs to harvest the parallel processing power of modern GPUs.

In CUDA terminology the CPU and main memory is called the host and GPU and video memory is the device. CUDA functions (processing constructs)

are called *kernels* and they are invoked by the host functions. Kernels process data residing on the device's global memory and execute a number of threads in parallel. These threads are grouped into blocks. Other than the global (and relatively slow) memory, each threads belonging to the same block share an on-chip memory space (the *shared memory*). Moreover, CUDA has the texture and constant memory spaces which are both read only and have shorter access times than the global memory. CUDA has some built-in vector types which are suitable for a physical based simulation. *float3* and *float4* vector types have 3 and 4 components, respectively, and these components are accessed through the fields x, y, z, and w (in case of *float4*).

In Section 6.3, we present the details of a particle-based simulation system implemented in CUDA. The system simulates fluid behavior and its interaction with rigid and cloth like objects.

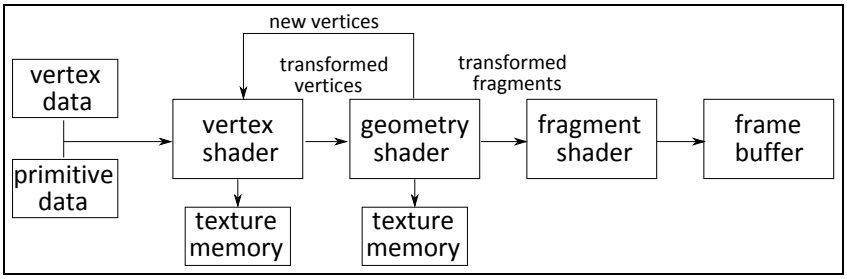


Figure 6.2: The overview of the graphics pipeline.

6.2 Particle-Based Simulation by GPGPU

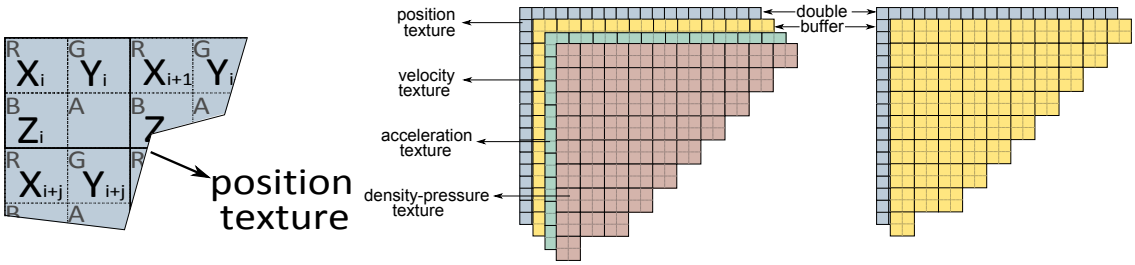
Particle-based simulations, such as SPH, mass-spring networks, and DEM, are perfect candidates to be implemented on GPUs because of GPUs ability to process multiple particles in parallel [114]. In this and following sections, we describe a simulation system where parallel processing power of GPUs are exploited by GPGPU for the purpose of particle simulations. Specifically, we have implemented our particle system by using Nvidia's shading language Cg (C for Graphics) [91].

In GPGPU, 2D textures are used to store the data where each texture element has four components corresponding to RGBA values. The simulation data that would be stored in arrays in a CPU implementation are instead stored in 2D textures. Particle positions, velocities, acceleration, which are 3D vectors are stored such that xyz components correspond to RGB values in 2D textures, as illustrated in Figure 6.3(a). For some of the particle properties such as position and velocity, two 2D textures are used as double buffer, as shown in Figure 6.3(b). Scalar particle properties can be combined into single texture. For example, particle densities and pressures are stored in R and G channels of the density-pressure texture, respectively.

The processing element we use in our simulations is the pixel (fragment) shader. Fragment shader is executed by drawing a full-screen quad. Drawing such a rectangle instructs the graphics hardware to call the fragment shader for each of the data elements. The output textures are, usually, targeted not to screen but some output textures. This is achieved by the frame buffer (FBO) extension of OpenGL. FBO enables the fragment shader to do an off-screen rendering.

Figure 6.4 (a) illustrates the flow diagram of our Cg implementation of SPH. The initial step of the simulation is one-time setup where GPU memory (textures) for the particle data are allocated and initialized. Once the particle data is initialized on GPU memory, it runs entirely on GPU avoiding data transfers between the main memory and video memory. The memory foot print of the entire system on the CPU memory is negligible.

The first part of the simulation loop is the neighbor search where we construct grid-particle map to be used in neighbor lookups. Then particle densities, internal fluid forces, boundary and rigid object interaction forces are computed. In the next step, the particle positions are updated by the numerical integration with respect to net force acting on each particle. The output of this numerical integration step is a render target so that the updated particle positions are rendered directly. In the following sections, we give the details of each of these simulation steps.



(a) (b)

Figure 6.3: (a) Particle positions are stored in a 2D texture. R, G, B, and A are the channels of each texture element and X_i, Y_i, and Z_i are the components of the 3D position vector of the particle i. (b) Particle data are stored in several 2D textures. For position and velocity double buffering is used.

6.2.1 GPGPU-based Neighbor Search

The biggest challenge of implementing a particle-based simulation by a shading language is to detect particle proximities efficiently. This is because of the fact that the fragment shaders that we use as the main processing units are not capable of *scatter*. That is, they cannot write a value to a memory location for a computed address since fragment programs run using precomputed texture addresses only and these addresses cannot be changed by the fragment program itself. This limitation makes several basic algorithmic operations (such as counting, sorting, finding maximum and minimum) more complicated.

One of the common methods for detecting the set of neighboring particle is to use a uniform grid to subdivide the simulation space. Kipfer et al. [69] use a uniform grid and sorting mechanism to detect inter-particle collisions on GPU. Purcell et al. [118] also use a sorting based grid method. They employ stencil buffer for dealing with multiple photons residing in the same cell. Harada et al. [51] present a SPH-based fluid simulation system on GPUs. Their system uses bucket textures to represent a 3D grid structure and make an efficient neighbor search. One limitation of their system is that it can only handle up to 4 particles within a grid cell.

We implement the neighbor search algorithm presented in Section 5.5 on GPU by using Nvidia's shading language Cg. Because of the aforementioned limitations

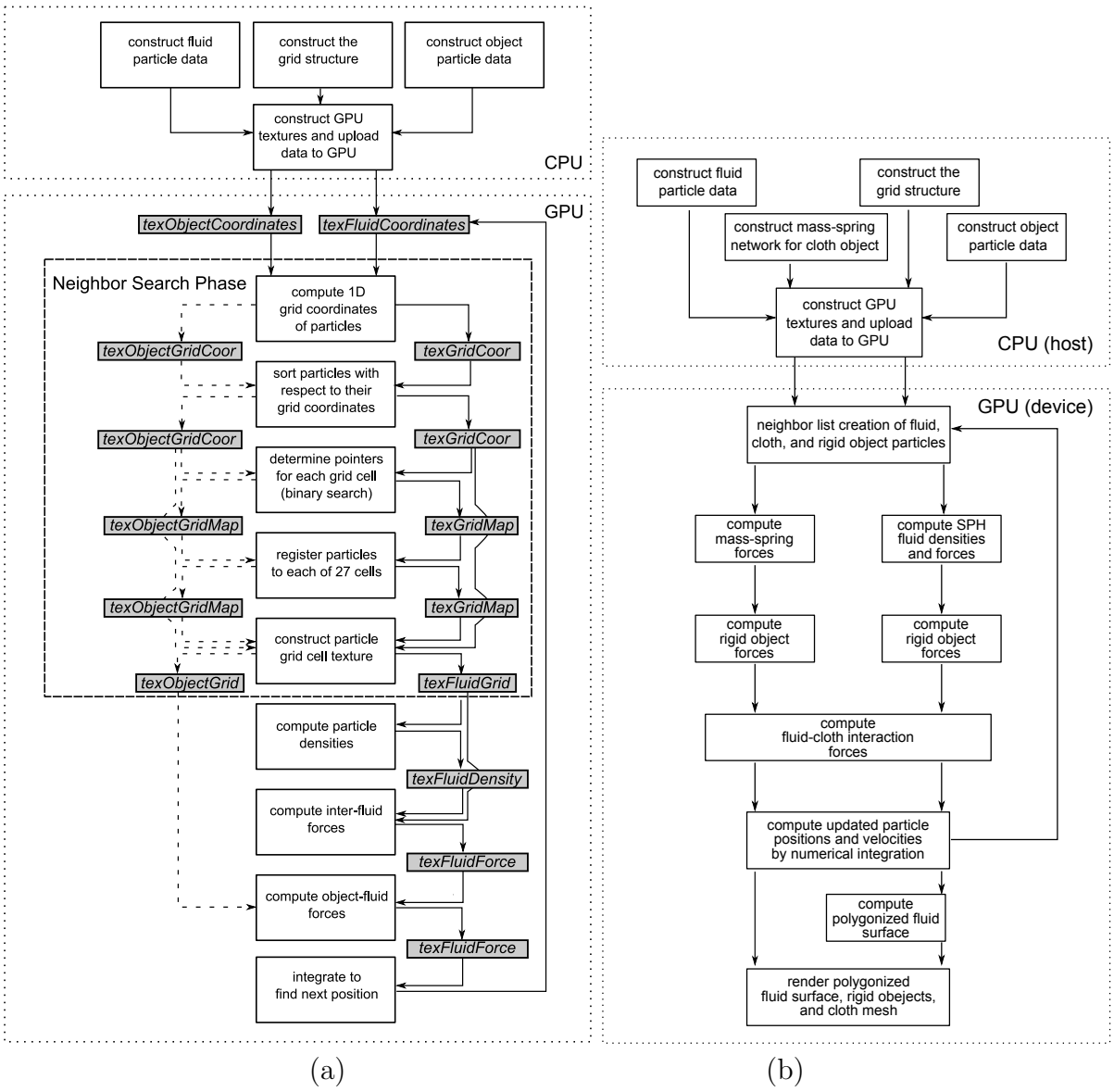


Figure 6.4: (a) The flow chart of the GPGPU-based SPH implementation. (b) The flow chart of the CUDA-based particle simulation system. In the system fluid and cloth particles interact with each other and the environment.

of shading languages, the algorithm has to be modified. The following section explains the modified version of the algorithm.

6.2.2 Grid Map Generation

Figure 6.4(a) illustrates the work flow of the GPU-based particle simulation that employs the presented technique to compute the particle neighborhood information. Each white rectangle in Figure 6.4(a) represents a rendering pass by a fragment program and each gray rectangle represents an RGBA float texture.

The first part of the neighbor search algorithm is to generate the grid map. The grid map stores the location information of particles per cell. By creating the grid map, we determine the number of particles in each cell and its immediate neighbors host, and IDs of these particles. Algorithm 2 outlines the grid map generation process.

Step 1 of the algorithm computes 1D grid coordinates of particles (Equation 6.1), and stores this information in the grid coordinate texture (*texGridCoor*) along with particle IDs. Step 2 sorts the texture *texGridCoor* with respect to computed 1D grid coordinates. The sorting phase employs the bitonic sort [118]. Bitonic sort is developed for parallel machines and makes $\log^2 N$ passes over the texture, N being the number of particles. It is a good choice for sorting data by shaders.

$$i_x + i_y \times grid\_width + i_z \times grid\_width \times grid\_height. \tag{6.1}$$

Step 3 searches *texGridCoor* texture for the first occurrence of each grid cell by using binary search. The fragment program searches the grid cell ID within the sorted *texGridCoor* and stores the first occurrence in the R channel of the grid map texture (*texGridMap*). If the grid cell does not host any particle, the fragment program stores a sentinel value in the texture.

Step 4 counts the total number of particles belonging to each grid cell and stores this information in the G channel. This is done by linear searching the *texGridCoor* texture from the position found in Step 3 until we hit the next grid cell ID.

Step 5 computes the total number of particles in and around of each grid cell. The fragment program sums the number of particles stored in G channels

of immediate neighbor grid cells. This information is stored in the B channel of the grid map texture. Step 6 accumulates the values in the B channel and stores this sum in the alpha channel.

```
1  for each grid cell i do
2  |  Step 1:
3  |    compute 1D grid coordinates of particles and store in texGridCoor;
4  |  Step 2:
5  |    sort texGridCoor with respect to 1D grid coordinates ;
6  |  Step 3:
7  |    find first occurrence of i using binary search on texGridCoor (output to
   |    channel R);
8  |  Step 4:
9  |    scan texGridMap to determine the number of particles belonging to i (output
   |    to channel G);
10 |  Step 5:
11 |    scan texGridMap to determine number of particles belonging to i and its
   |    immediate neighbors (output to channel B);
12 |  Step 6:
13 |    for j ← 0 to i do
14 |    |  sum the channel G values of j (output to channel A);
15 |    end
16 end
```

Algorithm 2: The algorithm for grid map generation

Figure 6.5 depicts a sample grid map texture. In this figure, i) the values in the R channel point to the first occurrence of the grid cell within the grid coordinate texture *texGridCoor*, ii) the values in the G channel are the total number of the particles hosted within the grid cell, and iii) B values are the total number of particles hosted in the grid cell itself and its immediate neighbors, and iv) the alpha channel values point to the first occurrence of the grid cell with the fluid grid texture (*texFluidGrid*), which is generated afterwards.

6.2.3 Generating Fluid Grid Texture and Neighbor Lookup

Algorithm 3 outlines the fluid grid texture generation. Fluid grid texture (*texFluidGrid*) is similar to grid coordinate texture (*texGridCoor*) in the sense that it contains particle IDs sorted with respect to their 1D grid coordinates. The difference between *texGridCoor* and *texFluidGrid* is that the latter has an entry for

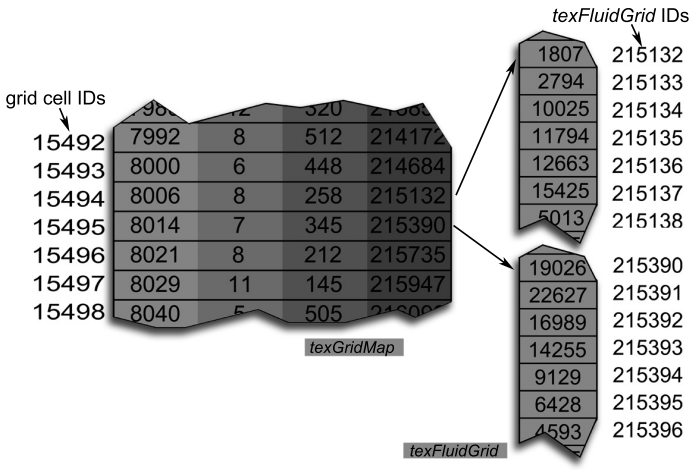


Figure 6.5: Grid map texture and fluid grid texture.

particle host cell and, additionally, an entry for each of the immediate neighbor cells of the host cell. Each particle appears up to 27 times in *texFluidGrid*; once for the host cell and 26 times (which could be less if the host cell lies on the grid boundary) for each neighboring cell.

```
1 for each texture coordinate i of texFluidGrid do
2 |   binary search the A channel of texGridCoor for i;
3 |   determine host cell c of i;
4 |   linear search i within particle list of c;
5 |   determine particle ID to be written into the current texture position i;
6 end
```

Algorithm 3: The algorithm for fluid grid texture generation.

To construct *texFluidGrid*, the fragment program has to decide which particle ID is to be stored in the current texture coordinate. Basically, the fragment program determines the grid cell hosting the particle by searching the particles using the alpha channel of the grid map texture *texGridMap*. Then by following the pointers of the grid cell to the grid coordinate texture *texGridCoor*, the fragment program finds the appropriate particle ID. For example, assume that the current texture coordinate is 215,425 and the current *texGridMap* is as shown in Figure 6.5. Binary searching for this coordinate in the alpha channel of the *texGridMap* reveals that the particle resides in the neighborhood of the grid cell 15,495. Then, to determine the particle ID to store in the current location, we go through the list of hosted particles in the neighborhood of grid cell 15,495.

After generating the textures *texGridMap* and *texFluidGrid*, finding possible neighbors of a particle is a simple task. We use the particle's 1D grid ID to look up the pointer to *texFluidGrid*. For example, if 1D grid cell coordinate of particle i is 15,494, then i has 258 potential neighbors starting with 1,807 (see Figure 6.5).

6.2.4 Density and Force Computations and Numerical Integration

The next step after neighbor search phase is to compute the scalar values of particle density and pressure, as defined by Equation 3.13 and Equation 3.18, respectively. The density-pressure computation step is implemented by a single fragment shader which outputs particle density and pressure values into the density-pressure texture.

The internal fluid forces are pressure and viscosity forces and they are defined by Equation 3.21 and Equation 3.22, respectively. The internal fluid forces are made symmetrical to satisfy the third law of the Newton's law of motion:

$$\mathbf{f}_{i,j}^{internal} = -\mathbf{f}_{j,i}^{internal}, \tag{6.2}$$

for each fluid particle pair i and j. The boundary forces and rigid body forces acting on fluid particles are computed by the discrete element method (DEM), which is explained in Section 4.3.

The internal fluid forces and rigid body boundary forces are computed by a single fragment shader, which takes particle positions, densities and pressures as input, and outputs net forces acting on each fluid particle.

Updating particle position from the new forces is done by the velocity Verlet integration, which is explained in Section 5.6. As mentioned in Section 6.2, to implement the velocity Verlet integration by fragment shaders, we use double buffers for particle positions and velocities. This is because the verlet algorithm takes particle positions (and velocities) as input and outputs new position values. However, fragment shader cannot output into the input texture. To work

around this limitation, we swap the input and output position (and velocity) textures at each time step. This practice is commonly called the the *Ping-Pong technique* [90].

The old fragment shaders can only output to one texture at a time. That limitation implies that the verlet algorithm as expressed in Equation 5.23 should be implemented in separate fragment shaders. That is, each step of the velocity verlet algorithm is implemented in a different fragment shader. Newer GPUs supports the *Multiple Render Targets* (MRT) extension by which fragment shaders can write values to multiple render targets [122]. With this capability, Verlet integration can be implemented by a single fragment shader.

6.3 Particle-Based Simulation in CUDA

We implement a SPH based fluid simulation system alongside with a mass-spring system in CUDA. The system is able to simulate cloth and fluid behavior and their interaction with each other and rigid objects. Figure 6.4(b) illustrates the main steps of the CUDA implementation.

The simulation starts with a one-time startup step which allocates and initializes the data structures for the particle system in the main memory (the host). The particle data structure includes 1D arrays of float3 or float4 vector types and they store coordinates, velocities, acceleration, and density-pressure values of particles.

To store the connectivity (spring neighborhood) of cloth particles in the cloth mesh we construct a simple data structure as illustrated in Figure 6.6. Since the the connectivity of the cloth mesh stays constant throughout the simulation, the depicted structure is created in the host and uploaded to the device. As Figure 6.6 shows, two lookups are needed to find the neighbors of a cloth particle.

After values of the particle data structure is copied over to allocated device

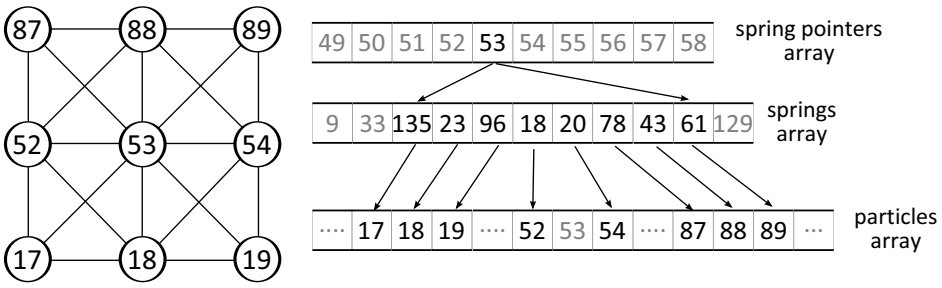


Figure 6.6: The constant cloth mesh connectivity is stored on the device memory as illustrated. Two lookups are performed to find the cloth particles that are connected to particle 53.

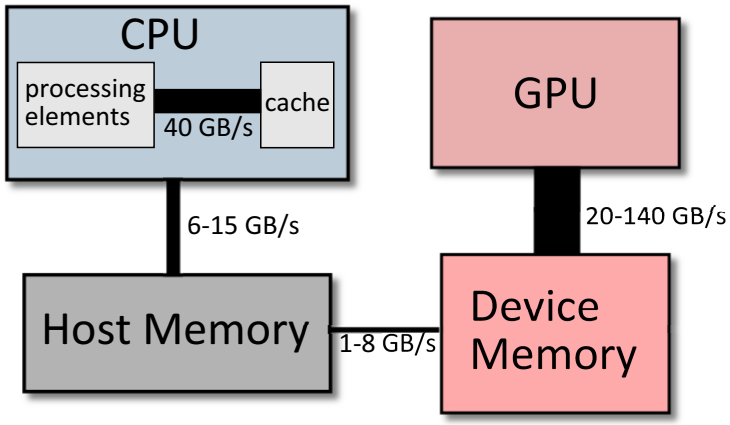


Figure 6.7: Memory bandwidths between and within CPU and GPU [109].

(GPU) arrays, the host arrays are deallocated. As in the GPGPU implementation, the entire simulation data resides on the device memory avoiding slow data transfers between the host and device. Figure 6.7 shows the memory bandwidths between the structural elements of the CPU-GPU computational model. As it is clear from Figure 6.7, it is crucial to avoid frequent data transfers between the host and device to prevent performance degradation.

As in the CPU and GPGPU case, the simulation cycle in CUDA implementation starts with neighbor search step where particle proxomities are determined.

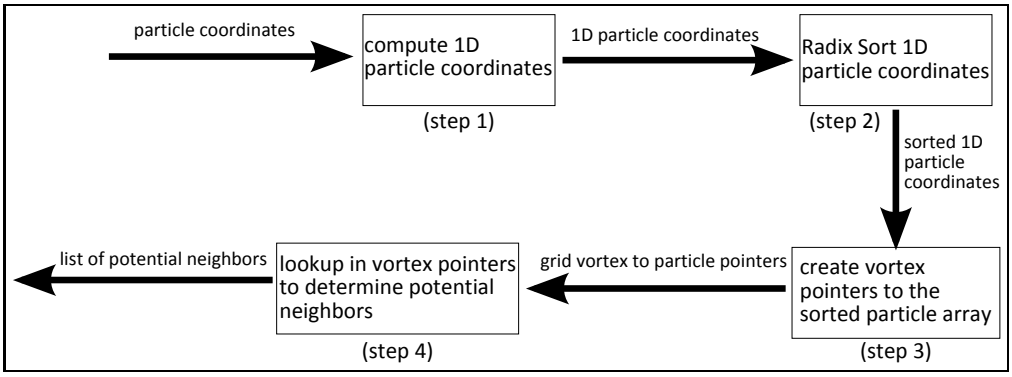


Figure 6.8: CUDA implementation of the neighbor search algorithm, which is similar to the corresponding CPU implementation. Boxes stand for kernels.

6.3.1 Neighbor Search Algorithm with CUDA

In the CUDA implementation of the particle system, we employ the neighbor search method that we detail in Chapter 5.5. We present the GPGPU implementation of the algorithm in Section 6.2.1. The CUDA implementation of the neighbor search algorithm is much simpler than the GPGPU case. This is because of the fact that CUDA kernels are able to perform arbitrary reads and writes from and into arrays residing in the global memory.

The steps of the neighbor search algorithm are underlined in Algorithm 1 of Section 5.5.1. Figure 6.8 illustrates the flowchart of the algorithm in CUDA. The boxes in Figure 6.8 depicts the kernels which are main computational elements of CUDA platform. The first kernel computes the 1D grid coordinates of each particle by applying Equation 6.1 to integral particle coordinates. The next kernel uses the radix sort algorithm [25] to sort the particles with respect to their 1D grid coordinates. The sorted particles are scanned to find the pointers for each grid voxel to the sorted particle array. The voxel pointers are used in potential neighbor lookup in fluid force-density computation kernel.

6.3.2 Density and Force Computations, and Numerical Integration in CUDA

The bottleneck of the CUDA implementation of our particle simulation is neighbor lookup (step 4 in Figure 6.8) where each thread scans the neighbor particle list to determine potential neighbors. Thus, it is crucial for the performance of the simulation to reduce number of lookups for each particle. In a naive SPH implementation, a kernel would first compute particle densities by scanning the neighbor list. Then a second kernel would use the densities to compute pressure based SPH forces again by computing the potential neighbors. Although this approach produces physically more precise results, it requires two kernels to determine potential neighbor lists for each particle. A faster way is to use a single kernel for force and density computations, in this case by using density values from the previous simulation step. In our experiments, using particle densities from the previous simulation step and combining SPH force and density computations into a single kernel improve the overall performance of the system about 30 %. The kernel computing the fluid particle densities and SPH forces uses Equations 3.13, 3.21, 3.22, and 3.28. The same kernel uses Lennard-Jones potential (Equation 5.3) for computing the boundary forces acting on the fluid particles.

As illustrated in Figure 6.4(b), the interaction between cloth and fluid particles are computed by another kernel which computes discrete element method forces (Equations 4.9 and 4.10) and forces due to capillary pressure (Equation 5.11). The same kernel adds the gravitational acceleration and environmental viscous drag to fluid and cloth particles. Updated particle position are computed by the integration kernel which uses the velocity Verlet integration schema.

Chapter 7

Results

This chapter presents several examples of particle-based simulations. The examples are presented by indicating the implementation details, such as the architecture the simulation runs on (CPU or GPU), the number of total particles (fluid, cloth, and rigid body particles), run time for each simulation step, memory footprint (main memory and graphics memory), and the details of the rendering process. The simulations are implemented with C++ programming language, Cg shading language, and CUDA for parallel computing on the GPU.

We exploit multicore architecture of CPU via OpenMP [139]. OpenMP is an API supporting shared memory multicore processors which have become the standard processor architecture for common use desktop and laptop computers. By using OpenMP, certain parts of a simulation algorithm (specifically *for* loops) can be parallelized easily [15].

The first example is from a 2D simulation where a fluid body is simulated by SPH, see Figure 7.1. Repulsive forces between the unmovable rigid body particles and fluid particles are computed by the Lennard-Jones potential as explained in Section 5.1. The simulation runs on CPU where OpenMP is employed to exploit the multicore structure of the processor. There are 12 thousand (12K) fluid particles in the scene and total number of rigid body particles is 1K. The simulation runs at 50 FPS.

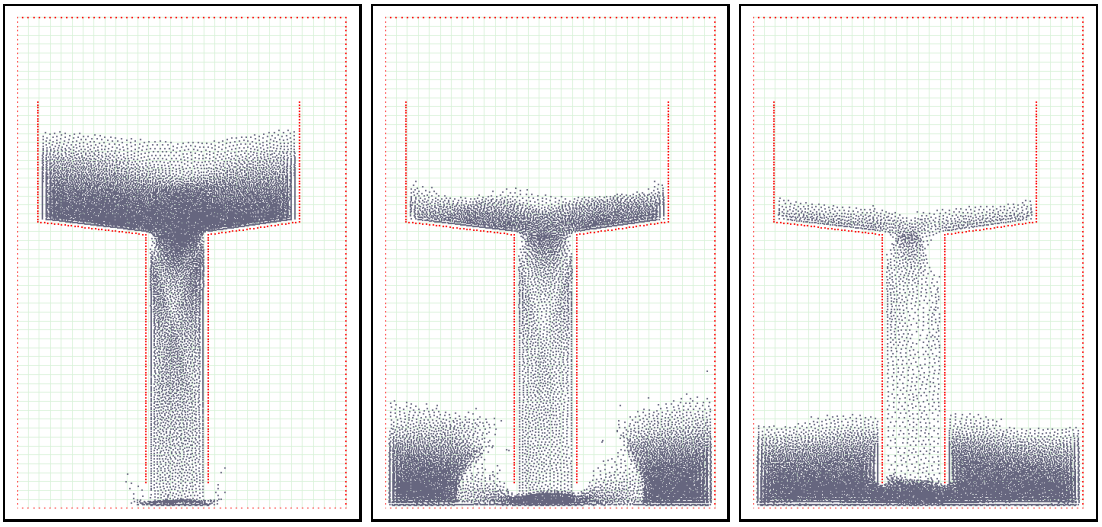


Figure 7.1: A 2D simulation where fluid particles run through a pipe-like rigid object. The simulation runs on CPU.

Figure 7.2: A cloth is draped onto a rigid object.

In Figure 7.2, a cloth mesh is draped onto a cylinder. Collision between the cloth mesh and the rigid object is handled in particle-to-particle basis. Collision response forces are computed by the discrete element method which is explained in Section 4.3. Cloth model and rigid object consists of 6K and 13K particles, respectively. The scene is rendered in Povray.

Figure 7.3 shows frames of a simulation where a river flowing through a stack of rocks blocking a valley bed. By employing the boundary conditions mentioned in Section 5.1, water flows through a complex structure realistically. The scene consists of 60K fluid particles and 220K solid boundary particles including the stone stack and the valley. The simulation runs on CPU with speed of 30 FPS excluding the rendering which is done by Povray [92].

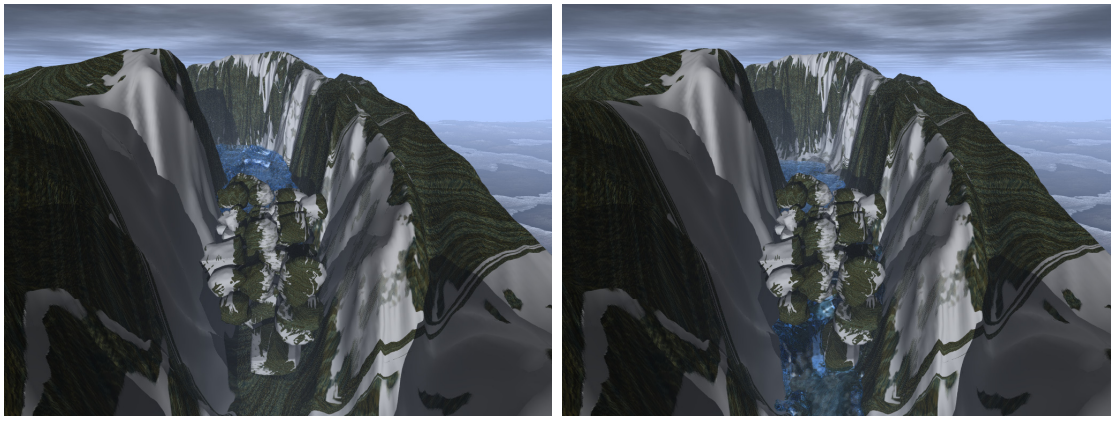


Figure 7.3: A river flowing through a stack of rocks.

The next example, which is illustrated by Figure 7.4, depicts a viscous liquid (lava) flowing down on a terrain. For this simulation, the viscosity coefficient μ of Equation 3.22 is set to 12.0. Moreover the rigid object particles exert a viscous drag on the fluid particle based on the rigid particle velocities (Equation 5.2). The lava and terrain consist of 50K and 60K particles, respectively and the simulation runs at 35 fps. Rendering is done offline with Povray.

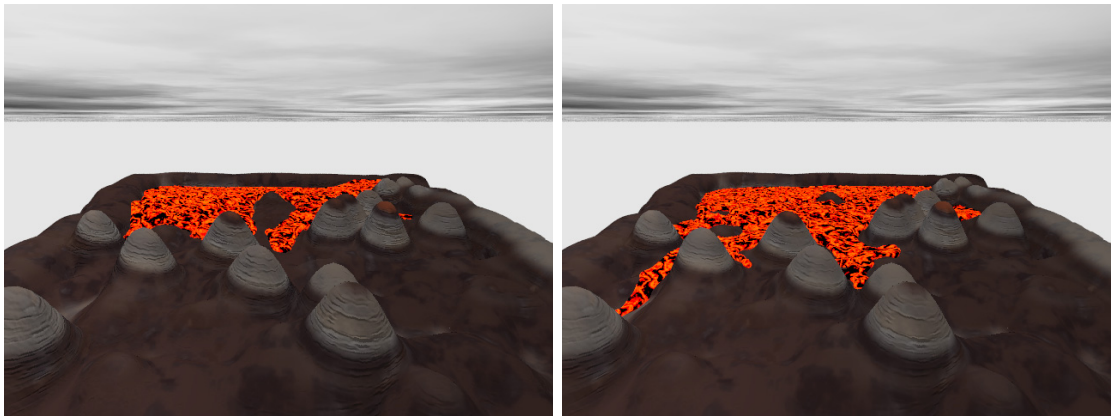


Figure 7.4: Lava flows down on a terrain.

Figure 7.5 illustrates frames from a simulation where two bodies of fluid are mixed by a rotating disk. The example demonstrates the splashing and mixing effects of two fluid bodies with different densities. The density difference of the fluids is simulated by assigning different values to the rest densities ρ_0 (Equation 3.18) The scene has a total of 50K fluid particles and 150K boundary particles. The simulation runs on CPU, at 34 fps.

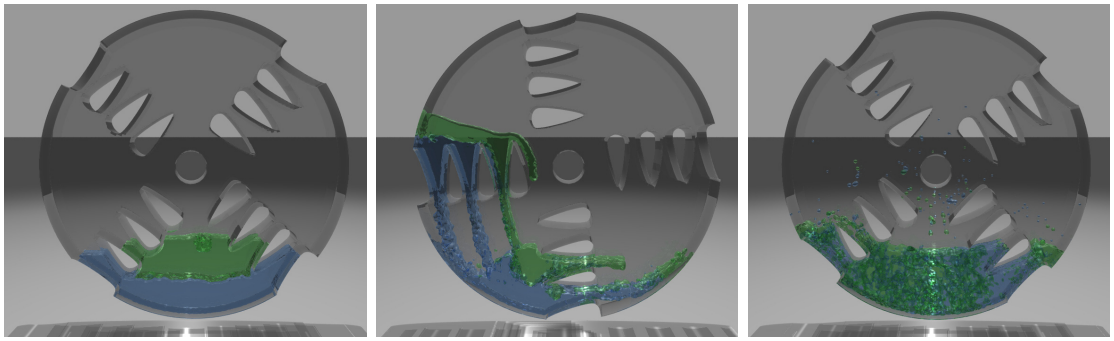


Figure 7.5: Two types of fluids with different densities are mixed by a rotating mixer.

Figure 7.6 shows still images from an animation where fluid dragons are dropped into a container. After the fluid stabilizes, it flows down through a pipe. Total number of fluid particles is 120K and object particles is 50K. Simulation is implemented on CPU and the memory footprint of the whole system is about 1.1 GB. The simulation runs at 25 fps.

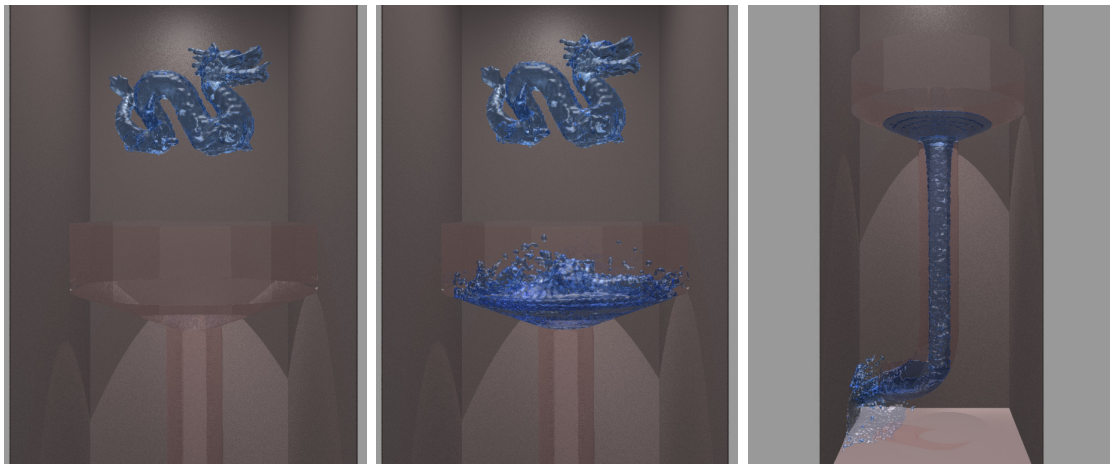


Figure 7.6: Fluid dragons dropped into a container and flow down through a pipe.

The next simulation demonstrates fluid cloth interaction where fluid particles leak through pores of a cloth, see Figure 7.7. The cloth is modeled with a mass-spring network as explained in Section 4.1, and the poured fluid passes through micropores. The simulation runs on CPU in 30 FPS.

Figure 7.8 illustrates two different simulations which are implemented by Cg on fragment shaders as described in Section 6.2. In Figure 7.8 (a) fluid particles

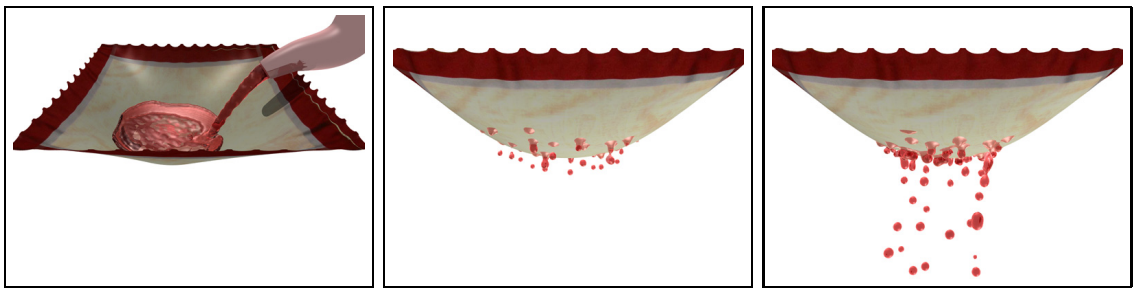
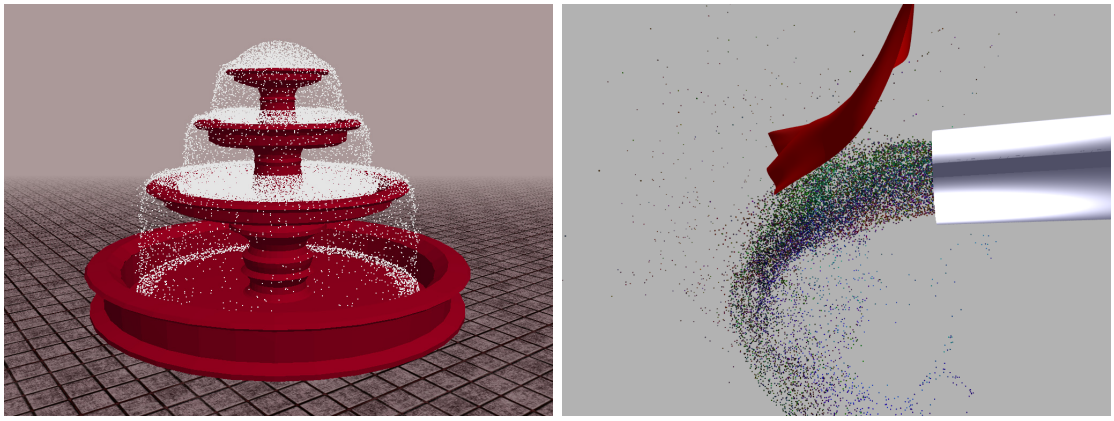


Figure 7.7: Fluid is poured onto a hanged cloth and leaks through the pores of cloth.



(a) (b)

Figure 7.8: (a) Fluid particles flow down onto a fountain (simulation speed 20 FPS). (b) Fluid particles flow into a cloth mesh simulation speed 22 FPS).

flow down onto a fountain model and fill it. There are 64K fluid particles and the fountain object consists of 64K particles. In Figure 7.8 fluid particles jet into a cloth mesh which moves because of the fluid flow. Both of these simulations run entirely on GPU with a memory footprint of approximately 700 MB. Both scenes are rendered by OpenGL. Figure 7.9 shows a series of frames from a breaking dam animation which run on GPU. The fluid body consists of 200K particles and memory footprint of the system is about 850 MB. The simulation is implemented by Cg as well and runs at 30 FPS.

Still images shown in Figure 7.10 are from a simulation where a fluid body flows down through a collection of rigid objects. The simulation is implemented in CUDA and runs at 60 FPS. The number of fluid and rigid body particles are 60K and 120K, respectively.

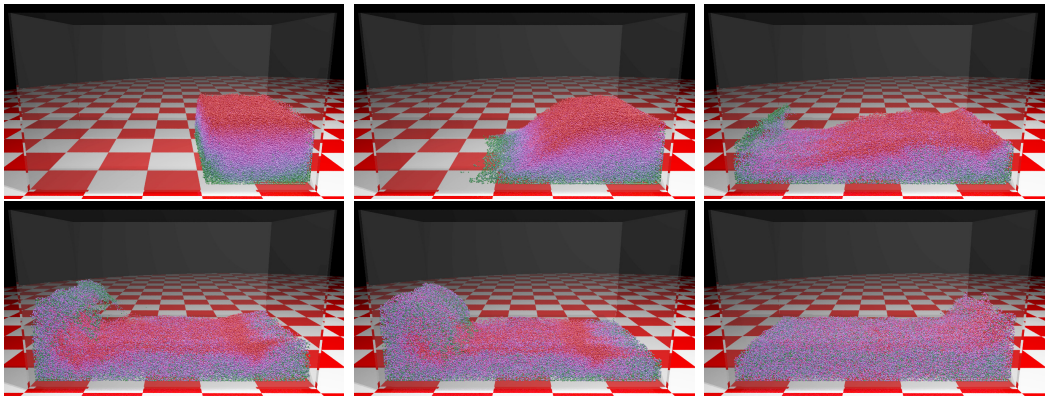


Figure 7.9: A series of frames from a breaking dam simulation. The simulation is implemented by GPGPU (Cg) and rendered by OpenGL.

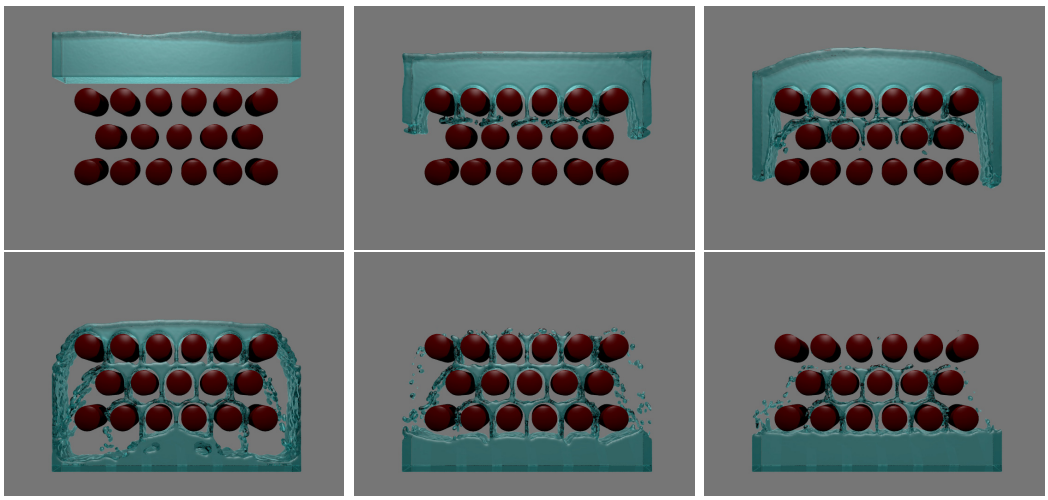


Figure 7.10: A fluid body flows down through a series of rigid objects.

Figure 7.11 shows still images from a simulation where fluid and a knitwear interact. Fluid flow jets into the hanged knitwear and pushes it. Some of the fluid gets through the knitwear and some is absorbed by it by capillary pressure forces as explained in Section 3.2.3 and Section 5.3. As it absorbs fluid, knitwear changes its color and weight. The whole system is implemented on CUDA. The rendering of fluid and knitwear is done by GLSL [122] shaders. Knitwear rendering is done as it is explained in Section 4.4.

Figure 7.12 (a) illustrates a simulation where a knitwear model is draped onto a rigid sphere. This simulation shows the volumetric structure of knitwear, detailed in Section 4.2, and its response to perpendicular forces alongside with

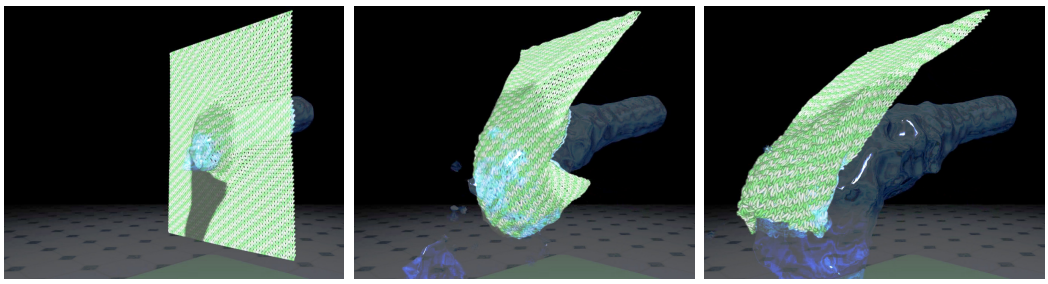


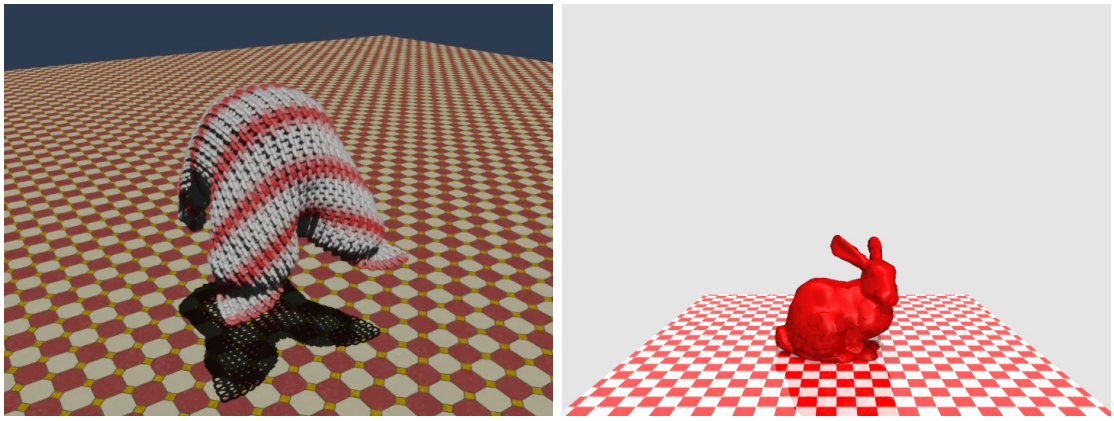
Figure 7.11: A fluid flow jets into a hanged piece of knitwear. Two way coupling is handled in particle-to-particle basis. Some of the fluid is absorbed by the knitwear because of its porous structure. Knitwear changes its appearance and weight as it gets wet.

the shear, stretch and bending forces. The scene is rendered by GLSL shaders with the method which is explained in Section 4.4. The rendering provides the knitwear a realistic fluffy look and detailed soft shadows. The spring mass part of the simulation is implemented to run on GPU with CUDA.

A deformable bunny model drops onto floor as shown in Figure 7.12 (b). The deformable model is modeled by placing spring constraints to the polygon edges of model. To prevent the model from collapsing into itself an addition set of springs is placed between each of the vertices and object centroid. Although this is not an optimal way to model a deformable object, the simulation shows that the computational power of GPU can attain interactive rates even the number of springs is very high. The bunny model has approximately 32K vertices and 64K edges. The deformable model has approximately 100K springs and simulation runs at 32 FPS.

The chart in Figure 7.14 shows the performance of a breaking dam scene with different number of fluid particles. The simulation is run with the same parameters (kernel radius, grid resolution, fluid parameters, etc.) except the number of fluid particles. The tests are performed in two different computers whose relevant technical details are given in Figure 7.13. These two computers (especially in terms of their GPU capabilities) are of different generations with Computer 2 being a high end performance computer.

Several of the technical parameters are of importance for the performance of



(a) (b)

Figure 7.12: (a) A piece of knitwear is dropped into a rigid body. (b) A deformable object falls onto floor.

our simulation method. The number of microprocessors and number of cores per microprocessor is crucial since these parameters directly relate to the number of concurrent computations. Maximum number of threads per block is also important since higher number of threads that can reside inside a single block means more threads can communicate through the shared memory and execution can switch from stalled threads to idle threads improving the occupancy of the whole system. Maximum register size is also crucial since registers are used to store temporary data (such as variable) during computations. Physically registers reside on the chip and their latency is very low. However, whenever the number of registers is insufficient, the slower shared or global memory is used for temporary storage. This of course impedes the performance.

Figure 7.14 illustrates the performance gain in terms of frames per second as a more powerful hardware is used. The difference in the performance of two tested graphics boards suggests that the performance of the simulations method is very responsive to improvement of hardware. Our CUDA implementation can provide interactive rates for 200K particles. One important statistics to show the amount of computation is the number of particle-to-particle interactions, that is number of particles close enough to affect each other. Although this number changes according to the parameters of the scene, in our tests there are approximately 6 million particle interactions for 240K fluid particles.

		Computer 1	Computer 2
Host	CPU	DualCore Intel Core i5	QuadCore Intel Core i7
	CPU Clock	2400 MHz	2666 MHz
	Memory Size	4 GB.	12 GB.
	Memory Type	DDR3	DDR3
	Memory Bus Clock	1079 MHz	
Device	GPU Name	nVIDIA GeForce GT 330M	nVIDIA GeForce GTX 480
	Clock Rate	1265 MHz	810 MHz
	Multiprocessors / Cores	6 / 48	15 / 120
	Number of Transistors	486 million	3200 million
	Bus Type	PCI Express 2.0 x16 @ x16	PCI Express 2.0 x16 @ x16
	Memory Bus Type	DDR3	DDR3
	Max Threads Per Block	512	1024
	Max Registers Per Block	16384	32768

Figure 7.13: Several parameters of the computers used in performance tests.

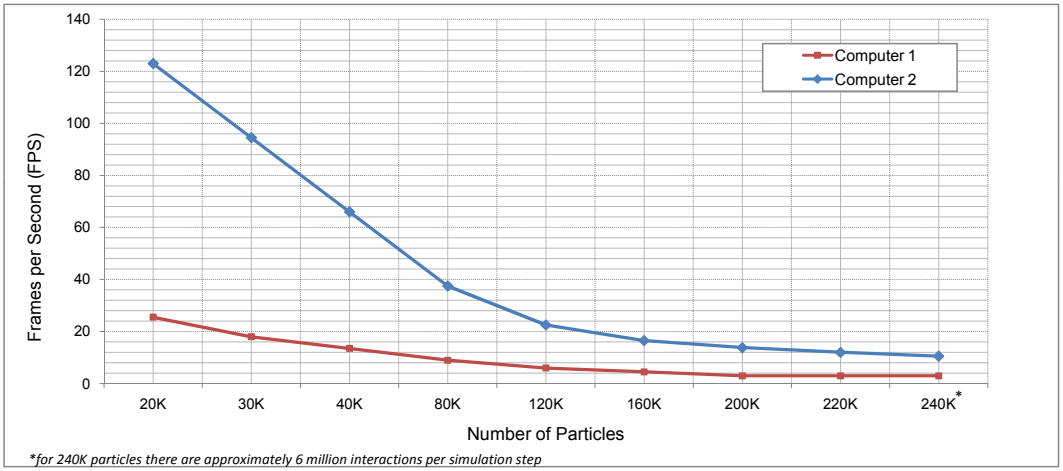


Figure 7.14: The frame rates of the breaking dam simulation for different number of particles.

Chapter 8

Conclusion

This Thesis propose several new methods for modeling and simulation of natural phenomena. We employ the physically-based simulation paradigm which provides great flexibility, control, and realism. Specifically, we investigate particle-based modeling and simulation methods with the aim of mimicking natural occurrences that is fluids, cloth-like deformable objects and their behaviors. Our priority during our research has been practicality, realism, and computational efficiency of the implemented methods.

Determining the set of neighboring particles at each simulation step is a vital stage of a particle system. We develop a sorting-based space subdivision method for the purpose of achieving fast, accurate, and robust neighbor detection. Our neighbor search method can be used with any particle simulation paradigm (e.g. Smoothed Particle Hydrodynamics, the Discrete Elements Method) since it does not depend on any assumptions of the nature of the simulation algorithm or simulation parameters. We use the Marching Cubes algorithm to tessellate the free fluid surface. To improve the quality of the final rendering, we propose a method where we differentiate the particles according to their relative position to the surface. The result is a fluid surface which is smoother in flat regions and gives better details in regions like water fronts.

In the proposed particle system, objects interact with each other through

interparticle forces. There are several alternatives for defining these forces and in our system, we use a combination of these alternatives. We simulate fluid-fluid, cloth-fluid, boundary-fluid interactions with interparticle forces.

We animate cloth-like deformable objects and knitwear by employing mass-spring method. We propose improvements upon the mass-spring methods so that knitwear which has a prominent thickness and complex microstructue can be simulated and visualized. We develop a hardware-based method to render knitwear realistically. We propose a method for simulation of absorption of fluids by cloth-like object where capillary pressure-based forces are employed.

One of the intrinsic characteristics of particle systems is that they are parallelizable. Even though particles interact with each other, they can be handled independently within a simulation step. Thus particle systems are very good candidates to be implemented on parallel computing systems that have become accessible even to low end computer systems. We implement the proposed particle system on multicore CPUs and programmable graphics processors (GPUs) to exploit their computational power. The details of these implementations are presented in the preceding chapters.

There are several possible extensions for improving the applicability, performance, and realism of the proposed particle system. A hardware accelerated rendering method can be incorporated into the system for a high quality, state of art visualization. GPU implemented ray-casting algorithms are very good candidates for this purpose. Isosurface extraction methods fail to capture details, such as, in mixing fluids. Rigid, semi-rigid, granular objects can be added to our particle system by modeling them as particle collection.

Bibliography

[1] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas. Adaptively sampled particle fluids. *ACM Transactions on Graphics*, 26(3): Article no. 48, 8 pages, 2007.

[2] T. Agui, Y. Nagao, and M. Nakajma. An expression method of cylindrical cloth objects. *Transactions of Society of Electronics, Information and Communications*, 7:1095–1097, 1990.

[3] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, New York, NY, USA, 1987.

[4] D. Baraff and A. Witkin. Large steps in cloth simulation. In *ACM Computer Graphics (Proceedings of SIGGRAPH'98)*, pages 43–54, 1998.

[5] D. Baraff, A. Witkin, and M. Kass. Untangling cloth. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'03)*, 22(3):862–870, 2003.

[6] M. Becker and M. Teschner. Weakly compressible SPH for free surface flows. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'07)*, pages 209–217, Aire-la-Ville, Switzerland, 2007.

[7] M. Becker, H. Tessendorf, and M. Teschner. Direct forcing for lagrangian rigid-fluid coupling. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):493–503, 2009.

[8] N. Bell, Y. Yu, and P. J. Mucha. Particle-based simulation of granular materials. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on*

Computer Animation (SCA'05), pages 77–86, New York, NY, USA, 2005. ACM.

[9] E. Boxerman and U. Ascher. Decomposing cloth. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'04)*, pages 153–161, 2004.

[10] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'02)*, 21(3):594–603, 2002.

[11] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'03)*, pages 28–36, San Diego, CA, 2003.

[12] T. Brochu, C. Batty, and R. Bridson. Matching fluid simulation elements to surface geometry and topology. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 47, 9 pages, 2010.

[13] M. Carlson, P. J. Mucha, I. R. Brooks Van Horn, and G. Turk. Melting and flowing. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '02)*, pages 167–174, New York, NY, USA, 2002. ACM Press.

[14] M. Carlson, P. J. Mucha, and G. Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'04)*, 23(3):377–384, 2004.

[15] B. Chapman, G. Jost, and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[16] J. Chen and N. Lobo. Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equations. *Graphical Models and Image Processing*, 57(2):107–116, 1995.

[17] J. X. Chen, N. da Vitoria Lobo, C. E. Hughes, and J. M. Moshell. Real-time fluid simulation in a dynamic virtual environment. *IEEE Computer Graphics and Applications*, 17(3):52–61, 1997.

[18] Y. Chen, S. Lin, H. Zhong, Y. Xu, B. Guo, and H. Shum. Realistic rendering and animation of knitwear. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):43–55, 2003.

[19] Z. Chen, G. Huan, and Y. Ma. *Computational Methods for Multiphase Flows in Porous Media*. SIAM Computational Science & Engineering, 2006.

[20] A. Cheng and D. Cheng. Heritage and early history of the boundary element method. *Engineering Analysis with Boundary Elements*, 29(3):268–302, 2005.

[21] E. V. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical report, CERN, 1995.

[22] K.-J. Choi and H.-S. Ko. Stable but responsive cloth. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'02)*, 21(3):604–611, 2002.

[23] S. Clavet, P. Beaudoin, and P. Poulin. Particle-based viscoelastic fluid simulation. In *Proceedings of the Symposium on Computer Animation (CA'05)*, pages 219–228, 2005.

[24] P. W. Cleary, S. H. Pyo, M. Prakash, and B. K. Koo. Bubbling and frothing liquids. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'07)*, 23(3): Article no. 97, 6 pages, 2007.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[26] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100:32–74, 1928.

[27] E. de Aguiar, L. Sigal, A. Treuille, and J. K. Hodgins. Stable spaces for real-time clothing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 106, 9 pages, 2010.

[28] M. Desbrun and M. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. In R. Boulic and G. Hegron, editors, *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, pages 61–76. Springer-Verlag, Aug 1996.

[29] M. Desbrun, P. Schroder, and A. H. Barr. Interactive animation of structured deformable objects. *Proceedings of Graphics Interface (GI'99)*, pages 1–8, 1999.

[30] R. Dimitrov. Cascaded shadow maps. Nvidia White Paper, 2007. Last Accessed on 28th of January, 2010.

[31] F. Durupinar and U. Gudukbay. Procedural visualization of knitwear and woven cloth. *Computers & Graphics*, 31(5):778–783, 2007.

[32] B. Eberhardt, A. Weber, and W. Strasser. A fast, flexible, particle-system model for cloth draping. *IEEE Computer Graphics and Applications*, 16(5):52–59, 1996.

[33] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *ACM Transactions on Graphics*, volume 21,3, pages 736–744, 2002.

[34] B. Feldman, J. O'Brien, B. Klingner, and T. Goktekin. Fluids in deforming meshes. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 255–259, 2005.

[35] W. Feng, Y. Yu, and B. Kim. A deformation transformer for real-time cloth animation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 108, 9 pages, 2010.

[36] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[37] C. Feynman. Modeling the appearance of cloth. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1986.

[38] C. Fletcher. *Computational Techniques for Fluid Dynamics*. Springer-Verlag, 1990.

[39] N. Foster and R. Fedkiw. Practical animation of liquids. In *ACM Computer Graphics (Proceedings of SIGGRAPH'01)*, pages 23–30, 2001.

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

[41] N. Foster and D. Metaxas. Modeling the motion of a hot, turbulent gas. In *ACM Computer Graphics (Proceedings of SIGGRAPH'97)*, pages 181–188, New York, NY, USA, 1997.

[42] A. Fournier and W. T. Reeves. A simple model of ocean waves. *ACM Computer Graphics (Proc. of SIGGRAPH)*, 20(4):75–84, 1986.

[43] R. Fox and A. McDonald. *Introduction to Fluid Mechanics*. John Wiley and Sons, 1998.

[44] M. N. Gamito, P. F. Lopes, and M. R. Gomes. Two-dimensional simulation of gaseous phenomena using vortex particles. *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 3–15, 1995.

[45] F. J. Gerstner. Theorie der wellen. *Abh. d. k. bohm. Ges. d. Wiss. Also reprinted in Ann. der Physik*, 32:412–440, 1809.

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

[47] T. Goktekin, A. Bargteil, and J. O'Brien. A method for animating viscoelastic fluids. *ACM Transactions on Graphics*, 23(3):463–468, 2004.

[48] R. Goldenthal, D. Harmon1, R. Fattal, M. Bercovier, and E. Grinspun. Efficient simulation of inextensible cloth. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'07)*, 26(3): Article no. 49, 8 pages, 2007.

[49] E. Guendelman, A. Selle, F. Losasso, and R. Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics*, 24(3):973–981, 2005.

[50] S. Hadap and N. Magnenat-Thalmann. Modeling dynamic hair as a continuum. *Computer Graphics Forum*, 20(3):329–338, 2001.

[51] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Proceedings of Computer Graphics International*, pages 63–70, 2007.

[52] F. Harlow and J. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8:2182–2189, 1965.

[53] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'03)*, pages 92–101, 2003.

[54] J. He, X. Chen, Z. Wang, C. Cao, H. Yan, and Q. Peng. Real-time adaptive fluid simulation with complex boundaries. *Visual Computer*, 26(4):243–252, 2010.

[55] K. Hegeman, N. A. Carr, and G. S. P. Miller. Particle-based fluid simulation on the GPU. In V. N. Alexandrov, D. G. van Albada, Peter, and J. Dongarra, editors, *Proceedings of the International Conference on Computational Science, Part IV*, volume 3994 of *Lecture Notes in Computer Science*, pages 228–235, 2006.

[56] R. Hoetzlein and T. Höllerer. Interactive water streams with sphere scan conversion. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'09)*, pages 107–114, 2009.

[57] J.-M. Hong, H.-Y. Lee, J.-C. Yoon, and C.-H. Kim. Bubbles alive. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'08)*, 27(3): Article no. 48, 4 pages, 2008.

[58] K. Iwasaki, Y. Dobashi, F. Yoshimoto, and T. Nishita. GPU-based rendering of point-sampled water surfaces. *Visual Computer*, 24(2):77–84, 2008.

[59] T. Jakobsen. Advanced character physics. In *Proceedings of the Game Developer's Conference*, San Jose, CA, 2001.

[60] J.F. Wendt (ed.). *Computational Fluid Dynamics*. Springer-Verlag, 1991.

[61] J. Kaldor, D. James, and S. Marschner. Efficient yarn-based cloth with adaptive contact linearization. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 105, 10 pages, 2010.

[62] J. M. Kaldor, D. L. James, and S. Marschner. Simulating knitted cloth at the yarn level. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'08)*, 27(3): Article no. 65, 10 pages, 2008.

[63] N. Kang, J. Park, J. Noh, and S. Y. Shin. A hybrid approach to multiple fluid simulation using volume fractions. *Computer Graphics Forum*, 29(2):685–694, 2010.

[64] Y.-M. Kang, J.-H. Choi, H.-G. Cho, and D.-H. Lee. An efficient animation of wrinkled cloth with approximate implicit integration. *The Visual Computer*, 17:147–157, 2001.

[65] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. In *ACM Computer Graphics (Proceedings of SIGGRAPH'90)*, pages 49–57, 1990.

[66] Z. Kačić-Alesić, M. Nordenstam, and D. Bullock. A practical dynamics system. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'03)*, pages 7–16, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[67] H. Kawashima, F. Kuwahara, and A. Nakayama. Similarity solutions for pressure-driven boiling flows in capillary porous media. *International Communications in Heat and Mass Transfer*, 26(3):319 – 327, 1999.

[68] J. Kim, D. Cha, B. Chang, B. Koo, and I. Ihm. Practical animation of turbulent splashing water. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'06)*, pages 335–344, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[69] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based Particle Engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*, pages 115–122, New York, NY, USA, 2004. ACM.

[70] P. Kipfer and R. Westermann. Realistic and interactive simulation of rivers. In *Proceedings of Graphics Interface (GI'06)*, pages 41–48, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[71] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*, pages 123–131, New York, NY, USA, 2004. ACM.

[72] P. Krištof, B. Beneš, J. Křivánek, and O. Šťava. Hydraulic erosion using smoothed particle hydrodynamics. *Computer Graphics Forum (Proceedings of Eurographics'09)*, 28(2):219228, mar 2009.

[73] J. Kruger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3D flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[74] T. Kunii and H. Gotoda. Singularity theoretical modeling and animation of garment wrinkle formation process. *The Visual Computer*, 6(6):326–336, 1990.

[75] W. Laan, S. Green, and M. Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'09)*, pages 91–98, New York, NY, USA, 2009. ACM.

[76] B. Lafleur, N. M. Thalmann, and D. Thalmann. *Cloth Animating with Self-Collision Detection in Modeling in Computer Graphics.* Springer-Verlag, Berlin, 1991.

[77] L. Latta. Building a million particle system. In *Proceedings of the Game Developers Conference*, 2004.

[78] H.-Y. Lee, J.-M. Hong, and C.-H. Kim. Interchangeable sph and level set method in multiphase fluids. *The Visual Computer*, 25(5-7):713–718, 2009.

[79] T. Lenaerts, B. Adams, and P. Dutré. Porous flow in particle-based fluid simulations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'08)*, 27(3): Article no. 49, 8 pages, 2008.

[80] T. Lenaerts and P. Dutré. Mixing fluids and granular materials. *Computer Graphics Forum*, 28(2):213–218, 2009.

[81] M. Lentine, W. Zheng, and R. Fedkiw. A novel algorithm for incompressible flow using only a coarse grid projection. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 114, 9 pages, 2010.

[82] M. C. Leverett. Capillary behavior in porous solids. *The American Institute of Mining, Metallurgical, and Petroleum Engineers*, 142:152–169, 1941.

[83] B. Long and E. Reinhard. Real-time fluid simulation using discrete sine/cosine transforms. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'09)*, pages 99–106, 2009.

[84] A. Lopes and K. Brodlie. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):16–29, 2003.

[85] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM Computer Graphics (Proceedings of SIGGRAPH'87)*, pages 163–169, 1987.

[86] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'04)*, 23(3):457–462, 2004.

[87] F. Losasso, G. Irving, E. Guendelman, and R. Fedkiw. Melting and burning solids into liquids and gases. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):343–352, 2006.

[88] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw. Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[89] N. Max. Vectorized procedural models for natural terrain: Waves and islands in the sunset. *ACM Computer Graphics (Proc. of SIGGRAPH'81)*, 15(3):317–324, 1981.

[90] GPGPU.org. General-Purpose computation on Graphics Processing Units, available at `http://gpgpu.org/`.

[91] NVIDIA. the Cg homepage, available at `http://developer.download.nvidia.com`.

[92] The Persistence of Vision Raytracer Pty. Ltd. POV-Ray: The Persistence of Vision Raytracer, available at `http://www.povray.org/`.

[93] M. Meissner and B. Eberhardt. The art of knitted fabrics, realistic physically based modelling of knitted patterns. *Computer Graphics Forum (Proceedings of Eurographics94*, 17(3):355–362, 1998.

[94] S. Melax. Dynamic plane shifting BSP traversal. In *Proceedings of Graphics Interface (GI'00)*, pages 213–220, 2000.

[95] G. Miller and A. Pearce. A connected particle system for animating viscous fluids. *Computers & Graphics*, 13(3):305–309, 1989.

[96] G. Miller and A. Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers & Graphics*, 13(3):305–309, 1989.

[97] J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543–574, 1992.

[98] J. Monaghan. Simulating free surface flows with SPH. *Journal of Computational Physics*, 110(2):399–406, 1994.

[99] J. P. Morris. Simulating surface tension with smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 33:333–353, June 2000.

[100] J. P. Morris, P. J. Fox, and Y. Zhu. Modeling low Reynolds number incompressible flows using SPH. *Journal of Computational Physics*, 136(1):214–226, 1997.

[101] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'03)*, pages 154–159, 2003.

[102] M. Müller, S. Schirm, and S. Duthaler. Screen space meshes. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'07)*, pages 9–15, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[103] M. Müller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of fluids with deformable solids. *Journal Computer Animation and Virtual Worlds*, 15(3–4):159–171, July 2004.

[104] M. Müller, B. Solenthaler, R. Keiser, and M. Gross. Particle-based fluid-fluid interaction. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'05)*, pages 237–244, 2005.

[105] H. Ng and R. Grimsdale. A geometrical editor for fold information. *Lecture Notes in Computer Science*, 1024:124–131, 1995.

[106] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[107] G. M. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceedings of the 2nd IEEE Conference on Visualization (Vis'91)*, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[108] O. Nocent, J. Nourrit, and Y. Remion. Towards mechanical level of detail for knitwear simulation. In *Proceedings of WSCG'01 Conference*, pages 252–259, 2001.

[109] NVIDIA Corporation. NVIDIA CUDA Programming Guide 3.0, available at `http://developer.download.nvidia.com`, 2008.

[110] J. O'Brien and J. Hodgins. Dynamic simulation of splashing fluids. *Proceedings of Computer Animation*, pages 198–205, 1995.

[111] S. Oh, J. Ahn, and K. Wohn. Low damped cloth simulation. *Visual Computer*, 22(2):70–79, 2006.

[112] S. Oh, J. Noh, and K. Wohn. A physically faithful multigrid method for fast cloth simulation. *The Journal of Computer Animation and Virtual Worlds*, 19(3-4):479–492, 2008.

[113] K. Perlin. An image synthesizer. *ACM Computer Graphics (Proc. of SIGGRAPH'85)*, 19(3):287–296, 1985.

[114] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[115] S. Premoze, T. Tasdizen, J. Bigler, A. E. Lefohn, and R. T. Whitaker. Particle-based simulation of fluids. *Eurographics 2003*, 22(3):401–410, 2003.

[116] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1997.

[117] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. *Proceedings of Graphics Interface (GI'95)*, pages 147–154, 1995.

[118] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[119] W. J. W. Rantkine. On the exact form of waves near the surfaces of deep water. *Philosophical Transactions of the Royal Society*, A 153:127–138, 1863.

[120] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

[121] A. Robinson-Mosher, T. Shinar, J. Gretarsson, J. Su, and R. Fedkiw. Two-way coupling of fluids to rigid and deformable solids and shells. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'08)*, 27(3): Article no. 49, 8 pages, 2008.

[122] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 2009.

[123] Y. Sakaguchi, M. Minoh, and K. Ikeda. Party: Physical environment of artificial reality for dress simulation. *Transactions of Society of Electronics, Information and Communications*, pages 25–32, 1991.

[124] H. Scharsach. Advanced GPU Raycasting. In *In Proceedings of Central European Seminar on Computer Graphics for Students (CESCG'05)*, pages 69–76, 2005.

[125] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible sph. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'09)*, 28(3): Article no. 40, 6 pages, 2009.

[126] B. Solenthaler, J. Schläfli, and R. Pajarola. A unified particle model for fluid-solid interactions: Research articles. *Computer Animation and Virtual Worlds*, 18(1):69–82, 2007.

[127] O.-Y. Song, D. Kim, and H.-S. Ko. Derivative particles for simulating detailed movements of fluids. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):711–719, 2007.

[128] J. Stam. Stable fluids. In *ACM Computer Graphics (Proceedings of SIGGRAPH'99)*, pages 121–128, Los Angeles, 1999. Addison Wesley Longman.

[129] K. Steele, D. Cline, P. K. Egbert, and J. Dinerstein. Modeling and rendering viscous liquids: Research articles. *The Journal of Computer Animation and Virtual Worlds*, 15(3-4):183–192, 2004.

[130] D. Stora, P. O. Agliati, M. P. Cani, F. N., and J. D. Gascuel. Animating lava flows. In *Proceedings of the Conference on Graphics Interface (GI'99)*, pages 203–210. Morgan Kaufmann Publishers, Inc., 1999.

[131] W. Swope, H. Andersen, P. Berenc, and K. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *Journal of Chemical Physics*, 76(1):637–649, 1982.

[132] A. M. Tartakovsky and P. Meakin. A smoothed particle hydrodynamics model for miscible flow in three-dimensional fractures and the two-dimensional rayleigh-taylor instability. *Journal of Computational Physics*, 207(2):610–624, 2005.

[133] N. Tatarchuk, J. Shopf, and C. DeCoro. Real-time isosurface extraction using the GPU programmable geometry pipeline. In *ACM SIGGRAPH 2007 Courses, No. 28*, pages 122–137, New York, NY, USA, 2007. ACM.

[134] D. Terzopoulos and K. Fleischer. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. *ACM Computer Graphics (Proceedings of SIGGRAPH'88)*, 22:269–278, 1988.

[135] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. *ACM Computer Graphics (Proceedings of SIGGRAPH'87)*, 21(4):205–214, 1987.

[136] D. Terzopoulos, J. Platt, and K. Fleischer. Heating and melting deformable models. *The Journal of Visualization and Computer Animation*, 2(2):68–73, 1991.

[137] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. H. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling and Visualization (VMV)*, pages 47–54, 2003.

[138] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and

W. Strasser. Collision detection for deformable objects. In *Eurographics State-of-the-Art Report (EG-STAR)*, pages 119–139. Eurographics Association, 2004.

[139] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming, available at `http://www.openmp.org/`.

[140] N. Thürey, R. Keiser, M. Pauly, and U. Rüde. Detail-preserving fluid control. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'06)*, pages 7–12, 2006.

[141] N. Thürey, F. Sadlo, S. Schirm, M. M. Fischer, and M. Gross. Real-time simulations of bubbles and foam within a shallow water framework. *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'07)*, pages 191–198, 2007.

[142] N. Thürey, C. Wojtan, M. Gross, and G. Turk. A multiscale approach to mesh-based surface tension flows. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 48, 10 pages, 2010.

[143] D. Tonnesen. Modeling liquids and solids using thermal particles. In *Proceedings of Graphics Interface*, pages 255–262, 1991.

[144] J. S. Venetillo and W. Celes. GPU-based particle simulation with inter-collisions. *Visual Computer*, 23(9):851–860, 2007.

[145] L. Verlet. Computer experiments on classical fluids: thermodynamical properties of Lennard-Jones molecules. *Physics Reviews*, 159:98–103, 1967.

[146] P. Volino, M. Courchesne, and N. M. Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. *ACM Computer Graphics (Proceedings of ACM SIGGRAPH'94)*, 28:137–144, 1994.

[147] P. Volino and N. Magnenat-Thalmann. Implicit midpoint integration and adaptive damping for efficient cloth simulation: Collision detection and deformable objects. *The Journal of Computer Animation and Virtual Worlds*, 16(3-4):163–175, 2005.

[148] P. Volino, N. Magnenat-Thalmann, and F. Faure. A simple approach to nonlinear tensile stiffness for accurate cloth simulation. *ACM Transactions on Graphics*, 28(4): Article no. 105, 16 pages, 2009.

[149] P. Volino and N. M. Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum*, 13:155–166, 1994.

[150] H. Wang, F. Hecht, R. Ramamoorthi, and J. O'Brien. Example-based wrinkle synthesis for clothing animation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 107, 8 pages, 2010.

[151] J. Weil. The synthesis of cloth objects. *ACM Computer Graphics (Proceedings of SIGGRAPH'86)*, 20:49–54, 1986.

[152] C. Wojtan, N. Thürey, M. Gross, and G. Turk. Physics-inspired topology changes for thin fluid features. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'10)*, 29(4): Article no. 50, 8 pages, 2010.

[153] C. Wojtan and G. Turk. Fast viscoelastic behavior with thin features. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'08)*, 27(3): Article no. 47, 8 pages, 2008.

[154] Y. Xu, Y. Chen, S. Lin, H. Zhong, E. Wu, B. Guo, and H. Shum. Photorealistic rendering of knitwear using the lumislice. In *Proceedings of ACM SIGGRAPH'01*, pages 391–398, 2001.

[155] Y. Yang and N. M. Thalmann. An improved algorithm for collision detection in cloth animation with human body. *Proceedings of Pacific Graphics (PG'93)*, pages 237–251, 1993.

[156] D. Young, B. Munson, and T. Okiishi. *A Brief Introduction to Fluid Mechanics*. John Wiley and Sons, Inc., 2000.

[157] Y. Zhu and P. J. Fox. Smoothed particle hydrodynamics model for diffusion through porous media. *Transport Porous Media*, 43(3):441–471, 2001.

[158] Y. Zhu and P. J. Fox. Simulation of pore-scale dispersion in periodic porous media using smoothed particle hydrodynamics. *Journal of Computational Physics*, 182(2):622–645, 2002.