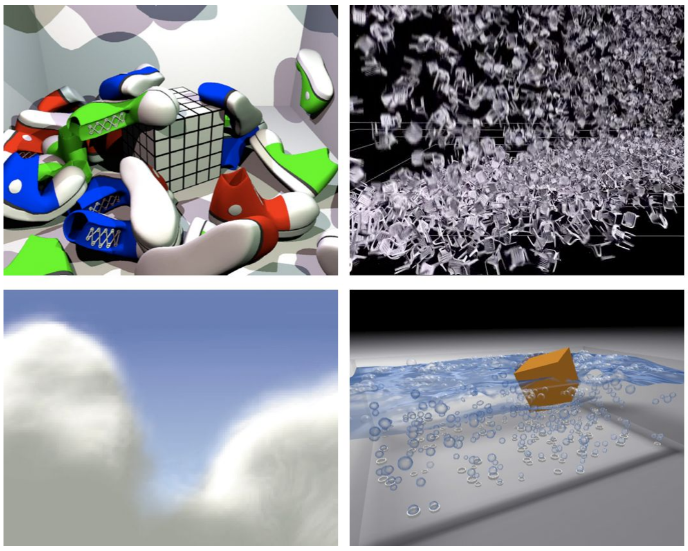
Real Time Physics
Class Notes

Matthias Müller, NVIDIA
Jos Stam, Autodesk
Doug James, Cornell University
Nils Thürey, ETH Zurich



Contents

Chapter 1

Introduction

Physically based simulation is a significant and active research field in computer graphics. It has emerged in the late eighties out of the need to make animations more physically plausible and to free the animator from explicitly specifying the motion of complex passively moving objects. In the early days, quite simple approaches were used to model physical behavior such as mass-spring networks or particle systems. Later, more and more sophisticated models borrowed from computational sciences were adopted. The computational sciences appeared decades before computer graphics with the goal of replacing real world experiments with simulations on computers. In contrast, the aim of using physical simulations in graphics was, and still is, the reproduction of the visual properties of physical processes for special effects in commercials and movies. Computer generated special effects have replaced earlier methods such as stop-motion frame-by-frame animation and offer almost unlimited possibilities.

Meanwhile, the rapid growth of the computational power of CPUs and GPUs in recent years has enabled real time simulation of physical effects. This possibility has opened the door to an entirely new world, a world in which the user can *interact* with the virtual physical environment. Real time physical simulations have become one of the main next-generation features of computer games. Allowing user interaction with physical simulations also poses new challenging research problems. In this class we address such problems and present basic as well as state-of-the-art methods for physically based simulation in real time.

1.1 Real-time vs. Off-line Physics

In off-line physical simulation, the top concern is visual quality. Computational efficiency is important because the simulations are typically done in high resolution using fine grids and large numbers of objects but performance is clearly not as important as the quality of the output. Often, farms of computers work for hours to produce a short sequence of a movie. Because off-line simulations are predictable, it is possible to re-run the process, adapt the time step in case of numerical instabilities or change parameters if the outcome does not meet the specifications or expectations.

In contrast, interactive systems run at a fixed frame rate, typically between 30 and 60 Hertz to avoid jerking and to guarantee a smooth and experience. This leaves between 15 and 30 milliseconds per time step for physical simulations. In games, the main part of this time budget is used for the core features, e.g. the graphics or artificial intelligence computations. Thus, only a few milliseconds remain for physics. Staying within this limit is a must. The resolution and visual quality have to be adjusted to meet that constraint. Because of these specific requirements, methods used in off-line simulations cannot be used one to one in real time applications. Simply reducing the resolution often yields blobby results and removes all the interesting detail. In computer games and in interactive environments it is also absolutely essential that simulations are unconditionally stable, i.e. stable under all circumstances. In contrast to off-line simulations, the outcome of interactive scenarios is non-predictable. In addition non-physical scenarios are common such as kinematic actors which move over long distances in a single time step, interpenetrating shapes or huge force fields acting on the objects in the scene.

Out of the need for specialized solutions for interactive environments emerged the new research field of real-time physically-based simulations in computer graphics. The lecturers of this class have made central contributions in this field. In this class, each lecturer presents core ideas and state-of-the-art methods in his own field.

1.2 Biographies of Authors in Alphabetical Order

- **Doug James** holds three degrees in applied mathematics, including a Ph.D. from the University of British Columbia in 2001. In 2002 he joined the School of Computer Science at Carnegie Mellon University as an Assistant Professor, then in 2006 he became an Associate Professor of Computer Science at Cornell University. His research interests are physically based animation, computational geometry, scientific computing, reduced-order modeling, and multi-sensory digital physics (including physics-based sound and haptic force-feedback rendering). He is a National Science Foundation CAREER awardee, and a fellow of the Alfred P. Sloan Foundation.

- **Matthias Müller** received his Ph.D. on atomistic simulation of dense polymer systems in 1999 from ETH Zurich. During his post-doc with the MIT Computer Graphics Group 1999-2001 he changed fields to macroscopic physically-based simulations. He has published papers on particle-based water simulation and visualization, Finite Element-based soft bodies, cloth and fracture simulation. In 2002 he co-founded the game physics middleware company NovodeX. NovodeX was acquired in 2004 by AGEIA where he was head of research and responsible for the extension of the physics simulation library PhysX by innovative new features. He has been head of the PhysX research team of nVidia since the acquisition in 2008.

- **Jos Stam** received dual BSc degrees in Computer Science and Mathematics from the University of Geneva. He got a MSc and Phd in Computer Science from the University of Toronto. After a postdoc in France and Finland he joined Alias—wavefront (now Autodesk). Stam's research spans several areas of computer graphics: natural phenomena, physics-based simulation, rendering and surface modeling. He has published papers in all of these. He has also participated in seven SIGGRAPH courses in these areas. He received the SIGGRAPH Technical Achievement Award and has two Academy Awards for his contributions to the film industry.

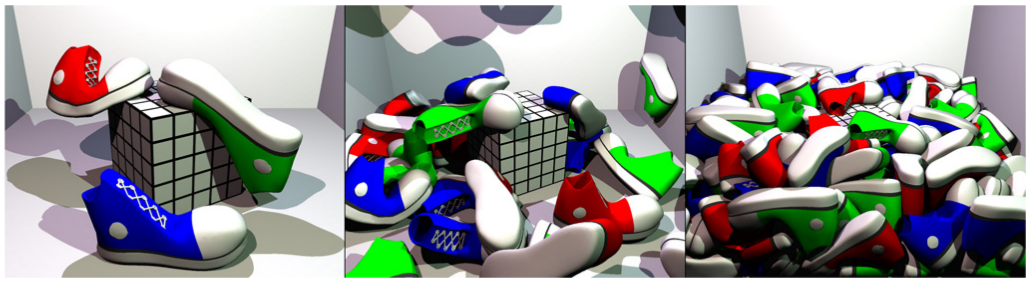
- **Nils Thuerey** is a post-doctoral researcher at the Computer Graphics Laboratory at ETH Zurich, working together with Prof. Markus Gross. In March 2007 he received his Phd (summa cum laude) in computer science from the University of Erlangen-Nuremberg. He has worked in the field of high-performance fluid simulations with free surfaces using, among others, lattice Boltzmannn methods. In addition, he has worked on the topic of real-time fluid simulations using height-field approaches, such as the shallow water equations, together with the research group of AGEIA.

1.3 Structure of the Class Notes

The notes are divided into three parts. In the first part various real time methods for the simulation of solid rigid and deformable objects are presented. In addition, multiple ways to represent solids in simulations are discussed, such as particle systems, finite element meshes or rigid bodies. The subject of the second part are fluids, i.e. liquids and gases. Simulation methods can be split into three main groups, 3-dimensional grid-based methods, 2.5-dimensional height field representations and particle based techniques. The last part discusses methods for handling interactions between objects of different types.

Chapter 2

Introduction to Solids



For physically-based simulation, solid objects are typically divided into three main groups: Rigid bodies, soft bodies and cloth. From the point of view of the underlying physics, there is no such distinction. Completely rigid bodies do not exist in nature, every object is deformable to some extent. Cloth is a general three-dimensional soft body as well since it has a certain thickness. However, from an algorithmic and simulation point of view, it makes sense to handle those three types separately. The assumption that objects made of stone are infinitely rigid produces no visual artifacts but simplifies the handling and simulation of such objects significantly. Also, simulating cloth as a 2d rather than a 3d object reduces simulation time and memory consumption.

There is a large body of work concerning the simulation of solid objects in computer graphics. We refer the interested reader to the two surveys [GM97] and [NMK$^+$05]. Since the early work of Terzopoulos and others [TPBF87] in the 80's, many techniques have been proposed to simulate solid objects. In the early days physical simulations were typically done off-line.

Because a single rigid body can be represented by just a few quantities, simulation of small sets of interacting rigid bodies was possible in real-time in the 80's already ([Hah88]). One of the first systems that made real-time interaction with deformable objects possible was *ArtDefo* described in [JP99]. The system made use of the effective method of model reduction in connection with the boundary element method (BEM) to speed up the simulation and achieve interactive frame rates.

The Finite Element Methods (FEM) is one of the most widely used techniques in computations sciences for the simulation of solid objects. The method reduces general partial differential equations to systems of algebraic equations. In general, these equations are non-linear. Solvers for systems of non-linear equations are typically too slow for real-time performance. In certain scenarios, it is feasible to work with linear approximations, mainly when the deformations are small such as in the analysis of buildings. However, in the case of freely moving highly deformable objects, the artifacts are significant. One way to make FEM-based simulations fast enough for real-time applications is to decompose deformations into linear and rotational parts [MG04] as detailed in Chapter 4.

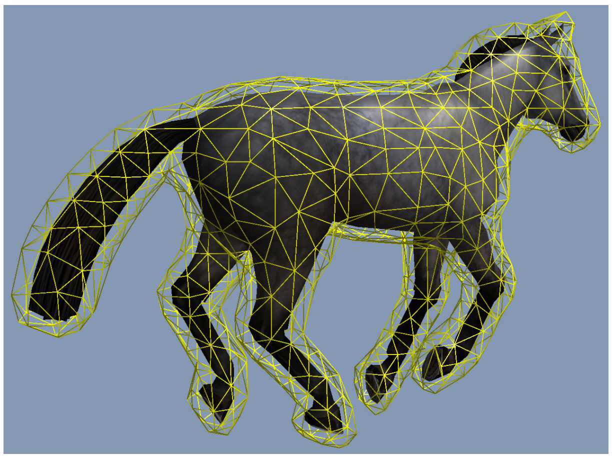
In real-time applications, solid objects are often represented by mass spring networks rather then FEM meshes. Mass spring systems will be discussed in Chapter 3. They are easier to program than FEM and faster in general with disadvantage that they are harder to tune and do not converge to the true solution as the mesh spacing goes to zero. Most of the time, this is not a significant problem in real-time scenarios. Cloth is almost always represented by mass-spring networks because these meshes can easily handle the 2-dimensional structure of cloth. In the FEM framework, special complex elements which can represent bending would have to be used for cloth, a solution that is unnecessarily complex and too slow in general. As mentioned in the introduction, it is essential that simulations are unconditionally stable. Simple explicit integration schemes have no such guarantee. Implicit integration, on the other hand, are more complex to code, slower and introduce damping. Chapter 5 introduces the concept of position based integration which allows to directly control the behavior of the simulation [Jak01, MHR06].

With the assumption of a body to be rigid, its state can be described by a single position, orientation and a linear and angular velocity. This observation allows the simulation of a large number of rigid bodies in real time as described in Chapter 6. Rigid body simulators are the essential part of every physics engine in games because most of the objects present in a level can be considered to be rigid.

Chapter 3

Mass Spring Systems

Matthias Müller



One of the simplest approaches to simulate solid deformable objects is to represent and simulate them as a mass spring system. Since this approach is so simple, it is an ideal framework to study various techniques for simulation and time integration. A mass spring system consists of a set of point masses that are connected by springs. The physics of such a system is straight forward and a simulator can be programmed on just a few pages. This simplicity comes at the price of a few drawbacks you have to be aware of:

- The behavior of the object depends on the way the spring network is set up
- It can be difficult to tune the spring constants to get the desired behavior.
- Mass spring networks cannot capture volumetric effects directly such as volume conservation or prevention of volume inversions.

For various applications these drawbacks are not essential. In those cases, mass spring systems are the best choice because they are easy to implement and fast. If more accurate physics is needed, other approaches like the Finite Element Methods detailed in Chapter 4 should be considered.

3.1 Physical Formulation

A mass spring system is composed of a set of N particles with masses m_i, positions $\mathbf{x}_i$ and velocities $\mathbf{v}_i$, with $i \in 1 \ldots N$. These particles are connected by a set S of springs (i, j, l_0, k_s, k_d), where i and j are the indices of the adjacent particles, l_0 the rest length, k_s the spring stiffness and k_d the damping coefficient. The spring forces acting on the adjacent particles of a spring are

$$\mathbf{f}_i = \mathbf{f}^s(\mathbf{x}_i, \mathbf{x}_j) = k_s \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|}(|\mathbf{x}_j - \mathbf{x}_i| - l_0) \tag{3.1}$$

$$\mathbf{f}_j = \mathbf{f}^s(\mathbf{x}_j, \mathbf{x}_i) = -\mathbf{f}^s(\mathbf{x}_i, \mathbf{x}_j) = -\mathbf{f}_i \tag{3.2}$$

These forces conserve momentum ($\mathbf{f}_i + \mathbf{f}_j = \mathbf{0}$) and are proportional to the elongation of the spring $(|\mathbf{x}_j - \mathbf{x}_i| - l_0)$. Alternatively, one can make them proportional to the *relative* elongation by replacing the constant k_s with the constant k_s/l_0. The damping forces

$$\mathbf{f}_i = \mathbf{f}^d(\mathbf{x}_i, \mathbf{v}_i,\ \mathbf{x}_j, \mathbf{v}_j) = k_d(\mathbf{v}_j - \mathbf{v}_i) \cdot \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|} \tag{3.3}$$

$$\mathbf{f}_j = \mathbf{f}^d(\mathbf{x}_j, \mathbf{v}_j,\ \mathbf{x}_i, \mathbf{v}_i) = -\mathbf{f}_i \tag{3.4}$$

are proportional to the velocity difference projected onto the spring and are momentum conserving as well. Let us combine the two forces into one unified spring force as

$$\mathbf{f}(\mathbf{x}_i, \mathbf{v}_i,\ \mathbf{x}_j, \mathbf{v}_j) = \mathbf{f}^s(\mathbf{x}_i, \mathbf{x}_j) + \mathbf{f}^d(\mathbf{x}_i, \mathbf{v}_i,\ \mathbf{x}_j, \mathbf{v}_j) \tag{3.5}$$

3.2 Simulation

Newton's second law of motion $\mathbf{f} = m\ddot{\mathbf{x}}$ is the key to get from the definition of forces to a simulation algorithm. Solving for the acceleration yields $\ddot{\mathbf{x}} = \mathbf{f}/m$, where $\ddot{\mathbf{x}}$ is the second derivative of the position with respect to time. This formula can be used to compute the accelerations of the particles based on the forces acting on them. As a first step towards simulation, we separate this second order ordinary differential equation into two coupled first order equations as

$$\dot{\mathbf{v}} = \mathbf{f}(\mathbf{x}, \mathbf{v})/m \tag{3.6}$$

$$\dot{\mathbf{x}} = \mathbf{v} \tag{3.7}$$

The analytical solutions of these equations are

$$\mathbf{v}(t) = \mathbf{v}_0 + \int_{t_0}^{t} \mathbf{f}(t)/m\, dt \text{ and} \tag{3.8}$$

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t} \mathbf{v}(t) dt. \tag{3.9}$$

Starting from the initial conditions $\mathbf{v}(t_0) = \mathbf{v}_0$ and $\mathbf{x}(t_0) = \mathbf{x}_0$, the integrals sum the infinitesimal changes up to time t. Simulation is the same as computing $\mathbf{x}(t)$ and $\mathbf{v}(t)$ from time t_0

on. Thus, the words *simulation* and *time integration* are often used interchangeably. The simplest way to solve the equations numerically is to approximate the derivatives with finite differences

$$\dot{\mathbf{v}} = \frac{\mathbf{v}^{t+1} - \mathbf{v}^t}{\Delta t} + O(\Delta t^2) \text{ and} \tag{3.10}$$

$$\dot{\mathbf{x}} = \frac{\mathbf{x}^{t+1} - \mathbf{x}^t}{\Delta t} + O(\Delta t^2), \tag{3.11}$$

where Δt is a discrete time step and t the frame number. Substituting these approximations into Eq. (3.6) and Eq. (3.7) yields two simple update rules

$$\mathbf{v}^{t+1} = \mathbf{v}^t + \Delta t\ \mathbf{f}(\mathbf{x}^t, \mathbf{v}^t)/m \tag{3.12}$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \Delta t\ \mathbf{v}^t. \tag{3.13}$$

This is the explicit Euler integration scheme. It is an explicit scheme because the quantities of the next time step can directly be computed from the quantities at the current time step using explicit formulas. A small trick which makes the scheme more stable is to use $\mathbf{v}^{t+1}$ in stead of $\mathbf{v}^t$ on the right hand side of Eq. (3.13) because the new velocity is already available at that point in time. We are now ready to write down an algorithm for the simulation of a mass spring system:

```
// initialization
(1)  forall particles $i$
(2)      initialize $\mathbf{x}_i, \mathbf{v}_i$ and $m_i$
(3)  endfor
// simulation loop
(4)  loop
(5)      forall particles $i$
(6)          $\mathbf{f}_i \leftarrow \mathbf{f}^g + \mathbf{f}_i^{\text{coll}} + \sum_{j,(i,j)\in S} \mathbf{f}(\mathbf{x}_i, \mathbf{v}_i,\ \mathbf{x}_j, \mathbf{v}_j)$
(7)      endfor
(8)      forall particles $i$
(9)          $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\ \mathbf{f}_i/m_i$
(10)         $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t\ \mathbf{v}_i$
(11)     endfor
(12)     display the system every $n^{th}$ time
(13) endloop
```

Here, $\mathbf{f}^g$ is the gravity force and $\mathbf{f}^{\text{coll}}$ forces due to collisions. Explicit Euler integration is one of the simplest integration methods but it has an essential drawback. It is only stable for relatively small steps. We will not go into the details of stability analysis. Just to give you an idea, the time step in a typical real-time scenario has to be of the order of $10^{-4} \ldots 10^{-3}$ seconds meaning that several simulation steps are necessary between two visualized frames.

This is why in the above algorithm step (12) is executed only every n^{th} time. The main reason for the instability is that the Euler scheme steps blindly into the future. It assumes that the force is constant throughout an entire step. Consider the situation when a spring is slightly stretched and the adjacent particles move towards each other. With a large time step, the particles pass the equilibrium configuration so the spring force changes sign during the time step. This is not accounted for if the force at the beginning is used throughout the entire time step. In this particular situation, the particles *overshoot* and gain energy which can result in an explosion eventually.

One way to improve the situation is to use more accurate integration schemes. Popular choices are second and fourth order *Runge-Kutta* integrators. These schemes sample the forces multiple times within the time step to reduce the problem mentioned.

3.3 Runge-Kutta Integration

The second order Runge-Kutta integrator replaces Eq. (3.12) and Eq. (3.13) with

$$\begin{aligned} \mathbf{a}_1 &= \mathbf{v}^t & \mathbf{a}_2 &= \mathbf{f}(\mathbf{x}^t, \mathbf{v}^t)/m \\ \mathbf{b}_1 &= \mathbf{v}^\mathbf{t} + \frac{\Delta t}{2}\mathbf{a}_2 & \mathbf{b}_2 &= \mathbf{f}(\mathbf{x}^t + \frac{\Delta t}{2}\mathbf{a}_1, \mathbf{v}^t + \frac{\Delta t}{2}\mathbf{a}_2)/m \\ \mathbf{x}^{t+1} &= \mathbf{x}^t + \Delta t \mathbf{b}_1 & \mathbf{v}^{t+1} &= \mathbf{v}^t + \Delta t \mathbf{b}_2 \end{aligned}$$

to get from the position $\mathbf{x}^t$ and velocity $\mathbf{v}^t$ at the current time step to the position and velocity $\mathbf{x}^{t+1}$ and $\mathbf{v}^{t+1}$ at the next time step. Here is the modified algorithm which uses a second order Runge Kutta integrator:

```
// initialization
(1)  forall particles i
(2)      initialize $\mathbf{x}_i, \mathbf{v}_i$ and $m_i$
(3)  endfor
// simulation loop
(4)  loop
(5)      forall particles i
(6)          $\mathbf{a}_{1,i} \leftarrow \mathbf{v}_i$
(7)          $\mathbf{a}_{2,i} \leftarrow \left[\mathbf{f}^g + \mathbf{f}_i^{\text{coll}} + \sum_{j,(i,j)\in S} \mathbf{f}(\mathbf{x}_i, \mathbf{v}_i, \mathbf{x}_j, \mathbf{v}_j)\right]/m_i$
(8)      endfor
(9)      forall particles i
(10)         $\mathbf{b}_{1,i} \leftarrow \mathbf{v}_i + \frac{\Delta t}{2}\mathbf{a}_{2,i}$
(11)         $\mathbf{b}_{2,i} \leftarrow \left[\mathbf{f}^g + \mathbf{f}_i^{\text{coll}} + \sum_{j,(i,j)\in S} \mathbf{f}(\mathbf{x}_i + \frac{\Delta t}{2}\mathbf{a}_{1,i}, \mathbf{v}_i + \frac{\Delta t}{2}\mathbf{a}_{2,i}, \mathbf{x}_j + \frac{\Delta t}{2}\mathbf{a}_{1,j}, \mathbf{v}_j + \frac{\Delta t}{2}\mathbf{a}_{2,j})\right]/m_i$
(12)         $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{b}_{1,i}$
(13)         $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \mathbf{b}_{2,i}$
(14)     endfor
(15)     display the system every $n^{th}$ time
```

(16) **endloop**

As you can see, the forces have to be evaluated twice at each time step. So one second order Runge-Kutta step takes as much time as two Euler steps. However, Runge-Kutta is second order accurate in contrast to the first order accurate Euler scheme. This means that when you half the time step using Euler, you end up with half the error in contrast to one fourth of the error in the Runge-Kutta case. There is a fourth order Runge Kutta integration scheme which looks like this:

$$\begin{aligned}
\mathbf{a}_1 &= \mathbf{v}^t & \mathbf{a}_2 &= \mathbf{f}(\mathbf{x}^t, \mathbf{v}^t)/m \\
\mathbf{b}_1 &= \mathbf{v}^t + \frac{\Delta t}{2}\mathbf{a}_2 & \mathbf{b}_2 &= \mathbf{f}(\mathbf{x}^t + \frac{\Delta t}{2}\mathbf{a}_1, \mathbf{v}^t + \frac{\Delta t}{2}\mathbf{a}_2)/m \\
\mathbf{c}_1 &= \mathbf{v}^t + \frac{\Delta t}{2}\mathbf{b}_2 & \mathbf{c}_2 &= \mathbf{f}(\mathbf{x}^t + \frac{\Delta t}{2}\mathbf{b}_1, \mathbf{v}^t + \frac{\Delta t}{2}\mathbf{b}_2)/m \\
\mathbf{d}_1 &= \mathbf{v}^t + \Delta t\mathbf{c}_2 & \mathbf{d}_2 &= \mathbf{f}(\mathbf{x}^t + \Delta t\mathbf{c}_1, \mathbf{v}^t + \Delta t\mathbf{c}_2)/m \\
\mathbf{x}^{t+1} &= \mathbf{x}^t + \frac{\Delta t}{6}(\mathbf{a}_1 + 2\mathbf{b}_1 + 2\mathbf{c}_1 + \mathbf{d}_1) & \mathbf{v}^{t+1} &= \mathbf{v}^t + \frac{\Delta t}{6}(\mathbf{a}_2 + 2\mathbf{b}_2 + 2\mathbf{c}_2 + \mathbf{d}_2)
\end{aligned}$$

This scheme is among the most popular integration techniques in computational sciences. The forces have to be evaluation four times which is rewarded by fourth order accuracy. Halving the time step results in one sixteenth of the error.

3.4 Verlet Integration

The way used in the Runge-Kutta scheme to improve stability and accuracy was to sample the forces multiple times during one time step. Another idea is to use quantities evaluated in the past to increase the degree of the approximation of the derivatives in Eq. (3.10) Eq. (3.11). A variety of schemes are based on this idea. In this class, Verlet integration is among the simplest and most popular in real-time applications. The basic idea is to keep the positions of time $t - \Delta t$ in an additional state variable and use this information for a more accurate prediction. Taylor expansion of the positions in the two time directions yields

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta t + \frac{1}{2}\ddot{\mathbf{x}}(t)\Delta t^2 + \frac{1}{6}\dddot{\mathbf{x}}(t)\Delta t^3 + O(\Delta t^4) \tag{3.14}$$

$$\mathbf{x}(t-\Delta t) = \mathbf{x}(t) - \dot{\mathbf{x}}(t)\Delta t + \frac{1}{2}\ddot{\mathbf{x}}(t)\Delta t^2 - \frac{1}{6}\dddot{\mathbf{x}}(t)\Delta t^3 + O(\Delta t^4) \tag{3.15}$$

Adding these two equations and rearranging terms gives

$$\begin{aligned}
\mathbf{x}(t+\Delta t) &= 2\mathbf{x}(t) - \mathbf{x}(t-\Delta t) + \ddot{\mathbf{x}}(t)\Delta t^2 + O(\Delta t^4) && (3.16)\\
&= \mathbf{x}(t) + [\mathbf{x}(t) - \mathbf{x}(t-\Delta t)] + \mathbf{f}(t)\Delta t^2/m + O(\Delta t^4). && (3.17)\\
& && (3.18)
\end{aligned}$$

As you can see, the linear and cubic terms cancel out which makes this a fourth order integration scheme. The velocity does not show up explicitly. Instead, the position at the previous time step has to be stored. We can bring the velocity back in by *defining* $\mathbf{v}(t) = [\mathbf{x}(t) - \mathbf{x}(t - \Delta t)]/\Delta t$. Incorporating this idea and going back to the frame time notation yields

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \mathbf{v}^t \Delta t + \mathbf{f}(\mathbf{x}^t)\Delta t^2/m \tag{3.19}$$

$$\mathbf{v}^{t+1} = (\mathbf{x}^{t+1} - \mathbf{x}^t)/\Delta t. \tag{3.20}$$

Unlike in the related *Velocity Verlet* scheme, the velocity here is only first order accurate and not really ”used”. It is just a different way of representing the position at the last time step. The interesting fact that this scheme operates on positions only is leveraged in the *Position Based Dynamics* approach introduced in Chapter 5.

3.5 Implicit Integration

The integration schemes discussed so far are only *conditionally stable* meaning that there is a certain range for the time step Δt size for which the simulation is stable. This range depends mainly on the stiffness of the springs. The stiffer the springs, the smaller the time step required to keep the simulation stable. In real-time situation, e.g. in a computer game, it is essential that an integration is *unconditionally stable* meaning stable in all circumstances and for the time step size given by the required frame rate. One way to achieve this is to use an *implicit* integration scheme. Another possibility will be discussed in Chapter 5.

The most popular implicit integration scheme used in computer graphics is implicit Euler. The explicit Euler scheme described in Eq. (3.12) and Eq. (3.13) has to be modified only slightly to make it implicit:

$$\mathbf{v}^{t+1} = \mathbf{v}^t + \Delta t\, \mathbf{f}(\mathbf{x}^{t+1})/m \tag{3.21}$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \Delta t\, \mathbf{v}^{t+1}. \tag{3.22}$$

We removed the friction force so the sum of all forces only depends on the positions, not the velocities. Friction forces stabilize the system so they can be added in an explicit step after the implicit solve. Second, implicit integration introduces quite a lot of numerical damping so in typical situations there is no need to add physical damping as well.

The core change, however, is to use the positions and velocities of the new time step on the right hand side as well. It is not possible any more to directly and explicitly evaluate these two equations. Instead, we have two implicit equations that form a non-linear algebraic system with the positions and velocities of the next time step as the unknowns. Non formally one could say that in contrast to explicit schemes, we don’t move blindly into the future but make sure that the quantities we arrive at are in accordance with physical laws.

As a first step towards solving these equations for the entire mass spring network we combine the positions, velocities and forces of all particles into two single vectors and a multidimensional force

$$\begin{aligned}
\mathbf{x} &= [\mathbf{x}_1^T, \dots, \mathbf{x}_n^T]^T \\
\mathbf{v} &= [\mathbf{v}_1^T, \dots, \mathbf{v}_n^T]^T \\
\mathbf{f}(\mathbf{x}) &= [\mathbf{f}_1(\mathbf{x}_1, \dots, \mathbf{x}_n)^T, \dots \mathbf{f}_n(\mathbf{x}_1, \dots \mathbf{x}_n)^T]^T
\end{aligned}$$

We also construct a mass matrix $\mathbf{M} \in \mathbb{R}^{3n\times 3n}$ which is diagonal with the values m_1, m_1, m_1, $m_2, m_2, m_2, \ldots, m_n, m_n, m_n$ along the diagonal. These definitions lead to the following system of equations

$$\mathbf{M}\mathbf{v}^{t+1} = \mathbf{M}\mathbf{v}^t + \Delta t \mathbf{f}(\mathbf{x}^{t+1}) \tag{3.23}$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \Delta \mathbf{v}^{t+1} \tag{3.24}$$

Substituting Eq. (3.24) into Eq. (3.23) results in a single system for the unknown velocities $\mathbf{v}^{t+1}$ at the next time step

$$\mathbf{M}\mathbf{v}^{t+1} = \mathbf{M}\mathbf{v}^t + \Delta t \mathbf{f}(\mathbf{x}^t + \Delta t \mathbf{v}^{t+1}). \tag{3.25}$$

This system is non-linear because the forces are non-linear in the positions.

3.5.1 Newton-Raphson Solver

The general way to solve such a system is to use the Newton-Raphson method. This method starts at a guess for the unknown $\mathbf{v}^{t+1}$ and iteratively improves this guess. To this end, the equations are *linearized* at the current state and the resulting linear system is solved to find a better approximation. This process is repeated until the error falls below a certain threshold.

For real-time applications, linearizing multiple times per time step is too expensive. It is custom, therefore, to linearize once and use the current velocities $\mathbf{v}^t$ as a guess for $\mathbf{v}^{t+1}$. Newton's first law says that without forces, velocities do not change which makes this guess a good one. Linearizing the forces at $\mathbf{x}^t$ yields

$$\mathbf{M}\mathbf{v}^{t+1} = \mathbf{M}\mathbf{v}^t + \Delta t \left[\mathbf{f}(\mathbf{x}^t) + \frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{x}^t) \cdot (\Delta t\, \mathbf{v}^{t+1})\right] \tag{3.26}$$

$$= \mathbf{M}\mathbf{v}^t + \Delta t \mathbf{f}(\mathbf{x}^t) + \Delta t^2 \mathbf{K}\mathbf{v}^{t+1}, \tag{3.27}$$

where $\mathbf{K} \in \mathbb{R}^{3n\times 3n}$ is the Jacobian of $\mathbf{f}$. It contains the derivatives of all $3n$ force components w.r.t. all $3n$ position components of the particles. This matrix is also called the *tangent stiffness matrix* in this context. While a standard stiffness matrix of an elastic element is evaluated at the equilibrium, $\mathbf{K}$ is evaluated at the current positions of the particles. We rearrange the terms to get a standard linear system for the unknown velocities

$$\left[\mathbf{M} - \Delta t^2 \mathbf{K}\right] \mathbf{v}^{t+1} = \mathbf{M}\mathbf{v}^t + \Delta t \mathbf{f}(\mathbf{x}^t) \tag{3.28}$$

$$\mathbf{A}\mathbf{v}^{t+1} = \mathbf{b}, \tag{3.29}$$

where $\mathbf{A}$ is a $3n \times 3n$ dimensional matrix and $\mathbf{b}$ a $3n$ dimensional vector. In real-time applications, this system is typically solved using iterative methods like Conjugate Gradients (CG). Direct methods are not practical because $\mathbf{K}$ changes at each time step. The right hand side vector $\mathbf{b}$ can directly be evaluated using known quantities only.

Now let us have a look at $\mathbf{K}$. A spring force between particles i and j adds the four 3×3 sub-matrices $\mathbf{K}_{i,i}$, $\mathbf{K}_{i,j}$, $\mathbf{K}_{j,i}$ and $\mathbf{K}_{j,j}$ to the global matrix $\mathbf{K}$ at positions $(3i, 3i)$, $(3i, 3j)$, $(3j, 3i)$ and $(3j, 3j)$ respectively. In order to evaluate these sub-matrices, we need to deduce the derivatives of the spring force defined in Eq. (3.1) w.r.t. the positions $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$\mathbf{K}_{i,i} = \frac{\partial}{\partial \mathbf{x}_i}\mathbf{f}^s(\mathbf{x}_i,\mathbf{x}_j) \tag{3.30}$$

$$= k_s \frac{\partial}{\partial \mathbf{x}_i}\left((\mathbf{x}_j - \mathbf{x}_i) - l_0 \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|}\right) \tag{3.31}$$

$$= k_s \left(-\mathbf{I} + \frac{l_0}{l}\left[\mathbf{I} - \frac{(\mathbf{x}_j - \mathbf{x}_i)(\mathbf{x}_j - \mathbf{x}_i)^T}{l^2}\right]\right) \tag{3.32}$$

$$= -\mathbf{K}_{i,j} = \mathbf{K}_{j,j} = -\mathbf{K}_{j,i} \tag{3.33}$$

where $\mathbf{I}$ is the 3×3 identity matrix and $l = |\mathbf{x}_j - \mathbf{x}_i|$ the actual length of the spring. The tangent stiffness matrix is not constant throughout the simulation because it depends on the current positions $\mathbf{x}_1 \ldots \mathbf{x}_n$ of the particles. So at each time step $\mathbf{K}$ is set to zero. Then, for each spring the four sub-matrices $\mathbf{K}_{i,i}$, $\mathbf{K}_{i,j}$, $\mathbf{K}_{j,i}$ and $\mathbf{K}_{j,j}$ are added at the right positions in $\mathbf{K}$. Next, the right hand side $\mathbf{b}$ of Eq. (3.29) is evaluated and the resulting linear system solved for $\mathbf{v}^{t+1}$. Finally the velocities can be updated using Eq. (3.24).

The tangent stiffness matrix is sparse in the general case. It would only be fully occupied if each particle was connected to all other particles which is typically not the case. Because all sub-matrices associated with a spring are equal up to their sign, only one 3×3 matrix has to be stored per spring.

As we saw in this chapter, there is a wide spectrum of integration methods ranging from very simple but unstable methods to quite complex unconditionally stable schemes. Which choice is optimal depends on the application. Implicit integration has two main advantages, its stability and the possibility of using a global solver such as Conjugate Gradients. Think of a long thin triangle mesh. If you pull on one side, the explicit Euler scheme can only propagate forces one triangle per integration step. A global solver propagates errors much faster across the unknowns in the system. This way, deformable objects can be made stiff. This is important in connection with cloth simulation for instance. Cloth has low bending resistance but is quite stiff w.r.t. stretching. On the other hand, in order to achieve real-time performance, the time step for an implicit integrator has to be chosen quite large because one integration step is so expensive. Also, implicit integration introduces substantial numerical damping. Both effects let interesting spatial and temporal detail disappear.

3.6 Mesh Creation

In computer games, artists model 3d objects as graphical meshes that represent their surface. These meshes are often not well suited to be used for simulation directly for several reasons. First, they do not represent the interior of the object and second, they contain badly shaped triangles, duplicated vertices, intersecting triangles or are just triangle soups in the extreme case. In general, it is better to use two representations of the object, one for visualization and one for simulation. Sometimes a third representation for collision detection is needed as well. If the simulation and visual representations are not the same, a mechanism to move the visual mesh along with the physical mesh is needed. If the connectivity is based on a tetrahedral mesh, one can simply use the barycentric coordinates of each visual vertex w.r.t. its surrounding tetrahedron to move the vertex via skinning.

There are many ways to create a simulation mesh from a graphical mesh. Let us have a look at a procedure that is quite simple. We create the mass spring system from a *tetrahedral mesh* by turning each vertex into a particle and each tetrahedral edge into a spring. What is missing are the masses of the particles and the stiffness and damping coefficients of the springs. Given a user specified density ρ, the mass of each tetrahedron can be computed as its volume multiplied by the density. Then, each tetrahedron distributes its mass evenly among its four adjacent vertices. This is a straight forward procedure. Finding reasonable spring coefficients is not. No matter how you do it, the behavior of the mass spring system will always depend on the structure of the mesh. This problem can only be solved by considering volumetric tetrahedra for simulation (see Chapter 4), not just their one dimensional edges. Since there is no correct way, we might as well assign a common stiffness and damping coefficient to all springs. If the lengths of the springs do not vary too much, this approach works quite well. Otherwise you might want to come up with some magic to compute the coefficients from the edge lengths.

The remaining question is how to go from a triangular surface mesh to a volumetric tetrahedral mesh. To this end, we distribute a certain number of particles p_i evenly inside the volume defined by the surface mesh and on the surface mesh itself. To do this, we need an inside-outside test. The traditional way is to create a ray originating from the particle and count the number of surface triangle intersections. If the number is odd, the particle is inside the surface. This procedure requires the surface to be watertight. Then we run Delaunay tetrahedralization on these points and only keep tetrahedra with centers inside the surface. A nice feature of this approach is that the resulting tetrahedralization is independent of the triangulation of the surface. There is a pretty simple formulation of the Delaunay algorithm which constructs a tetrahedral mesh from a set of points:

```
(1)  create large tetrahedron that contains all points $p_1, \ldots, p_n$
     using four new far points $q_0, \ldots, q_3$.
(2)  **forall** points $p_i$
(3)     clear face list $l$
(4)     **forall** tetrahedra $t_j$ the circumsphere of which contains $p_i$
(5)        **forall** faces $f_k$ of $t_j$
(6)           **if** $l$ contains $f_k$ remove $f_k$ from $l$
(7)           **else** add $f_k$ to $l$
(8)        **endfor**
(9)        delete $t_j$
(10)    **endfor**
(11)    **forall** faces $f_k$ in $l$
(12)       create a tetrahedron from $f_k$ to $p_i$
(13)    **endfor**
(14) **endfor**
(15) remove all tetrahedra that contain any of the points $q_0, \ldots, q_3$
```

3.7 Collision Detection

There is a large body of work on collision detection for deformable objects [TKZ+04]. In this section we will look at spatial hashing only [THM+03]. This method works well when all primitives in a scene have about the same size. An advantage of using a separate representation of the object for physics is, that this representation can be optimized for a specific simulation algorithm. In this case, we make sure that there is not much variation in the sizes of the tetrahedra.

Spatial hashing works as follows: First, a grid spacing h is chosen. The spacing should match the average size of the primitives. It defines a regular grid in space. Then, each tetrahedron is inserted into all grid cells it intersects. With this data structure in place, the collision detection algorithm can retrieve all objects in the neighborhood of a point in constant time. The integer coordinates of the grid cell a point $\mathbf{p}$ lies in are

$$(i_x, i_y, i_z) = (\lfloor \frac{\mathbf{p}_x}{h} \rfloor, \lfloor \frac{\mathbf{p}_y}{h} \rfloor, \lfloor \frac{\mathbf{p}_z}{h} \rfloor). \tag{3.34}$$

Obviously the number of grid cells defined this way is not bounded. In order to store the grid, one could restrict the objects to remain in a finite rectangular volume and then only work with the cells in this volume. A more flexible variant is to allocate a fixed number N of buckets and then map the cell coordinates to a bucket number using a hash function. Since there are way more real cells then buckets, multiple cells will be mapped to the same bucket. This is not a problem though because even though a spatial query might return tetrahedra from other parts of the world, it will always return all true candidates as well. Thus, hash collisions slow down collision detection but do not make it less accurate. Here is an example of a hash function

$$i = [(i_x \cdot 92837111) \text{ xor } (i_x \cdot 689287499) \text{ xor } (i_x \cdot 283923481)] \bmod N, \tag{3.35}$$

where i is the bucket number cell (i_x, i_y, i_z) is mapped to. At each time step, the grid is recreated from scratch. If N is substantially larger than the number of tetrahedra, clearing all the buckets at each time step is too expensive. Instead, each bucket contains a frame counter. If this counter is smaller than the global frame counter, the bucket is defined to be empty and whenever an object is inserted, the counter of the bucket is set to the global counter. At the beginning of collision detection, the global frame counter is increased which makes all buckets empty at once. Then, each tetrahedron is added to the buckets corresponding to the cells it overlaps. Finally, the algorithm iterates through all the particles on the surface of the mesh and retrieves all the tetrahedra in the neighborhood of that particle. If the surface particle is inside a tetrahedron, a collision is reported. This method works for both, inter and intra object collision. It does not handle edge-edge collisions though.

3.8 Collision Response

The basic event reported from collision detection is that a surface particle $\mathbf{q}$ of an object A is inside a tetrahedron t of an object B. In the case of a self collision, the two objects are identical. It is far from trivial to find a stable way of resolving such a collision. The first problem is to decide where the penetrated vertex should go. One possibility is to move it to the closest surface point of object B. In general, this is not a very good choice. It is more natural and more stable to move $\mathbf{q}$ back to where it penetrated object B. A way to achieve this is to construct a ray from $\mathbf{q}$ in the opposite direction of the surface normal at $\mathbf{q}$. Let us call the intersection of the ray with B's surface $\mathbf{q}'$. This point lies inside a face of a tetrahedron of B with adjacent vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$. Let β_1, β_2 and β_3 be the barycentric coordinates of $\mathbf{q}'$ such that

$$\mathbf{q}' = \beta_1 \mathbf{p}_1 + \beta_2 \mathbf{p}_2 + \beta_3 \mathbf{p}_3. \tag{3.36}$$

We can now define the collision response force

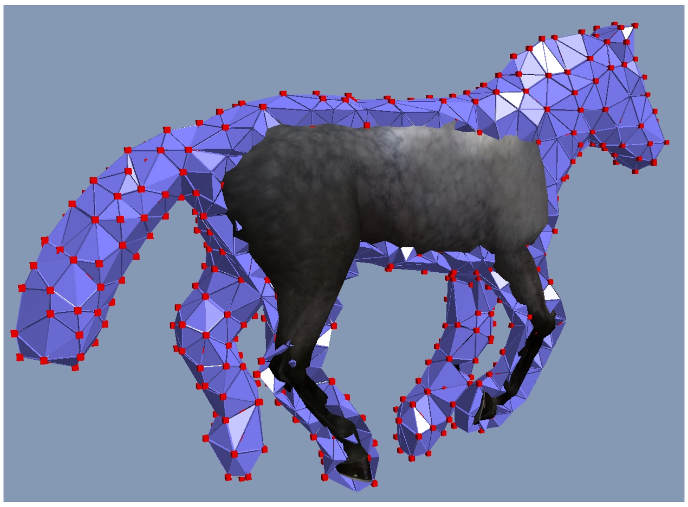
$$\mathbf{f}_{\text{coll}} = k(\mathbf{q}' - \mathbf{q}) \tag{3.37}$$

which is proportional to the penetration depth and a user specified stiffness coefficient. This force is applied to vertex $\mathbf{q}$. To make sure the momenta are conserved forces $-\beta_1 \mathbf{f}_{\text{coll}}$, $-\beta_2 \mathbf{f}_{\text{coll}}$ and $-\beta_2 \mathbf{f}_{\text{coll}}$ are applied to vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ respectively. The method described is quite simplistic. A more sophisticated and more robust technique is described in [HTK$^+$04].

Chapter 4

The Finite Element Method

Matthias Müller



Mass spring system cannot capture volumetric effects, as mentioned in the previous chapter. Also, their behavior depends on the structure of the mesh. To fix these problems, we will now treat the deformable body as a continuous volume. To this end, we will look into the world of continuum mechanics. Of course, summarizing such a complex mathematical framework in its generality on just a few pages is not possible. Therefore, we will focus on the most important concepts needed for the simulation of deformable objects in computer graphics. Do not worry if you don't grasp all the details of this section the first time.

In Section 4.2 we will turn the resulting partial differential equations into a relatively simple formula for computing the forces that act on a tetrahedron based on the positions of its vertices. Given this formula, one can think of the tetrahedron as a generalized four-way spring and build larger structures from it like in mass-spring systems.

In the last section we will look at ways to evaluate the forces efficiently in order to enable simulation of large tetrahedral meshes in real time.

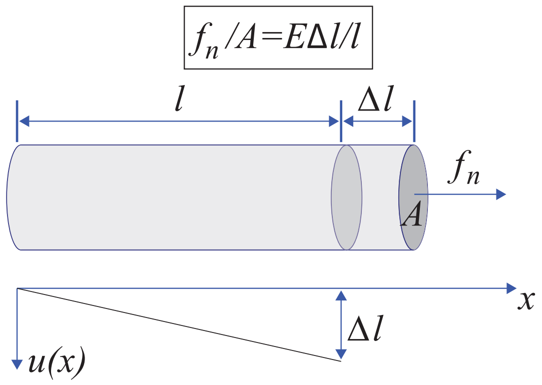


Figure 4.1: Hooke's law says that the force per area applied to a beam is proportional to its relative elongation.

4.1 Continuum Mechanics

In computational sciences, deformable objects are often modeled as continuous (three dimensional) objects. For the description of their behavior, the three quantities *displacement*, *strain* and *stress* play a major role. In one dimensional problems, these quantities are all one dimensional and have intuitive interpretations. Let us consider a beam with cross sectional area A as shown in Fig. 4.1. When a force f_n is applied in the direction of the beam perpendicular to the cross section, the beam with original length l expands by Δl. These quantities are related via Hooke's law as

$$\frac{f_n}{A} = E\frac{\Delta l}{l}. \tag{4.1}$$

The constant of proportionality E is Young's modulus. For steel E is in the order of $10^{11} N/m^2$ while for rubber it lies between 10^7 and $10^8 N/m^2$. The equation states that the stronger the force per area, the larger the relative elongation $\Delta l/l$ as expected. Also, the magnitude of the force per area needed to get a certain relative elongation increases with increasing E so Young's modulus describes the beam's *stiffness*. Hooke's law can be written in a more compact form as

$$\sigma = E\varepsilon, \tag{4.2}$$

where $\sigma = f_n/A$ is the applied *stress* and $\varepsilon = \Delta l/l$ the resulting *strain* or the other way around, σ the resulting internal stress due to applied strain ε. It follows that the unit of stress is force per area and that strain has no unit. This is true in three dimensions as well.

A three dimensional deformable object is typically defined by its undeformed shape (also called equilibrium configuration, original, rest or initial shape) and by a set of material parameters that define how it deforms under applied forces. If we think of the rest shape as a continuous connected subset Ω of $\mathbb{R}^3$, then the coordinates $\mathbf{x} \in \Omega$ of a point in the object are called *material coordinates* of that point.

When forces are applied, the object deforms and a point originally at location $\mathbf{x}$ (i.e. with material coordinates $\mathbf{x}$) moves to a new location $\mathbf{p}(\mathbf{x})$, the *spatial* or *world coordinates* of that point. Since new locations are defined for all material coordinates $\mathbf{x}$, $\mathbf{p}(\mathbf{x})$ is a vector field defined on Ω. Alternatively, the deformation can also be specified by the *displacement* field, which, in three dimensions, is a vector field $\mathbf{u}(\mathbf{x}) = \mathbf{p}(\mathbf{x}) - \mathbf{x}$ defined on Ω.

4.1.1 Strain

In order to formulate Hooke's law in three dimensions, we have to find a way to measure strain, i.e. the relative elongation (or compression) of the material. In the general case, the strain is not constant inside a deformable body. In a bent beam, the material on the convex side is stretched while the one on the concave side is compressed. Therefore, strain is a function of the material coordinate $\varepsilon = \varepsilon(\mathbf{x})$. If the body is not deformed, the strain is zero so strain must depend on the displacement field $\mathbf{u}(\mathbf{x})$. A spatially constant displacement field describes a pure translation of the object. In this situation, the strain should be zero as well. Thus, strain is derived from the *spatial variation* or *spatial derivatives* of the displacement field. In three dimensions the displacement field has three components $\mathbf{u} = \mathbf{u}(\mathbf{x}) = [u(x,y,z), v(x,y,z), w(x,y,z)]^T$ and each component can be derived with respect to one of the three spatial variables x, y and z. Therefore, strain cannot be expressed by a single scalar anymore. This makes sense because at a single point inside a three dimensional object, the material can be stretched in one direction and compressed in another at the same time. Thus, strain is represented in three dimensions by a symmetric 3 by 3 matrix or tensor

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{xy} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{xz} & \varepsilon_{yz} & \varepsilon_{zz} \end{bmatrix} \tag{4.3}$$

We consider two ways to compute the components of the strain tensor from the spatial derivatives of the displacement field

$$\varepsilon_G = \frac{1}{2}(\nabla\mathbf{u} + [\nabla\mathbf{u}]^T + [\nabla\mathbf{u}]^T\nabla\mathbf{u}) \text{ and} \tag{4.4}$$

$$\varepsilon_C = \frac{1}{2}(\nabla\mathbf{u} + [\nabla\mathbf{u}]^T), \tag{4.5}$$

where the symmetric tensor $\varepsilon_G \in \mathbb{R}^{3x3}$ is Green's nonlinear strain tensor (non-linear in the displacements) and $\varepsilon_C \in \mathbb{R}^{3x3}$ its linearization, Cauchy's linear strain tensor. The gradient of the displacement field is a 3 by 3 matrix

$$\nabla\mathbf{u} = \begin{bmatrix} u_{,x} & u_{,y} & u_{,z} \\ v_{,x} & v_{,y} & v_{,z} \\ w_{,x} & w_{,y} & w_{,z} \end{bmatrix}, \tag{4.6}$$

where the index after the comma represents a spatial derivative.

It is natural to derive strain from the spatial derivatives of the displacement field, but the specific definitions of ε_G and ε_C look quite arbitrary. There is a relatively easy way to motivate their definition though. Let us have a look at the the transformation $\mathbf{p}(\mathbf{x})$ in the neighborhood of a material point. Without loss of generality we can choose the material point $\mathbf{0}$. Close to that point, $\mathbf{p}(\mathbf{x})$ can be approximated with a linear mapping as

$$\mathbf{p}(\mathbf{x}) = \mathbf{p}(\mathbf{0}) + \nabla\mathbf{p} \cdot \mathbf{x} + O(|\mathbf{x}|^2) \tag{4.7}$$

$$= \mathbf{p}(\mathbf{0}) + \left[\frac{\partial \mathbf{p}}{\partial x}, \frac{\partial \mathbf{p}}{\partial y}, \frac{\partial \mathbf{p}}{\partial z}\right] \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + O(|\mathbf{x}|^2) \tag{4.8}$$

or

$$\mathbf{p}(x,y,z) \approx \mathbf{p}(\mathbf{0}) + \frac{\partial \mathbf{p}}{\partial x} \cdot x + \frac{\partial \mathbf{p}}{\partial y} \cdot y + \frac{\partial \mathbf{p}}{\partial z} \cdot z \tag{4.9}$$

$$= \mathbf{p}(\mathbf{0}) + \mathbf{p}_{,x} \cdot x + \mathbf{p}_{,y} \cdot y + \mathbf{p}_{,z} \cdot z. \tag{4.10}$$

This shows that the local neighborhood of material point $\mathbf{0}$ is moved into a new frame with origin $\mathbf{p}(\mathbf{0})$ and axes $\mathbf{p}_{,x}, \mathbf{p}_{,y}$ and $\mathbf{p}_{,z}$. The neighborhood does not get distorted by the translation along $\mathbf{p}(\mathbf{0})$. For the neighborhood not to get distorted by the entire transformation, the new axes must satisfy two conditions. All of them must have unit length, otherwise the neighborhood gets stretched or compressed along the axes. In addition, the axes must be perpendicular to each other, otherwise the neighborhood is sheared. These constraints can elegantly be expressed by requiring that $[\nabla\mathbf{p}]^T \nabla\mathbf{p} = \mathbf{I}$. Expanding this expression yields

$$[\nabla\mathbf{p}]^T \nabla\mathbf{p} = \begin{bmatrix} \mathbf{p}_{,x}^T \\ \mathbf{p}_{,y}^T \\ \mathbf{p}_{,z}^T \end{bmatrix} [\mathbf{p}_{,x}\mathbf{p}_{,y}\mathbf{p}_{,z}] = \begin{bmatrix} |\mathbf{p}_{,x}|^2 & \mathbf{p}_{,x} \cdot \mathbf{p}_{,y} & \mathbf{p}_{,x} \cdot \mathbf{p}_{,z} \\ \mathbf{p}_{,y} \cdot \mathbf{p}_{,x} & |\mathbf{p}_{,y}|^2 & \mathbf{p}_{,y} \cdot \mathbf{p}_{,z} \\ \mathbf{p}_{,z} \cdot \mathbf{p}_{,x} & \mathbf{p}_{,z} \cdot \mathbf{p}_{,y} & |\mathbf{p}_{,z}|^2 \end{bmatrix}. \tag{4.11}$$

For this to be equal to the identity matrix, all diagonal entries must be 1 saying that all axes must have unit length. The off-diagonal entries are 0 if all the dot products between pairs of axes are zero. This is the case when they are perpendicular to each other. In other words, $[\nabla\mathbf{p}]^T \nabla\mathbf{p} = \mathbf{I}$ means no strain. From this observation it makes sense to define strain as the deviation of $[\nabla\mathbf{p}]^T \nabla\mathbf{p}$ from $\mathbf{I}$, so

$$\varepsilon = [\nabla\mathbf{p}]^T \nabla\mathbf{p} - \mathbf{I}. \tag{4.12}$$

With this definition, the diagonal entries of ε are greater than zero if the material gets stretched and smaller if the material gets compressed. The off-diagonal entries indicate the amount of shearing. Now from $\mathbf{u}(\mathbf{x}) = \mathbf{p}(\mathbf{x}) - \mathbf{x}$ follows that $\nabla\mathbf{u} = \nabla\mathbf{p} - \mathbf{I}$. Substituting into (4.12) yields

$$\varepsilon = (\nabla\mathbf{u} + \mathbf{I})^T (\nabla\mathbf{u} + \mathbf{I}) - \mathbf{I} \tag{4.13}$$

$$= \nabla\mathbf{u} + [\nabla\mathbf{u}]^T + [\nabla\mathbf{u}]^T \nabla\mathbf{u} \tag{4.14}$$

which is Green's strain tensor up to a factor of $\frac{1}{2}$. Cauchy's strain is simply the linearization of Green's strain omitting the quadratic term $[\nabla\mathbf{u}]^T \nabla\mathbf{u}$.

4.1.2 Strain Examples

It is now time to look at two specific examples. Consider the displacement function $\mathbf{u}(x,y,z) = [x,y,z]^T$. It stretches material radially away from the origin. In this case $\nabla\mathbf{u} = \mathbf{I}$ with $\mathbf{I}$ the identity matrix. The two strain measures are $\varepsilon_G = \frac{3}{2}\mathbf{I}$ and $\varepsilon_C = \mathbf{I}$. Even though the displacements get larger and larger for points further and further away from the origin, the strain is constant everywhere. The two measures differ but only by a constant scalar factor.

In the second example we consider a rotation of 90 degrees about the z-axis. As a rigid-body mode, such a displacement field should not generate any strain. We have $\mathbf{p}(x,y,z) = [-y,x,z]^T$ and $\mathbf{u} = [-y-x, x-y, z-z]^T$. This yields

$$\nabla\mathbf{u} = \begin{bmatrix} -1 & -1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \; \varepsilon_C = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \; \varepsilon_G = \mathbf{0}. \tag{4.15}$$

In this case, only Green's non linear tensor yields the correct result, its linearization cannot capture the rotation correctly. This is an important observation we will discuss in Section 4.3.

4.1.3 Stress

Now let us turn to the measurement of stress, the force per unit area. As strain, stress is represented in three dimensions by a symmetric 3 by 3 matrix or tensor

$$\sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} \tag{4.16}$$

with the following interpretation: As we saw before, at a single material point the strain depends on the direction of measurement. The same is true for the stress. Let $\mathbf{n}$ be the normal vector in the direction of measurement. Then,

$$\frac{d\mathbf{f}}{dA} = \sigma \cdot \mathbf{n}. \tag{4.17}$$

In other words, to get the force per area $\mathbf{f}/A$ with respect to a certain plane with normal $\mathbf{n}$, the stress tensor is multiplied by $\mathbf{n}$.

4.1.4 Constitutive Laws

A constitutive law relates strain to stress. Hooke's law is a special case. It states that stress and strain are linearly related. This holds for so called *Hookean* materials under small deformations. In three dimensions, Hooke's law reads

$$\sigma = \mathbf{E}\varepsilon. \tag{4.18}$$

Both stress and strain are symmetric tensors so they have only 6 independent coefficients. The quantity $\mathbf{E}$ relating the two can, thus, be expressed by a 6 by 6 dimensional matrix. For isotropic materials (with equal behavior in all directions), Hooke's law has the form

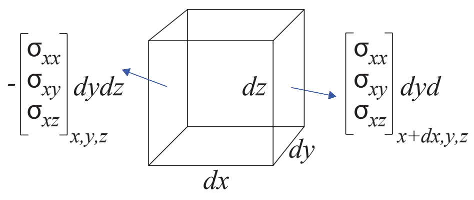

$-\begin{bmatrix}\sigma_{xx}\\ \sigma_{xy}\\ \sigma_{xz}\end{bmatrix}_{x,y,z} dydz$
$\begin{bmatrix}\sigma_{xx}\\ \sigma_{xy}\\ \sigma_{xz}\end{bmatrix}_{x+dx,y,z} dyd$
dz
dy
dx

Figure 4.2: An infinitesimal volumetric element of a deformable body. The blue arrows show the stress based force acting on the faces perpendicular to the x-axis.

$$\begin{bmatrix}\sigma_{xx}\\ \sigma_{yy}\\ \sigma_{zz}\\ \sigma_{xy}\\ \sigma_{yz}\\ \sigma_{zx}\end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)}\begin{bmatrix}1-\nu & \nu & \nu & 0 & 0 & 0\\ \nu & 1-\nu & \nu & 0 & 0 & 0\\ \nu & \nu & 1-\nu & 0 & 0 & 0\\ 0 & 0 & 0 & 1-2\nu & 0 & 0\\ 0 & 0 & 0 & 0 & 1-2\nu & 0\\ 0 & 0 & 0 & 0 & 0 & 1-2\nu\end{bmatrix}\begin{bmatrix}\varepsilon_{xx}\\ \varepsilon_{yy}\\ \varepsilon_{zz}\\ \varepsilon_{xy}\\ \varepsilon_{yz}\\ \varepsilon_{zx}\end{bmatrix}, \tag{4.19}$$

where the scalar E is Young's modulus describing the elastic stiffness and the scalar $\nu \in [0\ldots\frac{1}{2})$ Poisson's ratio, a material parameter that describes to which amount volume is conserved within the material.

4.1.5 Equation of Motion

The concepts we saw so far can be used to simulate a dynamic elastic object. First, we apply Newton's second law of motion $\mathbf{f} = m\ddot{\mathbf{p}}$ to the infinitesimal volumetric element dV at location $\mathbf{x}$ of the object (see Fig. 4.2). Since the mass of an infinitesimal element is not defined, both sides of the equation of motion are divided by the volume $dx \cdot dy \cdot dz$ of the element. This turns mass $[kg]$ into density $[kg/m^3]$ and forces $[N]$ into body forces $[N/m^3]$. We get

$$\rho\ddot{\mathbf{p}} = \mathbf{f}(\mathbf{x}), \tag{4.20}$$

where ρ is the density and $\mathbf{f}(\mathbf{x})$ the body force acting on the element at location $\mathbf{x}$. This force is the sum of external forces (e.g. gravity or collision forces) and internal forces (due to deformation). The next step is to compute the internal elastic force at the center of the element due to the stress. To get this force, we are going to sum up the forces that act on each of the six faces of the infinitesimal element. Let us first look at the faces perpendicular to the x-axis. The center of the face with normal $[-1,0,0]^T$ is located at $[x,y,z]^T$ and the one with face normal $[1,0,0]^T$ at position $[x+dx,y,z]^T$. According to Eqn. (4.17) the forces per unit area acting on these faces are

$$-\begin{bmatrix}\sigma_{xx}\\ \sigma_{xy}\\ \sigma_{xz}\end{bmatrix}_{x,y,z} \quad \text{and} \quad \begin{bmatrix}\sigma_{xx}\\ \sigma_{xy}\\ \sigma_{xz}\end{bmatrix}_{x+dx,y,z} \tag{4.21}$$

To get forces, we have to multiply by the face area $dy \cdot dz$. Finally, the body forces are the forces divided by $dV = dx \cdot dy \cdot dz$. This yields a total force for the two faces of

$$\mathbf{f} = \left(\begin{bmatrix} \sigma_{xx} \\ \sigma_{xy} \\ \sigma_{xz} \end{bmatrix}_{x+dx,y,z} - \begin{bmatrix} \sigma_{xx} \\ \sigma_{xy} \\ \sigma_{xz} \end{bmatrix}_{x,y,z} \right) / dx = \begin{bmatrix} \sigma_{xx,x} \\ \sigma_{xy,x} \\ \sigma_{xz,x} \end{bmatrix}, \tag{4.22}$$

where the comma denotes spatial derivatives. If we take the forces acting on the other faces into account as well, we arrive at the final expression for the body forces acting on an infinitesimal element due to internal stresses

$$\mathbf{f}_{stress} = \nabla \cdot \sigma = \begin{bmatrix} \sigma_{xx,x} + \sigma_{xy,y} + \sigma_{xz,z} \\ \sigma_{yx,x} + \sigma_{yy,y} + \sigma_{yz,z} \\ \sigma_{zx,x} + \sigma_{zy,y} + \sigma_{zz,z} \end{bmatrix}, \tag{4.23}$$

where, again, the comma represents a spatial derivative. We are now ready to write down the entire partial differential equation (PDE) governing dynamic elastic materials:

$$\rho \ddot{\mathbf{p}} = \nabla \cdot \sigma + \mathbf{f}_{ext}, \tag{4.24}$$

where $\mathbf{f}_{ext}$ are externally applied body forces such as gravity or collision forces. This hyperbolic PDE can be used to compute the world coordinates $\mathbf{p}$ of all material points inside the elastic body at all times which is the same as simulating the deformable object. How would one do this?

The density ρ and the external forces $\mathbf{f}_{ext}$ are known quantities. The process starts by defining the material coordinates $\mathbf{x}$ and the world coordinates $\mathbf{p}$ of all points in the body (the initial condition). Then, at each time, the displacement field is $\mathbf{u} = \mathbf{p} - \mathbf{x}$ from which the strain field ε can be computed. The stress field σ is derived from the strains via Hooke's law. Finally, σ is used to compute the acceleration of the world positions of the body $\ddot{\mathbf{p}}$ using Newton's law and these accelerations define how the positions evolve.

A linear dependency of the stresses on the strains such as in a Hookean material is called *material linearity*. A linear measure of strain such as Cauchy's linear strain tensor defined in Eqn. (4.5) is called *geometric linearity*. Only with both assumptions, material and geometric linearity, Eqn. (4.24) becomes a linear PDE. Linear PDE's are easier to solve because discretizing them via Finite Differences of Finite Elements yields linear algebraic systems. However, for large deformations or rotations, the simplification of geometric linearity causes significant visual artifacts as we saw in one of the examples above.

4.2 Finite Element Discretization

The unknown in Eq. (4.24) is $\mathbf{p}$ which is a continuous vector field. This rises the question of how one would write down the solution. A general continuous vector field cannot be described in compact form because it contains an uncountable set of vectors. A special class of vector fields can be specified by compact analytical formulas though. An example is the rotation field $\mathbf{p}(x,y,z) = [y, -x, z]$ we discussed earlier. To end up with a description in the form of a formula, one would have to solve Eq. (4.24) analytically. This is only possible for toy problems with simple domains and boundary conditions.

Fortunately there is another way to arrive at solutions that can be written down explicitly. Just *restrict* the possible solutions to a certain class of continuous vector fields that can be described by a finite number of values. From such a restriction it follows that in the general case, a solution computed in this way will only be an *approximation* of the true solution. Ideally the restricted solution is the one within the restricted class of vector fields that is closest to the true solution.

This is the core idea of the Finite Element Method (FEM). First, the domain is divided into a finite set of (polygonal) elements of finite size which cover the entire domain without overlaps. Within each element, the vector field is described by an analytical formula that depends on the positions of the vertices belonging to the element. More general parametrizations are possible but we will work with vertex positions only.

4.2.1 Constant Strain Tetrahedral Meshes

To keep things simple, we use tetrahedra as finite elements and represent the domain, i.e. the volume of the deformable body by a tetrahedral mesh. Within each tetrahedron, we use the simplest possible deformation field, a linear mapping. A constant deformation field within the element would be even simpler but a constant mapping would yield zero strain, and would therefore not be practical to simulate deformable bodies.

Let $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ be the the corners of the tetrahedron in the undeformed rest state and $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ the same corners in the deformed state. We now have to find a linear vector field $\mathbf{p}(\mathbf{x})$ that maps points within the tetrahedron in the rest state to the points within the deformed tetrahedron. Pure translation does not generate any elastic forces so we can assume that $\mathbf{x}_0 = \mathbf{0}$ and $\mathbf{p}_0 = \mathbf{0}$ without loss of generality. In the general case you have to replace $\mathbf{x}_i$ with $\mathbf{x}_i - \mathbf{x}_0$ and $\mathbf{p}_i$ with $\mathbf{p}_i - \mathbf{p}_0$ in the following formulas. Let us describe a point inside the undeformed tetrahedron by a weighted sum of the corner positions, i.e.

$$\mathbf{x} = \mathbf{x}_1 b_1 + \mathbf{x}_2 b_2 + \mathbf{x}_3 b_3 = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3]\,\mathbf{b}. \tag{4.25}$$

The transformed position $\mathbf{p}(\mathbf{x})$ will be a weighted sum of the deformed corner positions using the same weights:

$$\mathbf{p}(\mathbf{x}) = \mathbf{p}_1 b_1 + \mathbf{p}_2 b_2 + \mathbf{p}_3 b_3 = [\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3]\,\mathbf{b}. \tag{4.26}$$

Solving Eq. (4.25) for $\mathbf{b}$ and substituting into Eq. (4.26) yields

$$\mathbf{p}(\mathbf{x}) = [\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3]\,[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3]^{-1}\,\mathbf{x} = \mathbf{P}\mathbf{x} \tag{4.27}$$

This is a linear mapping with $\mathbf{P}$ a 3×3 matrix. The part $\bar{\mathbf{X}} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3]^{-1}$ is constant throughout the simulation and can be pre-computed. Because $\mathbf{p}(\mathbf{x})$ is linear, we have

$$\nabla \mathbf{p} = \mathbf{P} \text{ and } \nabla \mathbf{u} = \mathbf{P} - \mathbf{I} \tag{4.28}$$

independent of the position $\mathbf{x}$ within the tetrahedron. This means we will end up with constant strain and stress inside the tetrahedron. Using Green's stress tensor we have

$$\varepsilon = \frac{1}{2}(\nabla \mathbf{u} + [\nabla \mathbf{u}]^T + [\nabla \mathbf{u}]^T \nabla \mathbf{u}) \tag{4.29}$$

and with the assumption of a Hookean material the stress is

$$\sigma = \mathbf{E}\varepsilon \tag{4.30}$$

with **E** defined in Eq. (4.19). Multiplying the stress tensor by a normal vector yields the elastic force per area so for face $(0,1,2)$ of the tetrahedron the force is

$$\mathbf{f}_{0,1,2} = \sigma \cdot \mathbf{n}_{0,1,2} \cdot A_{0,1,2} = \sigma[(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)] \tag{4.31}$$

Finally, we distribute this force evenly among the vertices 0,1 and 2 and do the same for all faces. Equations (4.27) through (4.31) yield a recipe for computing the forces $\mathbf{f}_0, \ldots \mathbf{f}_3$ acting on the vertices of a tetrahedron based on the deformed positions $\mathbf{p}_0, \ldots \mathbf{p}_3$. As you can see, the whole process is quite simple. It is just a step-by-step of process of computing quantities based on previous ones. Also, we could easily replace Hooke's law by any non-linear stress-strain relationship without increasing the complexity of the computation significantly. The resulting forces - position relationship computed this way is highly non-linear. A simple simulation algorithm using explicit Euler integration could look like this

```
// initialization
(1)  forall vertices $i$
(2)     $\mathbf{p}_i = \mathbf{x}_i$
(3)     initialize $\mathbf{v}_i$ and $m_i$
(4)  endfor
(5)  forall tetrahedra $i = (i_0, i_1, i_2, i_3)$
(6)     $\bar{\mathbf{X}}_i = [\mathbf{x}_{i_1} - \mathbf{x}_{i_0}, \mathbf{x}_{i_2} - \mathbf{x}_{i_0}, \mathbf{x}_{i_3} - \mathbf{x}_{i_0}]^{-1}$
(7)  endfor
// simulation loop
(8)  loop
(9)     froall vertices $i$
(10)       $\mathbf{f}_i = \mathbf{f}^g + \mathbf{f}_i^{\text{coll}}$
(11)    endfor
(12)    forall tetrahedra $i = (i_0, i_1, i_2, i_3)$
(13)       $\mathbf{P} = [\mathbf{p}_{i_1} - \mathbf{p}_{i_0}, \mathbf{p}_{i_2} - \mathbf{p}_{i_0}, \mathbf{p}_{i_3} - \mathbf{p}_{i_0}] \cdot \bar{\mathbf{X}}_i$
(14)       $\nabla\mathbf{u} = \mathbf{P} - \mathbf{I}$
(15)       $\varepsilon = \frac{1}{2}(\nabla\mathbf{u} + [\nabla\mathbf{u}]^T + [\nabla\mathbf{u}]^T\nabla\mathbf{u})$
(16)       $\sigma = \mathbf{E}\varepsilon$
(17)       forall faces $j = (j_0, j_1, j_2)$ of tetrahedron $i$
(18)          $\mathbf{f}_{\text{face}} = \sigma[(\mathbf{p}_{j_1} - \mathbf{p}_{j_0}) \times (\mathbf{p}_{j_2} - \mathbf{p}_{j_0})]$
(19)          $\mathbf{f}_{j_0} \leftarrow \mathbf{f}_{j_0} + \frac{1}{3}\mathbf{f}_{\text{face}}$
(20)          $\mathbf{f}_{j_1} \leftarrow \mathbf{f}_{j_1} + \frac{1}{3}\mathbf{f}_{\text{face}}$
(21)          $\mathbf{f}_{j_1} \leftarrow \mathbf{f}_{j_1} + \frac{1}{3}\mathbf{f}_{\text{face}}$
(22)       endfor
(23)    endfor
(24)    forall vertices $i$
(25)       $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\ \mathbf{f}_i / m_i$
(26)       $\mathbf{p}_i \leftarrow \mathbf{p}_i + \Delta t\ \mathbf{v}_i$
(27)    endfor
```

```
(28)     display the system every $n^{th}$ time
(29) endloop
```

This algorithm is very similar to the algorithm for simulating mass-spring systems. We basically replaced the one dimensional springs connecting pairs of mass points by three dimensional hyper-springs connecting four mass points. The computation of the forces acting at the adjacent vertices of such a hyper-spring is more expensive than computing spring forces but it does not make the algorithm more difficult to understand or to implement. Also, replacing the explicit Euler scheme by a Verlet or Runge-Kutta integrator would be straight forward. All this comes from the fact, that all force evaluations are explicit.

For unconditional stability and the use of a global solver, implicit integration is needed. In this case, things get more involved. Linearizing the spring force in Eq. (3.33) was not too hard. Doing the same for the FEM-based forces is a bit more complicated. You have to work through all the steps given in Eq. (4.27) through Eq. (4.31). The resulting formulas get more complex and more computationally intensive to evaluate, especially for general stress-strain relationships. Remember, this evaluation has to be done at every time step. For a spring force connecting points i and j we derived local tangent stiffness matrices $\mathbf{K}_{i,i}$, $\mathbf{K}_{i,j}$, $\mathbf{K}_{j,i}$ and $\mathbf{K}_{j,j}$. In the case of a tetrahedron, the tangent stiffness matrix $\mathbf{K}$ is 12×12 dimensional. It contains 4×4 submatrices $\mathbf{K}_{i,j}$, where $\mathbf{K}_{i,j}$ describes the interaction between vertex i and vertex j. Because each vertex in the tetrahedron influences all others, the stiffness matrix of the tetrahedron is dense. With the stiffness matrix, the tetrahedral forces can be approximated as

$$\mathbf{f}(\mathbf{p} + \Delta t \mathbf{v}) = \mathbf{f}(\mathbf{p}) + \mathbf{K}|_{\mathbf{x}} (\Delta t \mathbf{v}) + O(\Delta t^2), \tag{4.32}$$

where all vectors are 12 dimensional containing the 3 components of the 4 adjacent vertices of the tetrahedron, i.e.

$$\mathbf{p} = [\mathbf{p}_1^T, \mathbf{p}_2^T, \mathbf{p}_3^T, \mathbf{p}_4^T]^T \tag{4.33}$$

$$\mathbf{v} = [\mathbf{v}_1^T, \mathbf{v}_2^T, \mathbf{v}_3^T, \mathbf{v}_4^T]^T \text{ and} \tag{4.34}$$

$$\mathbf{f}(\mathbf{p}) = [\mathbf{f}_1(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4)^T, \mathbf{f}_2(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4)^T, \mathbf{f}_3(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4)^T, \mathbf{f}_4(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4)^T]^T. \tag{4.35}$$

4.2.2 Linear FEM

If $\mathbf{K}$ was constant at all times, the simulation would obviously be much faster and more suitable for implicit integration in real time. Let us see how we can make it constant. First we have to decide where we want to evaluate $\mathbf{K}$. The most natural choice is to evaluate it at the equilibrium configuration so $\mathbf{K}_{\text{const}} = \mathbf{K}|_x$. Every other location would be biased in some way. This choice has a further advantage. The first term on the right hand side of Eq. (4.32) cancels out because the forces at the equilibrium configuration are zero and we get

$$\mathbf{f}(\mathbf{x}+\Delta\mathbf{p}) = \mathbf{f}(\mathbf{x}) + \mathbf{K}|_{\mathbf{x}}(\Delta\mathbf{p}) \qquad +O(|\Delta\mathbf{p}|^2) \tag{4.36}$$
$$\mathbf{f}(\mathbf{x}+(\mathbf{p}-\mathbf{x})) = \mathbf{K}|_{\mathbf{x}}(\mathbf{p}-\mathbf{x}) \qquad +O(|\mathbf{p}-\mathbf{x}|^2) \tag{4.37}$$
$$\mathbf{f}(\mathbf{p}) = \mathbf{K}|_{\mathbf{x}}(\mathbf{p}-\mathbf{x}) \qquad +O(|\mathbf{p}-\mathbf{x}|^2). \tag{4.38}$$
$$\tag{4.39}$$

By omitting the higher order terms and writing **K** for $\mathbf{K}|_{\mathbf{x}}$ from now on we have a very simple equation to compute the forces acting on the vertices of the tetrahedron given the displacements of the vertices, namely

$$\mathbf{f}(\mathbf{p}) = \mathbf{K}(\mathbf{p}-\mathbf{x}). \tag{4.40}$$

This approximation is only valid close to the equilibrium because that is where the forces were linearized. There are many applications in which the displacements remain small, for instance when the stress distribution in a bridge or building have to be analyzed. In general, if the deformable body does not undergo large deformations it is reasonable to use this approximation.

But how do we compute **K** for a tetrahedron? We could linearize Eq. (4.27) through Eq. (4.31) and evaluate what we get at $\mathbf{p} = \mathbf{x}$. There is a simpler more intuitive way to do it though. Three restrictions make the forces linear to begin with so linearization is not necessary at all. For this we have to

- use Cauchy strain $\varepsilon = \frac{1}{2}(\nabla\mathbf{u} + [\nabla\mathbf{u}]^T)$,
- assume a linear stress-strain relationship, i.e. a Hookean material with $\sigma = \mathbf{E}\varepsilon$
- and use the original positions of the vertices in Eq. (4.31) for the computation of the face normals and face areas.

We will not go through the whole process of the derivation. Here is the final result, the recipe to compute the stiffness matrix of a tetrahedron. First you compute the four auxiliary vectors $\mathbf{y}_0$, $\mathbf{y}_1$,$\mathbf{y}_2$ and $\mathbf{y}_3$ as

$$\begin{bmatrix} \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \mathbf{y}_3^T \end{bmatrix} = \mathbf{X}^{-1} = [\mathbf{x}_1-\mathbf{x}_0, \mathbf{x}_2-\mathbf{x}_0, \mathbf{x}_3-\mathbf{x}_0]^{-1} \tag{4.41}$$
$$\mathbf{y}_0 = -\mathbf{y}_1 - \mathbf{y}_2 - \mathbf{y}_3 \tag{4.42}$$

With these, the 3×3 sub-matrix of **K** connecting vertex i and j $(i, j \in 1\ldots 4)$ can be directly computed as

$$\mathbf{K}_{i,j} = \begin{bmatrix} \mathbf{y}_{i,x} & 0 & 0 \\ 0 & \mathbf{y}_{i,y} & 0 \\ 0 & 0 & \mathbf{y}_{i,z} \end{bmatrix} \begin{bmatrix} a & b & b \\ b & a & b \\ b & b & a \end{bmatrix} \begin{bmatrix} \mathbf{y}_{j,x} & 0 & 0 \\ 0 & \mathbf{y}_{j,y} & 0 \\ 0 & 0 & \mathbf{y}_{j,z} \end{bmatrix} \tag{4.43}$$
$$+ \begin{bmatrix} \mathbf{y}_{i,y} & 0 & \mathbf{y}_{i,z} \\ \mathbf{y}_{i,x} & \mathbf{y}_{i,z} & 0 \\ 0 & \mathbf{y}_{i,y} & \mathbf{y}_{i,x} \end{bmatrix} \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} \mathbf{y}_{j,y} & \mathbf{y}_{j,x} & 0 \\ 0 & \mathbf{y}_{j,z} & \mathbf{y}_{j,y} \\ \mathbf{y}_{j,z} & 0 & \mathbf{y}_{j,x}, \end{bmatrix}, \tag{4.44}$$

where

$$a = VE\frac{1-\nu}{(1+\nu)(1-2\nu)}, \tag{4.45}$$

$$b = VE\frac{\nu}{(1+\nu)(1-2\nu)}, \tag{4.46}$$

$$c = VE\frac{1-2\nu}{(1+\nu)(1-2\nu)}, \tag{4.47}$$

$V = \det(\mathbf{X})$, E is Youngs Modulus and ν the Poisson ratio. The center matrix on line (4.43) is the upper left sub-matrix of $\mathbf{E}$ given in Eq. (4.19) and the matrices to the left and right of it represent normal strain. The center matrix on line (4.44) is the lower right sub-matrix of $\mathbf{E}$ and the matrices to the left and right correspond to shear strain, where the subscripts after the comma are the spatial components of the vectors, not their derivatives.

4.3 Warped Stiffness

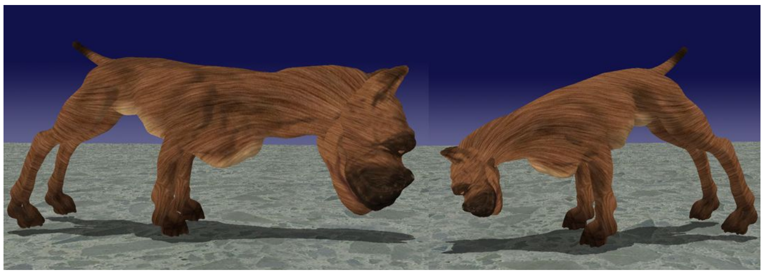


Figure 4.3: The pitbull with its inflated head (left) shows the artifact of linear FEM under large rotational deformations. The correct deformation is shown on the right.

In the framework of linear FEM discussed in the previous section, the stiffness matrices of all tetrahedra are constant, i.e. depend only on the rest configuration of the vertices and can be precomputed. This makes the simulation fast and allows real-time simulation of high resolution tetrahedra meshes. However, we have seen in Section 4.1.2 that Cauchy strain cannot capture rotational deformations correctly (see Fig. 4.3). In graphics, this is a problem, because only large deformations and free rotational motion yield visually interesting effects.

The technique of *stiffness warping* which we will discuss in this section solves this problem by explicitly extracting the rotational part of the deformation. The method removes the visual artifacts under large rotational deformations and allows to use the constant stiffness matrices of linear FEM. Let us see how this works.

4.3.1 Extraction of Rotation

If we write $\mathbf{K}_e$ for the stiffness matrix of a tetrahedral element e to distinguish it from the stiffness matrix $\mathbf{K}$ of the entire mesh, the computation of the linear forces acting on the four vertices of the element are

$$\mathbf{f} = \mathbf{K}_e \cdot (\mathbf{p} - \mathbf{x}). \tag{4.48}$$

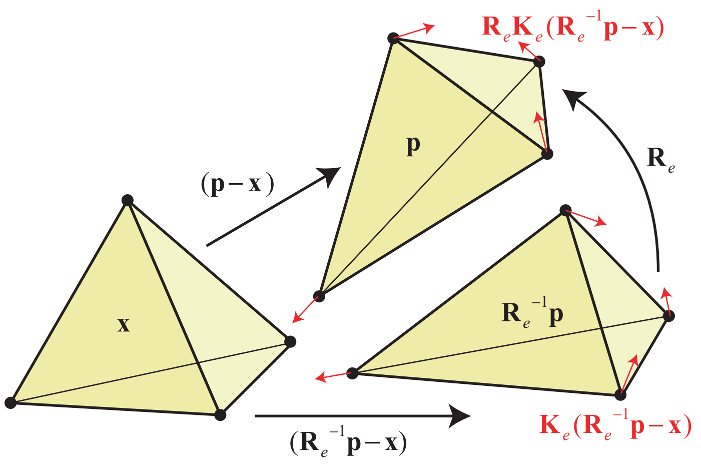

$\mathbf{R}_e\mathbf{K}_e(\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x})$
$\mathbf{p}$
$(\mathbf{p}-\mathbf{x})$
$\mathbf{R}_e$
$\mathbf{x}$
$\mathbf{R}_e^{-1}\mathbf{p}$
$(\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x})$
$\mathbf{K}_e(\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x})$

Figure 4.4: To compute the elastic forces acting at the vertices of a tetrahedron, its deformed coordinates $\mathbf{p}$ are rotated back to an unrotated frame $\mathbf{R}_e^{-1}\mathbf{p}$. There, the displacements $\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x}$ are multiplied with the stiffness matrix yielding the forces $\mathbf{K}_e(\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x})$ that are finally rotated back to the frame of the deformed tetrahedron by multiplying them with $\mathbf{R}_e$.

Now comes the trick. Let us assume that we know the rotational part $\mathbf{R}_e$ of the deformation of the tetrahedron. We use this matrix to rotate the world coordinates $\mathbf{p}$ of the vertices back into an unrotated configuration $\mathbf{R}_e^{-1}\mathbf{p}$. For this to work, $\mathbf{R}_e$ needs to be a 12×12 matrix containing four identical 3×3 rotation matrices along its diagonal. After the transformation, the displacements are $\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x}$. The linear forces in this unrotated frame are $\mathbf{K}_e(\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x})$. Finally, the forces have to be moved back into the rotated current configuration of the tetrahedron yielding

$$\mathbf{f}_{\text{warped}} = \mathbf{R}_e\mathbf{K}_e(\mathbf{R}_e^{-1}\mathbf{p}-\mathbf{x}). \tag{4.49}$$

This computation is fast. Also remember that for a rotation matrix $\mathbf{R}_e^{-1} = \mathbf{R}_e^T$. To use these force in an implicit Euler scheme we substitute them into Eq. (3.23) and Eq. (3.24) which yields

$$\begin{aligned}\mathbf{M}\mathbf{v}^{t+1} &= \mathbf{M}\mathbf{v}^t + \Delta t\left(\mathbf{R}_e\mathbf{K}_e(\mathbf{R}_e^T\mathbf{p}^{t+1}-\mathbf{x}) + \mathbf{f}_{\text{ext}}\right) &(4.50)\\ \mathbf{p}^{t+1} &= \mathbf{p}^t + \Delta t\mathbf{v}^{t+1}. &(4.51)\end{aligned}$$

Again, we substitute the second line into the first and get

$$\begin{aligned}\mathbf{M}\mathbf{v}^{t+1} &= \mathbf{M}\mathbf{v}^t + \Delta t\left(\mathbf{R}_e\mathbf{K}_e(\mathbf{R}_e^T(\mathbf{p}^t+\Delta t\mathbf{v}^{t+1})-\mathbf{x}) + \mathbf{f}_{\text{ext}}\right) &(4.52)\\ (\mathbf{M}-\Delta t^2\mathbf{R}_e\mathbf{K}_e\mathbf{R}_e^T)\mathbf{v}^{t+1} &= \mathbf{M}\mathbf{v}^t + \Delta t\left(\mathbf{R}_e\mathbf{K}_e(\mathbf{R}_e^T\mathbf{p}^t-\mathbf{x}) + \mathbf{f}_{\text{ext}}\right) &(4.53)\\ (\mathbf{M}-\Delta t^2\mathbf{K}'_e)\mathbf{v}^{t+1} &= \mathbf{M}\mathbf{v}^t + \Delta t\left(\mathbf{K}'_e\mathbf{p}^t - \mathbf{f}_0 + \mathbf{f}_{\text{ext}}\right), &(4.54)\end{aligned}$$

where $\mathbf{K}'_e = \mathbf{R}_e\mathbf{K}_e\mathbf{R}_e^T$ and $\mathbf{f}_0 = \mathbf{R}_e\mathbf{K}_e\mathbf{x}$. The last line is the linear equation for the new velocities. All quantities except $\mathbf{v}^{t+1}$ are known at the current time step and form a linear system for the new velocities. This is the equation for a single tetrahedron. The one for the entire mesh is

$$(\mathbf{M} - \Delta t^2 \mathbf{K}')\mathbf{v}^{t+1} = \mathbf{M}\mathbf{v}^t + \Delta t \left(\mathbf{K}'\mathbf{p}^t - \mathbf{f}_0 + \mathbf{f}_{\text{ext}}\right). \tag{4.55}$$

Constructing the global stiffness matrix $\mathbf{K}$ from the element matrices $\mathbf{K}_e$ and the global right hand side force offset is called *element assembly*. This process can be written as

$$\mathbf{K} = \sum_e \mathbf{R}_e \mathbf{K}_e \mathbf{R}_e^T \tag{4.56}$$

$$\mathbf{f}_0 = \sum_e \mathbf{R}_e \mathbf{K}_e \mathbf{x}. \tag{4.57}$$

These are not valid equations as they stand here. On the left we have $3n$ dimensional quantities, n being the number of mesh vertices, while on the right the quantities are 12 dimensional due to the four vertices from a tetrahedral element. For the formulas to be valid, the quantities on the right are made $3n$ dimensional as well by placing in the sub matrices and vectors corresponding to the vertices of the tetrahedron at the right spots w.r.t. to the global mesh numbering and filling the rest with zeros. The remaining question is how to get the rotations $\mathbf{R}_e$ of the tetrahedra.

4.3.2 Determining Element Rotation

From Eq. (4.27) we know that the non-translational part of the mapping from the original configuration $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ to the current configuration $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ of a tetrahedron is the 3×3 dimensional matrix

$$\mathbf{A} = [\mathbf{p}_1 - \mathbf{p}_0, \mathbf{p}_2 - \mathbf{p}_0, \mathbf{p}_3 - \mathbf{p}_0]\,[\mathbf{x}_1 - \mathbf{x}_0, \mathbf{x}_2 - \mathbf{x}_0, \mathbf{x}_3 - \mathbf{x}_0]^{-1}. \tag{4.58}$$

The rotational part of $\mathbf{A}$ is the rotation we are looking for. There are several ways to construct a rotation matrix from $\mathbf{A}$, also called *orthonormalization* of $\mathbf{A}$. The simples is the *Gram-Schmidt* method. Let $\mathbf{A} = [\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2]$ with $\mathbf{a}_i$ the column vectors of $\mathbf{A}$ and the axes of the transformation. We need to make sure that they all have unit length and that they are perpendicular to each other. The *Gram-Schmidt* method computes the three columns of the rotation matrix

$$\mathbf{r}_0 = \frac{\mathbf{a}_0}{|\mathbf{a}_0|} \tag{4.59}$$

$$\mathbf{r}_1 = \frac{\mathbf{a}_1 - \mathbf{r}_0 \cdot \mathbf{a}_1}{|\mathbf{a}_1 - \mathbf{r}_0 \cdot \mathbf{a}_1|} \tag{4.60}$$

$$\mathbf{r}_2 = \mathbf{r}_0 \times \mathbf{r}_1. \tag{4.61}$$

The third line is non-*Gram-Schmidt* but simpler and makes sure that the resulting rotation matrix $\mathbf{R} = [\mathbf{r}_0, \mathbf{r}_1, \mathbf{r}_2]$ is right handed. This procedure is dependent on the order in which the axes are processed. The first axis of the rotation will always be aligned with the first axis of the transformation for instance. This fact might yield some artifacts. Typically, this is not a big issue. You just have to make sure that you keep the order the same throughout the simulation.

A mathematically more correct and unbiased way to extract the rotations is to find the rotation matrix $\mathbf{R}$ that is closest to $\mathbf{A}$ in the least squares sense. It is a well known fact from linear algebra that each matrix can be factored into a rotation matrix and a symmetric matrix, i.e.

$$\mathbf{A} = \mathbf{R}\mathbf{S}. \tag{4.62}$$

This factorization is called *polar decomposition* and should be used for unbiased and more accurate results.

Chapter 5

Position Based Dynamics

Matthias Müller

Figure 5.1: Various deformable objects simulated using the Position Based Dynamics approach.

The most popular approaches for the simulation of dynamic systems in computer graphics are force based. Internal and external forces are accumulated from which accelerations are computed based on Newton's second law of motion. A time integration method is then used to update the velocities and finally the positions of the object. A few simulation methods (most rigid body simulators) use impulse based dynamics and directly manipulate velocities. In this chapter we present an approach which omits the velocity layer as well and immediately works on the positions. The main advantage of a position based approach is its controllability. Overshooting problems of explicit integration schemes in force based systems can be avoided. In addition, collision constraints can be handled easily and penetrations can be resolved completely by projecting points to valid locations.

5.1 Position Based Simulation

The objects to be simulated are represented by a set of N particles and a set of M constraints. Each particle i has three attributes, namely

m_i	mass
$\mathbf{x}_i$	position
$\mathbf{v}_i$	velocity

A constraint j is defined by the five attributes

n_j	cardinality
$C_j : \mathbb{R}^{3n_j} \to \mathbb{R}$	scalar constraint function
$\{i_1, \ldots i_{n_j}\}$, $i_k \in [1, \ldots N]$	set of indices
$k_j \in [0 \ldots 1]$	stiffness parameter
unilateral or *bilateral*	type

.

Constraint j with type *bilateral* is satisfied if $C_j(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_{n_j}}) = 0$. If its type is *unilateral* then it is satisfied if $C_j(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_{n_j}}) \geq 0$. The stiffness parameter k_j defines the strength of the constraint in a range from zero to one.

Given this data and a time step Δt, the simulation proceeds as follows:

```
(1)  forall particles $i$
(2)    initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i = \mathbf{v}_i^0, w_i = 1/m_i$
(3)  endfor
(4)  loop
(5)    forall particles $i$ do $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
(6)    forall particles $i$ do $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
(7)    forall particles $i$ do generateCollisionConstraints($\mathbf{x}_i \to \mathbf{p}_i$)
(8)    loop solverIterations times
(9)      projectConstraints($C_1, \ldots, C_{M+M_{coll}}, \mathbf{p}_1, \ldots, \mathbf{p}_N$)
(10)   endloop
(11)   forall particles $i$
(12)     $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$
(13)     $\mathbf{x}_i \leftarrow \mathbf{p}_i$
(14)   endfor
(15) endloop
```

Since the algorithm simulates a system which is second order in time, both, the positions and the velocities of the particles need to be specified in (1)-(3) before the simulation loop starts. Lines (5)-(6) perform a simple explicit forward Euler integration step on the velocities and the positions. The new locations $\mathbf{p}_i$ are not assigned to the positions directly but are only used as predictions. Non-permanent external constraints such as collision constraints are generated at the beginning of each time step from scratch in line (7). Here the original and the predicted positions are used in order to perform continuous collision detection. The solver (8)-(10) then iteratively corrects the predicted positions such that they satisfy the M_{coll} external as well as the M internal constraints. Finally the corrected positions $\mathbf{p}_i$ are used to update the positions *and* the velocities. It is essential here to update the velocities along with the positions. If this is not done, the simulation does not produce the correct behavior of a second order system. As you can see, the integration scheme used here is very similar to the Verlet method described in Eq. (3.19) and Eq. (3.20).

5.2 The System to be solved

The goal of the solver step (8)-(10) is to correct the predicted positions $\mathbf{p}_i$ of the particles such that they satisfy all constraints. The problem that needs to be solved comprises of a set of M equations for the $3N$ unknown position components, where M is now the total number of constraints. This system does not need to be symmetric. If $M > 3N$ (M <3N) the system is over-determined (under-determined). In addition to the asymmetry, the equations are in general non-linear. The function of a simple distance constraint $C(\mathbf{p}_1, \mathbf{p}_2) = (\mathbf{p}_1 - \mathbf{p}_2)^2 - d^2$ yields a non-linear equation. What complicates things even further is the fact that collisions produce inequalities rather than equalities. Solving a non-symmetric, non-linear system with equalities and inequalities is a tough problem.

Let $\mathbf{p}$ be the concatenation $[\mathbf{p}_1^T, \dots, \mathbf{p}_N^T]^T$ and let all the constraint functions C_j take the concatenated vector $\mathbf{p}$ as input while only using the subset of coordinates they are defined for. We can now write the system to be solved as

$$\begin{aligned} C_1(\mathbf{p}) &\succ 0 \\ &\dots \\ C_M(\mathbf{p}) &\succ 0, \end{aligned}$$

where the symbol $\succ$ denotes either $=$ or $\geq$. As we saw in Section 3.5.1, Newton-Raphson iteration is a method to solve non-linear symmetric systems with equalities only. The process starts with a first guess of a solution. Each constraint function is then linearized in the neighborhood of the current solution using

$$C(\mathbf{p} + \Delta\mathbf{p}) = C(\mathbf{p}) + \nabla_{\mathbf{p}} C(\mathbf{p}) \cdot \Delta\mathbf{p} + O(|\Delta\mathbf{p}|^2) = 0. \tag{5.1}$$

This yields a *linear* system for the global correction vector $\Delta\mathbf{p}$

$$\begin{aligned} \nabla_{\mathbf{p}} C_1(\mathbf{p}) \cdot \Delta\mathbf{p} &= -C_1(\mathbf{p}) \\ &\dots \\ \nabla_{\mathbf{p}} C_M(\mathbf{p}) \cdot \Delta\mathbf{p} &= -C_M(\mathbf{p}), \end{aligned}$$

where $\nabla_{\mathbf{p}} C_j(\mathbf{p})$ is the $1 \times N$ dimensional vector containing the derivatives of the function C_j w.r.t. all its parameters, i.e. the N components of $\mathbf{p}$. It is also the jth row of the linear system. Both, the rows $\nabla_{\mathbf{p}} C_j(\mathbf{p})$ and the right hand side scalars $-C_j(\mathbf{p})$ are constant because they are *evaluated* at the location $\mathbf{p}$ before the system is solved. When $M = 3N$ and only equalities are present, the system can be solved by any linear solver, e.g. PCG. Once it is solved for $\Delta\mathbf{p}$ the current solution is updated as $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$. A new linear system is generated by evaluating $\nabla_{\mathbf{p}} C_j(\mathbf{p})$ and $-C_j(\mathbf{p})$ at the new location after which the process repeats.

If $M \neq 3M$ the resulting matrix of the linear system is non-symmetric and not invertible. [GHF$^+$07] solve this problem by using the pseudo-inverse of the system matrix which yields the best solution in the least-squares sense. Still, handling inequalities is not possible directly.

5.3 The Non-Linear Gauss-Seidel Solver

In the position based dynamics appraoch, non-linear Gauss-Seidel is used. It solves each constraint equation separately. Each constraint yields a single scalar equation $C(\mathbf{p}) \succ 0$ for all the particle positions associated with it. The subsystem is therefore highly under-determined. PBD solves this problem as follows. Again, given $\mathbf{p}$ we want to find a correction $\Delta\mathbf{p}$ such that $C(\mathbf{p}+\Delta\mathbf{p}) = 0$. It is important to notice that PBD also linearizes the constraint function but individually for each constraint. The constraint equation is approximated by

$$C(\mathbf{p}+\Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla_{\mathbf{p}} C(\mathbf{p}) \cdot \Delta\mathbf{p} = 0. \tag{5.2}$$

The problem of the system being under-determined is solved by restricting $\Delta\mathbf{p}$ to be in the direction of $\nabla_{\mathbf{p}} C$ which conserves the linear and angular momenta. This means that only one scalar λ - a Lagrange multiplier - has to be found such that the correction

$$\Delta\mathbf{p} = \lambda \nabla_{\mathbf{p}} C(\mathbf{p}). \tag{5.3}$$

solves (5.2). This yields the following formula for the correction vector of a single particle i

$$\Delta\mathbf{p}_i = -s\, w_i \nabla_{\mathbf{p}_i} C(\mathbf{p}), \tag{5.4}$$

where

$$s = \frac{C(\mathbf{p})}{\sum_j w_j |\nabla_{\mathbf{p}_j} C(\mathbf{p})|^2} \tag{5.5}$$

and $w_i = 1/m_i$.

As mentioned above, this solver linearizes the constraint functions. However, in contrast to the Newton-Raphson method, the linearization happens individually *per constraint*. Solving the linearized constraint function of a single distance constraint for instance yields the correct result in a single step. Because the positions are immediately updated after a constraint is processed, these updates will influence the linearization of the next constraint because the linearization depends on the actual positions. Asymmetry poses no problem because each constraint produces one scalar equation for one unknown Lagrange multiplier λ. Inequalities are handled trivially by first checking whether $C(\mathbf{p}) \geq 0$. If this is the case, the constraint is simply skipped.

We have not considered the stiffness k of the constraint so far. There are several ways of incorporating the it. The simplest variant is to multiply the corrections $\Delta\mathbf{p}$ by $k \in [0 \ldots 1]$.

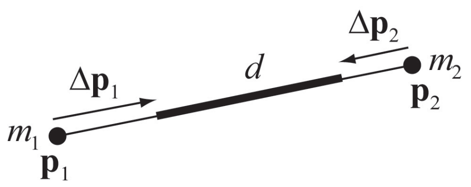


Figure 5.2: Projection of the constraint $C(\mathbf{p}_1,\mathbf{p}_2) = |\mathbf{p}_1 - \mathbf{p}_2| - d$. The corrections $\Delta\mathbf{p}_i$ are weighted according to the inverse masses $w_i = 1/m_i$.

However, for multiple iteration loops of the solver, the effect of k is non-linear. The remaining error for a single distance constraint after n_s solver iterations is $\Delta\mathbf{p}(1-k)^{n_s}$. To get a linear relationship we multiply the corrections not by k directly but by $k' = 1-(1-k)^{1/n_s}$. With this transformation the error becomes $\Delta\mathbf{p}(1-k')^{n_s} = \Delta\mathbf{p}(1-k)$ and, thus, becomes linearly dependent on k and independent of n_s as desired. However, the resulting material stiffness is still dependent on the time step of the simulation. Real time environments typically use fixed time steps in which case this dependency is not problematic.

5.4 Constraint Examples

5.4.1 Stretching

To give an example, let us consider the distance constraint function $C(\mathbf{p}_1,\mathbf{p}_2) = |\mathbf{p}_1 - \mathbf{p}_2| - d$. The derivative with respect to the points are $\nabla_{\mathbf{p}_1} C(\mathbf{p}_1,\mathbf{p}_2) = \mathbf{n}$ and $\nabla_{\mathbf{p}_2} C(\mathbf{p}_1,\mathbf{p}_2) = -\mathbf{n}$ with $\mathbf{n} = \frac{\mathbf{p}_1-\mathbf{p}_2}{|\mathbf{p}_1-\mathbf{p}_2|}$. The scaling factor s is, thus, $s = \frac{|\mathbf{p}_1-\mathbf{p}_2|-d}{1+1}$ and the final corrections

$$\Delta\mathbf{p}_1 = -\frac{w_1}{w_1+w_2}(|\mathbf{p}_1-\mathbf{p}_2|-d)\frac{\mathbf{p}_1-\mathbf{p}_2}{|\mathbf{p}_1-\mathbf{p}_2|} \tag{5.6}$$

$$\Delta\mathbf{p}_2 = +\frac{w_2}{w_1+w_2}(|\mathbf{p}_1-\mathbf{p}_2|-d)\frac{\mathbf{p}_1-\mathbf{p}_2}{|\mathbf{p}_1-\mathbf{p}_2|} \tag{5.7}$$

which are the formulas proposed in [Jak01] for the projection of distance constraints (see Figure 5.2). They pop up as a special case of the general constraint projection method.

5.4.2 Bending

In cloth simulation it is important to simulate bending in addition to stretching resistance. To this end, for each pair of adjacent triangles $(\mathbf{p}_1,\mathbf{p}_3,\mathbf{p}_2)$ and $(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_4)$ a bilateral bending constraint is added with constraint function

$$C_{bend}(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4) =$$
$$\operatorname{acos}\left(\frac{(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_3-\mathbf{p}_1)}{|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_3-\mathbf{p}_1)|}\cdot\frac{(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_4-\mathbf{p}_1)}{|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_4-\mathbf{p}_1)|}\right) - \varphi_0,$$

stiffness k_{bend}. The scalar φ_0 is the initial dihedral angle between the two triangles and k_{bend} is a global user parameter defining the bending stiffness of the cloth (see Figure 5.3). The advantage of this bending term over adding a distance constraint between points $\mathbf{p}_3$ and $\mathbf{p}_4$

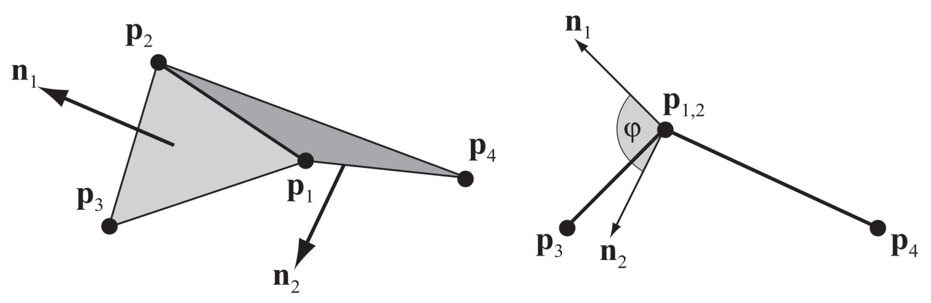


Figure 5.3: For bending resistance, the constraint function $C(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4) = \arccos(\mathbf{n}_1 \cdot \mathbf{n}_2) - \varphi_0$ is used. The actual dihedral angle φ is measure as the angle between the normals of the two triangles.

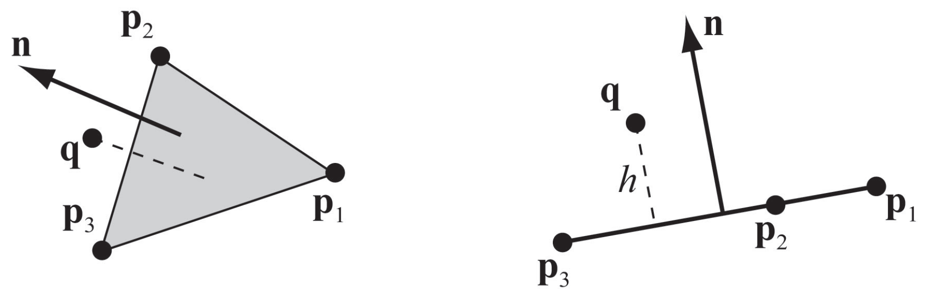


Figure 5.4: Constraint function $C(\mathbf{q},\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3) = (\mathbf{q}-\mathbf{p}_1)\cdot\mathbf{n} - h$ makes sure that $\mathbf{q}$ stays above the triangle $\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3$ by the the cloth thickness h.

or over the bending term proposed by [GHDS03] is that it is *independent of stretching*. This is because the term is independent of edge lengths.

5.4.3 Triangle Collisions

The handling of self collisions within cloth can be handled by an additional unilateral constraint. For vertex $\mathbf{q}$ moving through a triangle $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, the constraint function reads

$$C(\mathbf{q},\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3) = (\mathbf{q}-\mathbf{p}_1)\cdot\frac{(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_3-\mathbf{p}_1)}{|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_3-\mathbf{p}_1)|} - h, \tag{5.8}$$

where h is the cloth thickness. If the vertex enters from below with respect to the triangle normal, the constraint function has to be

$$C(\mathbf{q},\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3) = (\mathbf{q}-\mathbf{p}_1)\cdot\frac{(\mathbf{p}_3-\mathbf{p}_1)\times(\mathbf{p}_2-\mathbf{p}_1)}{|(\mathbf{p}_3-\mathbf{p}_1)\times(\mathbf{p}_2-\mathbf{p}_1)|} - h \tag{5.9}$$

5.4.4 Volume Conservation

For tetrahedral meshes it useful to have a constraint that conserves the volume of single tetrahedra. Such a constraint has the form

$$C(\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4) = \frac{1}{6}\left((\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_3-\mathbf{p}_1)\right)\cdot(\mathbf{p}_4-\mathbf{p}_1) - V_0, \tag{5.10}$$

where $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ and $\mathbf{p}_4$ are the four corners of the tetrahedron and V_0 its rest volume.

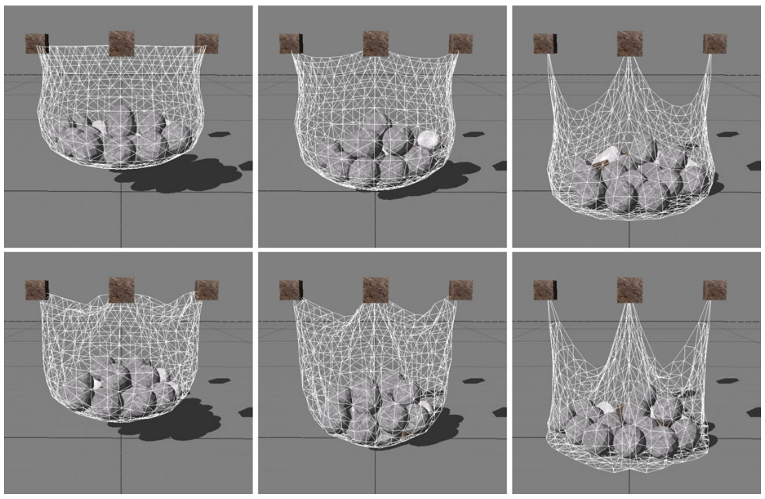


Figure 5.5: The image shows a mesh that is simulated using stretching and bending constraints. The top row shows $(k_{\text{stretching}}, k_{\text{bending}})$ = $(1,1)$, $(\frac{1}{2},1)$ and $(\frac{1}{100},1)$. The bottom row shows $(k_{\text{stretching}}, k_{\text{bending}})$ = $(1,0)$, $(\frac{1}{2},0)$ and $(\frac{1}{100},0)$.

5.4.5 Shape Matching

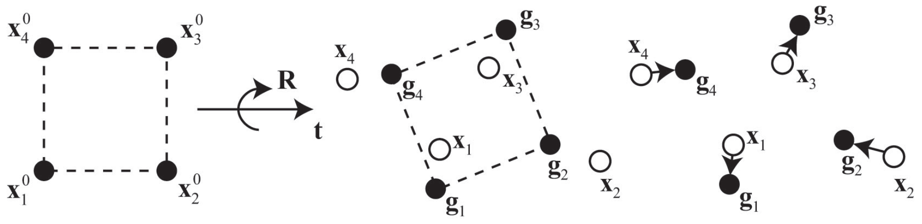


Figure 5.6: First, the original shape $\mathbf{x}_i^0$ is matched to the deformed shape $\mathbf{x}_i$. Then, the deformed points $\mathbf{x}_i$ are pulled towards the matched shape $\mathbf{g}_i$.

Shape Matching can be used to formulate a constraint for the entire set of particles. No connectivity is needed in this case. In correspondence to rest lengths, rest angles and rest volumes in the previous constraints, this constraint needs the original positions of all the particles $\mathbf{x}_i^0$. We do not formulate the constraint using a function $C(\ldots)$ but describe the projection of the points due to this constraint directly.

In order to find the projected positions, a shape matching problem with a priori known correspondences has to be solved: Given two sets of points $\mathbf{x}_i^0$ and $\mathbf{p}_i$, find the rotation matrix $\mathbf{R}$ and the translation vectors $\mathbf{t}$ and $\mathbf{t}_0$ which minimize

$$\sum_i w_i(\mathbf{R}(\mathbf{x}_i^0 - \mathbf{t}_0) + \mathbf{t} - \mathbf{p}_i)^2, \tag{5.11}$$

where the w_i are weights of individual points. The natural choice for the weights is $w_i = m_i$. The optimal translation vectors turn out to be the center of mass of the initial shape and the center of mass of the actual shape, i. e.

$$\mathbf{t}_0 = \mathbf{x}_{\text{cm}}^0 = \frac{\sum_i m_i \mathbf{x}_i^0}{\sum_i m_i}, \ \mathbf{t} = \mathbf{x}_{\text{cm}} = \frac{\sum_i m_i \mathbf{p}_i}{\sum_i m_i}, \tag{5.12}$$

which is physically plausible. Finding the optimal rotation is slightly more involved. Let us define the relative locations $\mathbf{q}_i = \mathbf{x}_i^0 - \mathbf{x}_{\text{cm}}^0$ and $\mathbf{p}_i = \mathbf{x}_i - \mathbf{x}_{\text{cm}}$ of points with respect to their center of mass and let us relax the problem of finding the optimal rotation matrix $\mathbf{R}$ to finding the optimal linear transformation $\mathbf{A}$. Now, the term to be minimized is $\sum_i m_i(\mathbf{A}\mathbf{q}_i - \mathbf{p}_i)^2$. Setting the derivatives with respect to all coefficients of $\mathbf{A}$ to zero yields the optimal transformation

$$\mathbf{A} = (\sum_i m_i \mathbf{p}_i \mathbf{q}_i^T)(\sum_i m_i \mathbf{q}_i \mathbf{q}_i^T)^{-1} = \mathbf{A}_{pq}\mathbf{A}_{qq}. \tag{5.13}$$

The second term $\mathbf{A}_{qq}$ is a symmetric matrix and, thus, contains only scaling but no rotation. Therefore, the optimal rotation $\mathbf{R}$ is the rotational part of $\mathbf{A}_{pq}$ which can be found via a polar decomposition $\mathbf{A}_{pq} = \mathbf{R}\mathbf{S}$, where the symmetric part is $\mathbf{S} = \sqrt{\mathbf{A}_{pq}^T \mathbf{A}_{pq}}$ and the rotational part is $\mathbf{R} = \mathbf{A}_{pq}\mathbf{S}^{-1}$. Finally, the projected positions can be computed as

$$\mathbf{p}_i = \mathbf{R}(\mathbf{x}_i^0 - \mathbf{x}_{\text{cm}}^0) + \mathbf{x}_{\text{cm}}. \tag{5.14}$$

Linear Deformations

The method described so far can only simulate small deviations from the rigid shape. To extend the range of motion, the linear transformation matrix $\mathbf{A}$ computed in (5.13) can be used. This matrix describes the best linear transformation of the initial shape to match the actual shape in the least squares sense. Instead of using $\mathbf{R}$ in (5.14) to compute the $\mathbf{g}_i$, we use the combination $\beta\mathbf{A} + (1-\beta)\mathbf{R}$, where β is an additional control parameter. This way, the goal shape is allowed to undergo a linear transformation. The presence of $\mathbf{R}$ in the sum ensures that there is still a tendency towards the undeformed shape. To make sure that volume is conserved, we divide $\mathbf{A}$ by $\sqrt[3]{det(\mathbf{A})}$ ensuring that $det(\mathbf{A}) = 1$. For the standard approach we only need to compute $\mathbf{A}_{pq}$. Here, we also need the matrix $\mathbf{A}_{qq} = (\sum_i m_i \mathbf{q}_i \mathbf{q}_i^T)^{-1}$. Fortunately, this symmetric 3×3 matrix can be pre-computed.

Quadratic Deformations

Linear transformations can only represent shear and stretch. To extend the range of motion by twist and bending modes, we move from linear to quadratic transformations. We define a quadratic transformation as follows:

$$\mathbf{g}_i = [\mathbf{A}\ \mathbf{Q}\ \mathbf{M}]\tilde{\mathbf{q}}_i, \tag{5.15}$$

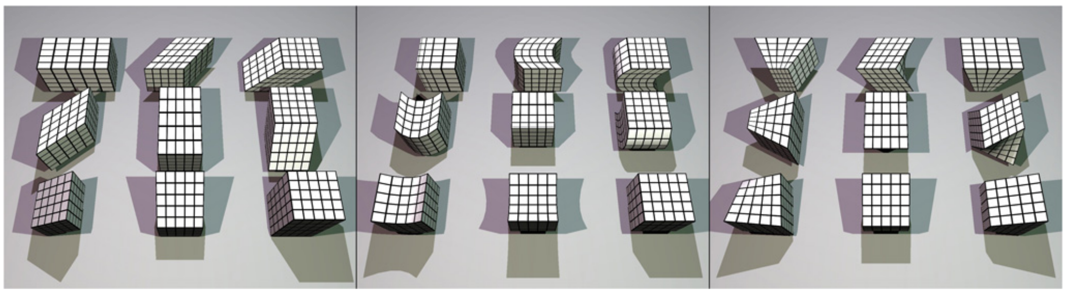


Figure 5.7: Visualization of all 3×9 modes defined by the coefficients of $\tilde{\mathbf{A}} = [\mathbf{A}\ \mathbf{Q}\ \mathbf{M}]$ defined in (5.15).

where $\mathbf{g}_i \in \mathbb{R}^3$, $\tilde{\mathbf{q}} = [q_x, q_y, q_z,\ q_x^2, q_y^2, q_z^2,\ q_x q_y, q_y q_z, q_z q_x]^T \in \mathbb{R}^9$, $\mathbf{A} \in \mathbb{R}^{3\times 3}$ contains the coefficients for the linear terms, $\mathbf{Q} \in \mathbb{R}^{3\times 3}$ the coefficients for the purely quadratic terms and $\mathbf{M} \in \mathbb{R}^{3\times 3}$ the coefficients for the mixed terms. With $\tilde{\mathbf{A}} = [\mathbf{A}\ \mathbf{Q}\ \mathbf{M}] \in \mathbb{R}^{3\times 9}$ we now have to minimize $\sum_i m_i(\tilde{\mathbf{A}}\tilde{\mathbf{q}}_i - \mathbf{p}_i)^2$. The optimal quadratic transformation turns out to be

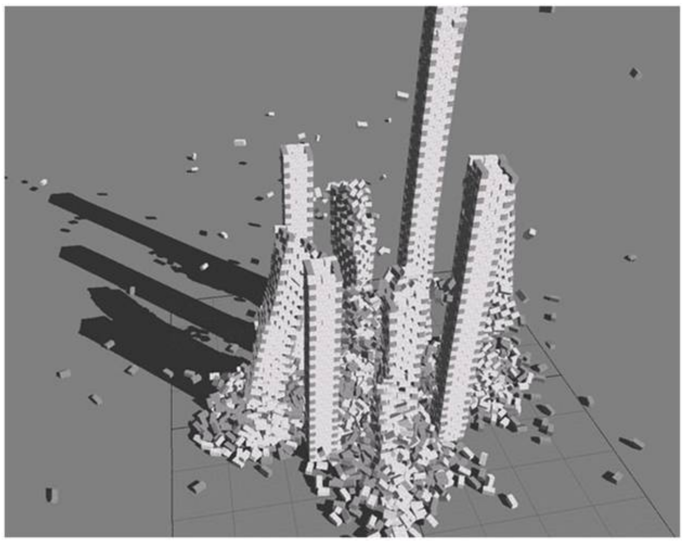
$$\tilde{\mathbf{A}} = (\sum_i m_i \mathbf{p}_i \tilde{\mathbf{q}}_i^T)(\sum_i m_i \tilde{\mathbf{q}}_i \tilde{\mathbf{q}}_i^T)^{-1} = \tilde{\mathbf{A}}_{pq}\tilde{\mathbf{A}}_{qq}. \tag{5.16}$$

Again, the symmetric $\tilde{\mathbf{A}}_{qq} \in \mathbb{R}^{9\times 9}$ as well as the $\tilde{\mathbf{q}}_i$ can be pre-computed. Analogous to the linear case, we use $\beta\tilde{\mathbf{A}} + (1-\beta)\tilde{\mathbf{R}}$ to compute the goal shape, where $\tilde{\mathbf{R}} \in \mathbb{R}^{3\times 9} = [\mathbf{R}\ \mathbf{0}\ \mathbf{0}]$. The algorithm based on quadratic deformations is a computationally cheap imitation of methods using modal analysis. The linear shear and stretch modes and the additional bend and twist modes are shown in Fig. 5.7.

Chapter 6

Rigid Body Simulation

Matthias Müller



Rigid bodies play a central role in real-time applications such as computer games because most objects around us can be considered as non-deformable. This is why the core of most game physics engines is a rigid body solver. Representing objects as rigid bodies is very effective, both in terms of simulation speed and memory footprint. Simulating rigid bodies in free flight and handling dynamic collisions is not that hard. Surprisingly it is much harder to stably simulate stacks of rigid bodies at rest than to simulate an explosion in which many objects interact with each other dynamically. In the resting scenario the player will notice even tiny jittering so impulse exchanges between interacting bodies have to be computed accurately and consistently. In the explosion scenario is is much harder to see whether objects behave in a physically correct way or not.

In the first part of this chapter, we will think of a rigid body as a set of point masses connected by springs of infinite stiffness. This allows us to start from what we learned in Chapter 3 about mass spring systems. Later, we will get rid of the particle representation.

6.1 Linear Motion

We represent a rigid body by particles having masses m_i, original positions $\bar{\mathbf{x}}_i$, actual positions $\mathbf{x}_i$ and velocities $\mathbf{v}_i$. The original and current center of mass are given by the mass-weighted sum of the positions of the particles as

$$\bar{\mathbf{x}} = \frac{1}{M}\sum_i m_i \bar{\mathbf{x}}_i \tag{6.1}$$

$$\mathbf{x} = \frac{1}{M}\sum_i m_i \mathbf{x}_i, \tag{6.2}$$

where $M = \sum_i m_i$ is the total mass of the body. We will now reorder the terms a bit to get

$$M\mathbf{x} = \sum_i m_i \mathbf{x}_i \tag{6.3}$$

$$M\ddot{\mathbf{x}} = \sum_i m_i \ddot{\mathbf{x}}_i \tag{6.4}$$

$$M\ddot{\mathbf{x}} = \sum_i \mathbf{f}_i \tag{6.5}$$

$$M\ddot{\mathbf{x}} = \mathbf{F}, \tag{6.6}$$

where we have used Newton's second law and $\mathbf{F} = \sum_i \mathbf{f}_i$. Thus, for linear motion the entire rigid body can be represented by a single particle at the center of mass with the total mass of the particles on which the sum of all particle forces acts.

6.2 Angular Motion

For linear motion, the assumption of rigidity let us replace individual particles by a single particle at the center of mass with the total mass of the body. The rigidity constraint also simplifies angular motion of the particles about the center of mass. On a rigid body, all the particles have to have the *same angular velocity*. Let us first look at Newton's second law for the angular case. The angular quantity that corresponds to linear impulse $\mathbf{p}_i = m_i \dot{\mathbf{x}}$ is called *angular momentum* and is defined as.

$$\mathbf{L}_i = \mathbf{r}_i \times (m_i \dot{\mathbf{x}}_i) = \mathbf{r}_i \times \mathbf{p}_i \tag{6.7}$$

where $\mathbf{r}_i$ is the current distance vector from the center of mass to the particle, i.e. $\mathbf{r}_i = \mathbf{x}_i - \mathbf{x}$. The cross product makes sure that only the part of the impulse perpendicular to the radius, i.e. along the tangent of the rotation is considered. The quantity that corresponds to force is called *torque* τ_i and defined as

$$\tau_i = \mathbf{r}_i \times \mathbf{f}_i. \tag{6.8}$$

In correspondence with Eq. (6.6), Newton's second law for the angular case reads

$$\begin{aligned} \sum_i \mathbf{r}_i \times \mathbf{f}_i &= \frac{d}{dt}\sum_i \mathbf{r}_i \times m_i \dot{\mathbf{x}} \\ \tau &= \dot{\mathbf{L}}. \end{aligned} \tag{6.9}$$

At this point we use the fact that all particles have the same rotational velocity ω. This allows us to express the angular part of their velocity as

$$\dot{\mathbf{x}}_i = \omega \times \mathbf{r}_i. \tag{6.10}$$

where ω is the angular velocity of the entire body. Using this fact we can simplify the expression for the angular momentum

$$\begin{aligned}
\mathbf{L} &= \sum_i \mathbf{r}_i \times m_i \dot{\mathbf{x}} \\
&= \sum_i \mathbf{r}_i \times m_i \, \omega \times \mathbf{r}_i \\
&= \sum_i -m_i \mathbf{r}_i \times \mathbf{r}_i \times \omega \\
&= \sum_i -m_i \, \mathrm{skew}(\mathbf{r}_i)\, \mathrm{skew}(\mathbf{r}_i) \omega \\
&= \mathbf{J}\omega,
\end{aligned} \tag{6.11}$$

where $\mathrm{skew}(\mathbf{r}) \in \mathbb{R}^{3\times 3}$ is the matrix with the property $\mathrm{skew}(\mathbf{r})\,\mathbf{x} = \mathbf{r} \times \mathbf{x}$. Because ω is the same for all particles, it can be pulled out of the sum while the remaining part is called the moment of inertia or inertia tensor

$$\mathbf{J} \in \mathbb{R}^{3\times 3} = \sum_i -m_i \, \mathrm{skew}(\mathbf{r}_i)\, \mathrm{skew}(\mathbf{r}_i) \tag{6.12}$$

Note that $\mathbf{J}$ depends on the current orientation of the rigid body. Given the current orientation as a rotation matrix $\mathbf{R}$ the current inertia tensor can computed from the one related to the original pose by using $\mathbf{J} = \mathbf{R}\bar{\mathbf{J}}\mathbf{R}^T$. There are four quantities that describe the state of a rigid body, namely

- the position of the center of mass $\mathbf{x}$,
- the orientation represented by a rotation matrix $\mathbf{R}$,
- the linear velocity of the center of mass $\dot{\mathbf{x}}$ and
- the angular velocity ω about the center of mass.

In rigid body simulators, the orientation is often represented by a *quaternion*. The orientation of a rigid body is a three dimensional quantity. A matrix has six additional degrees of freedom while a quaternion only has one additional dimension. Due to numerical errors in the time integration, both representations drift away from representing true orientations. This problem is more severe for matrices. Here we stick with the matrix representation because it is easier to understand. Also, one can always go back and forth from a quaternion to a matrix if needed.

It is more convenient to use the linear momentum $\mathbf{p} = M\dot{\mathbf{x}}$ instead of $\dot{\mathbf{x}}$ and the angular momentum $\mathbf{L} = \mathbf{J}\omega$ instead of ω. The state of a rigid body can then be expressed by

$$\mathbf{S}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{R}(t) \\ \mathbf{p}(t) \\ \mathbf{L}(t) \end{bmatrix}. \tag{6.13}$$

In order to simulate the body we need to know how this state vector changes over time, i.e. we need to know its time derivative $\dot{\mathbf{S}}$. First, we have Newton's second law saying that

$$\dot{\mathbf{p}} = \mathbf{F}$$

and from the definition of the linear momentum we have

$$\dot{\mathbf{x}} = M^{-1}\mathbf{p}.$$

while in the angular case Newton's second law states that

$$\dot{\mathbf{L}} = \tau.$$

The last piece is a bit more involved. What is $\dot{\mathbf{R}}$? The angular velocity ω rotates a vector around the origin yielding $\dot{\mathbf{x}} = \omega \times \mathbf{x}$. Likewise ω rotates the axes of $\mathbf{R} = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3]$ so that $\dot{\mathbf{R}} = [\omega \times \mathbf{r}_1, \omega \times \mathbf{r}_2, \omega \times \mathbf{r}_3] = \text{skew}(\omega)\mathbf{R}$. The angular velocity ω can be computed via $\omega = \mathbf{J}^{-1}\mathbf{L}$ while the inverse of the current inertia $\mathbf{J}^{-1}$ is related to the inverse of the original inertia tensor $\bar{\mathbf{J}}^{-1}$ by $\mathbf{J}^{-1} = \mathbf{R}\bar{\mathbf{J}}^{-1}\mathbf{R}^T$. This yields the final equation of motion for the rigid body:

$$\dot{\mathbf{S}} = \begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{R}} \\ \dot{\mathbf{p}} \\ \dot{\mathbf{L}} \end{bmatrix} = \begin{bmatrix} M^{-1}\mathbf{p} \\ \text{skew}(\mathbf{R}\bar{\mathbf{J}}^{-1}\mathbf{R}^T\mathbf{L})\mathbf{R} \\ \mathbf{F} \\ \tau \end{bmatrix}. \tag{6.14}$$

A simulator starts with an initial condition for the state of the rigid body and then integrates Eq. (6.14) in time. Using explicit Euler integration, a simulation algorithm would look like this:

```
// initialization
(1)  $M \leftarrow \sum_i m_i$
(2)  $\bar{\mathbf{x}} \leftarrow (\sum_i \bar{\mathbf{x}}_i)/M$
(3)  $\bar{\mathbf{r}}_i \leftarrow \bar{\mathbf{x}}_i - \bar{\mathbf{x}}$
(4)  $\bar{\mathbf{J}}^{-1} \leftarrow (-\sum_i m_i \text{skew}(\bar{\mathbf{r}}_i)\text{skew}(\bar{\mathbf{r}}_i))^{-1}$
(5)  initialize $\mathbf{x}, \mathbf{v}, \mathbf{R}, \mathbf{L}$
(6)  $\mathbf{J}^{-1} \leftarrow \mathbf{R}\bar{\mathbf{J}}^{-1}\mathbf{R}^T$
(7)  $\omega \leftarrow \mathbf{J}^{-1}\mathbf{L}$
// simulation
(8)  loop
(9)     $\mathbf{F} \leftarrow \sum_i \mathbf{f}_i$
(10)    $\tau \leftarrow \sum_i \mathbf{r}_i \times \mathbf{f}_i$
(11)    $\mathbf{x} \leftarrow \mathbf{x} + \Delta t \mathbf{v}$
(12)    $\mathbf{v} \leftarrow \mathbf{v} + \Delta t \mathbf{F}/M$
(13)    $\mathbf{R} \leftarrow \mathbf{R} + \Delta t\, \text{skew}(\omega)\mathbf{R}$
(14)    $\mathbf{L} \leftarrow \mathbf{L} + \Delta t \tau$
(15)    $\mathbf{J}^{-1} \leftarrow \mathbf{R}\bar{\mathbf{J}}^{-1}\mathbf{R}^T$
(16)    $\omega \leftarrow \mathbf{J}^{-1}\mathbf{L}$
(17)    $\mathbf{r}_i \leftarrow \mathbf{R}\bar{\mathbf{r}}_i$
```

```
(18)    $\mathbf{x}_i \leftarrow \mathbf{x} + \mathbf{r}_i$
(19)    $\mathbf{v}_i \leftarrow \mathbf{v} + \omega \times \mathbf{r}_i$
(20) endloop
```

6.3 Collision handling

There are two non-degenerate cases for collisions of two rigid bodies A and B, namely corner-face collisions and edge-edge collisions. The *contact normal* $\mathbf{n}$ is an important quantity in rigid body collision handling. In the case of a corner-face collision, $\mathbf{n}$ is perpendicular to the collision face. When two edges collide, $\mathbf{n}$ points along the cross product of the direction of the two edges. Forces and impulses are always exchanged along $\mathbf{n}$ and only the component of contact velocities along $\mathbf{n}$ have to be considered. Therefore, all the quantities relevant for collision handling at a given contact point can be represented by scalars along $\mathbf{n}$ which simplifies things quite a lot!

In the real world, fully rigid bodies do not exist. In almost rigid objects, impacts cause tiny deformations at the contact point which, in turn, cause high stress forces that accelerate the bodies away from each other. Only after a finite amount of time, the bodies are fully separated again. The stiffer the materials, the stronger the stresses and the shorter the response time. So hypothetically, for infinitely stiff materials, the response time is zero and the velocities change immediately. We use the symbols u^- and u^+ for velocities before and after the collision along the contact normal, so

$$\begin{aligned} u_A^- &= (\mathbf{v}_A^- + \mathbf{w}_A^- \times \mathbf{r}_A) \cdot \mathbf{n} \\ u_B^- &= (\mathbf{v}_B^- + \omega_B^- \times \mathbf{r}_B) \cdot \mathbf{n}, \end{aligned} \tag{6.15}$$

where $\mathbf{r}_A$ and $\mathbf{r}_B$ are the positions of the collision point relative the center of mass of body A and B respectively. The relative velocity at the collision point before the collision is

$$u_{\text{rel}}^- = u_A^- - u_B^- \tag{6.16}$$

Instead of using a collision force we use a collision *impulse* $\mathbf{p} = \mathbf{n}p$ which changes this relative velocity immediately. Because it acts parallel to the contact normal it can be represented by a scalar p. For a resting contact and a bouncy contact the impulse p has to be chosen such that the relative velocity after the collision

$$\begin{aligned} u_{\text{rel}}^+ &\geq 0 \\ u_{\text{rel}}^+ &= -\varepsilon u_{\text{rel}}^- \end{aligned} \tag{6.17}$$

respectively, where ε is the coefficient of restitution. For $\varepsilon = 1$ the collision is fully elastic while for $\varepsilon = 0$ it is fully inelastic. An impulse $\mathbf{p}$ changes the linear and angular velocities of body A via

$$\begin{aligned} \mathbf{p} &= M_A \Delta \mathbf{v}_A \\ \mathbf{r}_A \times \mathbf{p} &= \mathbf{J}_A \Delta \omega_A \end{aligned} \tag{6.18}$$

which results in a change in u_A of

$$\begin{aligned}
\Delta u_A &= [\Delta \mathbf{v}_A + \Delta \omega_A \times \mathbf{r}_A] \cdot \mathbf{n} && (6.19)\\
&= \left[\mathbf{p} M_A^{-1} + (\mathbf{J}_A^{-1}(\mathbf{r}_A \times \mathbf{p})) \times \mathbf{r}_A\right] \cdot \mathbf{n} && (6.20)\\
&= p \left[\mathbf{n} M_A^{-1} + (\mathbf{J}_A^{-1}(\mathbf{r}_A \times \mathbf{n})) \times \mathbf{r}_A\right] \cdot \mathbf{n} && (6.21)\\
&= p \left[\mathbf{n} \cdot \mathbf{n} M_A^{-1} + (\mathbf{J}_A^{-1}(\mathbf{r}_A \times \mathbf{n})) \times \mathbf{r}_A \cdot \mathbf{n}\right] && (6.22)\\
&= p \left[M_A^{-1} + (\mathbf{r}_A \times \mathbf{n})^T \mathbf{J}_A^{-1}(\mathbf{r}_A \times \mathbf{n})\right] && (6.23)\\
&= p\, w_A && (6.24)
\end{aligned}$$

From step (6.22) to (6.23), the fact that the term after the plus sign is a triple product was used. The formula for Δu_b is obviously analogous. The scalar w_A can be computed from quantities known before the collision. We are now ready to solve for p using Eq. (6.17):

$$\begin{aligned}
u_{\text{rel}}^+ &= u_{\text{rel}}^- + \Delta u_A - \Delta u_B \\
&= u_{\text{rel}}^- + p w_A - (-p w_B) \\
&= u_{\text{rel}}^- + p(w_A + w_B) \overset{!}{=} -\varepsilon u_{\text{rel}}^-
\end{aligned} \tag{6.25}$$

It is important to note that while $\mathbf{p}$ is applied to body A, $-\mathbf{p}$ is applied to body B. Solving for p yields

$$p = \frac{-(1+\varepsilon) u_{\text{rel}}^-}{w_A + w_B} \tag{6.26}$$

Once p is known, the kinematic quantities can be updated using

$$\begin{aligned}
\mathbf{v}_A^+ &= \mathbf{v}_A^- + p\, \mathbf{n} M_A^{-1} \\
\mathbf{v}_B^+ &= \mathbf{v}_B^- - p\, \mathbf{n} M_B^{-1} \\
\omega_A^+ &= \omega_A^- + p\, \mathbf{J}_A^{-1}(\mathbf{r}_A \times \mathbf{n}) \\
\omega_B^+ &= \omega_A^- - p\, \mathbf{J}_B^{-1}(\mathbf{r}_B \times \mathbf{n})
\end{aligned} \tag{6.27}$$

6.4 Resting Contacts

In a resting configuration there are multiple (say N) contact points for which the solver has to make sure that $u_{\text{rel}}^+ >= 0$. The change of the velocity Δu_A^i of body A at contact i is influenced by *all* the impulses of contacts j that influence body A:

$$\begin{aligned}
\Delta u_A^i &= \left[\sum_j \mathbf{p}^j M_A^{-1} + \sum_j (\mathbf{J}_A^{-1}(\mathbf{r}_A^j \times \mathbf{p}^j)) \times \mathbf{r}_A^i\right] \cdot \mathbf{n}^i \\
&= \sum_j p_j \left[\mathbf{n}^j M_A^{-1} + (\mathbf{J}_A^{-1}(\mathbf{r}_A^j \times \mathbf{n}^j)) \times \mathbf{r}_A^i\right] \cdot \mathbf{n}^i \\
&= \sum_j p_j \left[\mathbf{n}^i \cdot \mathbf{n}^j M_A^{-1} + (\mathbf{r}_A^i \times \mathbf{n}^i)^T \mathbf{J}_A^{-1}(\mathbf{r}_A^j \times \mathbf{n}^j)\right] \\
&= \sum_j p_j\, w_A^{ij}
\end{aligned} \tag{6.28}$$

This yields a linear equation for each contact

$$\begin{aligned} u_{\mathrm{rel}}^{i+} &= u_{\mathrm{rel}}^{i-} + \Delta u_A^i - \Delta u_B^i \\ &= u_{\mathrm{rel}}^{i-} + \sum_j (\sigma_A^j w_A^{ij} - \sigma_B^j w_B^{ij})\, p_j \\ &= u_{\mathrm{rel}}^{i-} + \sum_j a_{ij}\, p_j \\ &\geq 0 \end{aligned} \tag{6.29}$$

which is a system of N inequalities for the scalars p_i. The coefficient σ_A^j is $+1$ if $\mathbf{p}_j$ is applied to body A in the direction of $\mathbf{n}_j$ and -1 if $\mathbf{p}_j$ is applied to body A in the opposite direction of $\mathbf{n}_j$. It is zero if $\mathbf{p}_j$ is not applied to body A at all. The coefficients σ_B^j are defined similarly. We use the following definitions

$$\begin{aligned} \mathbf{A} &= [a_{ij}] \in \mathbb{R}^{N\times N} \\ \mathbf{p} &= [p_1, p_2, \dots p_N]^T \in \mathbb{R}^{N\times 1} \\ \mathbf{u} &= [u_{\mathrm{rel}}^{1-}, u_{\mathrm{rel}}^{2-}, \dots, u_{\mathrm{rel}}^{N-}]^T \in \mathbb{R}^{N\times 1} \\ \mathbf{c} &= [c_1, c_2, \dots c_N]^T \in \mathbb{R}^{N\times 1}, \end{aligned} \tag{6.30}$$

where $c_i = |u_{\mathrm{rel}}^{i-}|$ are the magnitudes of the pre impact relative velocities. Now we can state the problem as

$$\begin{aligned} \mathbf{Ap} + \mathbf{u} &\geq \mathbf{0} \\ \mathbf{p} &\geq \mathbf{0}, \end{aligned} \tag{6.31}$$

where the symbol $\geq$ is used componentwise. All the vectors $\mathbf{p}$ within the polytope defined by the inequalities are solutions to this problem. To get a unique solution we need additional constraints. One additional constraint requires $\mathbf{Ap} + \mathbf{u} \leq \mathbf{c}$, i.e. that kinetic energy cannot be gained by the collision. Among all remaining solution vectors $\mathbf{p}$ we want the one that minimizes the quadratic function $|\mathbf{Ap} + \mathbf{u}|^2$ namely the magnitudes of the post velocities. This is a convex quadratic programming problem (QP) that can be formulated as a linear complementarity problem (LCP).

6.5 Dynamic and Static Friction

For simulating friction, velocities perpendicular to the contact normal $\mathbf{n}$ have to be considered. Let p be the magnitude of the impulse along $\mathbf{n}$ computed as shown above. One way to do this is to add two additional constraints to each contact point with normals $\mathbf{n}_1^\perp$ and $\mathbf{n}_2^\perp$ perpendicular to $\mathbf{n}$ and to each other. The scalar impulses $p_1^\perp$ and $p_2^\perp$ along these two additional directions have to be solved for with the original impulses simultaneously.

In the case of dynamic friction, the condition must hold that $\sqrt{p_1^{\perp 2} + p_2^{\perp 2}} = c_{\mathrm{dyn}} p$ where c_{dyn} is the dynamic friction constant. The three impulse magnitudes can be thought of as lying on a cone centered at the collision point.

For static friction the situation is different. If $p \geq p_{max}$ the solver has to make sure that the relative velocities along the tangential directions are zero by applying the appropriate impulses $p_1^\perp$ and $p_2^\perp$. The scalar p_{max} is a material parameter. It is non-trivial to include these constraints into the non-frictional problem. Fortunately there is a very simple way to solve both, the original LCP as well as the additional friction constraints, namely the Gauss-Seidel technique.

6.6 Real Time Simulation using a Gauss-Seidel Solver

The main drawback of a Gauss-Seidel solver (GS) is its slow convergence in comparison to global solvers. There are many advantages though. The most important one is the simplicity with which the complex LCP including static and dynamic friction constraints can be solved. In addition, ill posed and over constraint problems are handled in a robust manner. These properties are ideal for the use in computer games and other real time environments.

The general idea is very simple. Instead of considering the global problem, GS iterates through each constraint one after the other, computes impulses locally using Eq. (6.26) and updates the kinematic quantities of the participating bodies immediately using Eq. (6.27). Before such a step, the relative velocity along the constraint is computed using the current linear and angular velocities of the bodies. If this velocity is greater than zero, the constraint is simply skipped. The solver also makes sure that the impulses applied are always larger than zero which can be easily done when looking at one single constraint. For static friction, the tangential constraints solve for zero velocities but only if the current normal impulse is above the static friction threshold, which can be different for the two directions resulting in an-isotropic friction.

An important problem that comes with a velocity based approach is drift. Making sure that relative velocities are zero does not prevent penetrations. Once there is a penetration due to numerical errors for instance, the solver does not remove them. The solution to this problem is to move the bodies apart. One has to be careful to do this correctly. It is not correct to apply a simple translation. Rather, one has to solve for velocities that remove the penetration when multiplied by the time step.

Chapter 7

Reduced-order deformable models

Chapter 8

User Interaction and Control

Chapter 9

Introduction to Fluids

Nils Thuerey

As fluids are everywhere around us, they are an important part of physical simulations for virtual environments. In addition, correctly solving the underlying equations can result in complex and beautiful structures and motions. Numerical simulations of fluid flow have been used for a long time in the field of computer graphics, although mainly for off-line animations (an example is shown in Fig. 9.1). In [YUM86], fluid simulations were first used for the generation of animated textures. In 1990, Kass and Miller [KM90] demonstrated the practical use and efficiency of height field fluids. First three dimensional fluid simulations were performed in [FM96] by Foster and Metaxas.

The *stable fluids* approach, as presented by Jos Stam in [Sta99], made it possible to guarantee stability of the advection step, and lead to a variety of applications in the field. With this approach, level-set based simulations of liquids were made popular by the group of Ron Fedkiw, starting with [FF01, EMF02]. Since then, a variety of extensions and improvements of the original algorithm have been proposed. Fluid simulations have been used for fire [NFJ02], large scale smoke [RNGF03], particle based explosions [FOA03], and were coupled to rigid bodies [CMT04, BBB07]. A first real-time GPU implementation of a level-set based fluid solver, yielding a high performance, was demonstrated in [CLT07]. Due to the long run-times, control of off-line simulations also became an important topic [MTPS04, SY05a, SY05b, TKPR06]. We will discuss eulerian fluid simulations, based on this semi Lagrangian method, within the field of real-time applica-

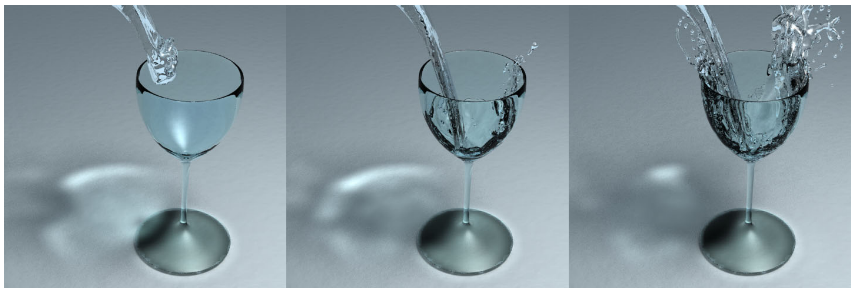


Figure 9.1: A typical off-line fluid animation: liquid is poured into a glass shaped obstacle. Animation and rendering require many hours to compute.

tions in Chapter 10. In contrast to the previously mentioned work, the group of James O'Brien worked on algorithms to use dynamic meshes instead of equidistant Cartesian grids [KFCO06, FOK05, CFL$^+$07]. However, due to the complexity of these approaches, they are not aimed at real-time simulations, and will not be discussed in the following. A different approach to guided fluid simulations was presented by Angelidis et al. in [AN05] and [ANSN06]. It is based on vortex filaments, and allows for real-time smoke simulations and control.

Due to the limitations of an underlying grid, particle based approaches are especially popular for real-time simulations, as they do not require a grid throughout the whole domain. They are based on the so-called *smoothed particle hydrodynamics* (SPH) that were originally developed for astro-physics applications [Mon92]. The first SPH paper to target the real-time simulation of liquids was [MCG03]. This method was later extended to, e.g., handle multi-phase fluids in [MSKG05]. As SPH does not require a global correction step for the incompressibility of the fluid, the resulting compressibility artifacts are still a topic of current research activities [BT07].

Due to the simplicity and efficiency of the underlying algorithm, *lattice Boltzmann methods* (LBM) have become an interesting alternative to the approaches mentioned above. They have been demonstrated, e.g., in [WZF$^+$03, LWK03, WLMK04] for wind simulations with interacting bodies. In [WWXP06], the LBM was used to simulate falling snow in real-time. It was also used for simulations of free surface flows, e.g., in [TR04] and [TRS06]. We will explain how to implement a fluid solver with LBM later on in Section 10.2.

For real-time applications, the *shallow water equations* (SWE) are especially interesting. The SWE are a reduced form of the Navier-Stokes equations, and can be solved very efficiently due to their two-dimensional nature. In simplified versions, such two-dimensional methods have been used, e.g., for generating splashes [OH95] or coupled to terrains and rigid bodies [CdVLHM97]. In [LvdP02], an algorithm to apply a semi Lagrangian advection to the SWE with an implicit time integration scheme was introduced. Moreover, the underlying algorithm was extended to achieve a variety of different effects in real-time, such as bubble dynamics [TSS$^+$07], or terrain erosion [MDH07]. A different class of methods for wave generation are the spectral approaches, e.g., as described in [HNC02], [Lov03], or in [Tes04]. These approaches are especially suitable for deep water waves with dispersive effects, but will not be discussed in more detail here. In Chapter 11, we will explain how to solve the SWE, and describe the handling of different boundary conditions, to achieve, e.g., fluid flowing through a terrain.

Chapter 10

Grid Based Fluid Simulation

Nils Thuerey

10.1 Navier-Stokes Equations

The motion of a fluid is typically computed by solving the *Navier-Stokes* (NS) equations [1], which describe the fluid in terms of a continuous velocity field **u** and a pressure p. In the following, we will give an overview of how a simplified form of these equations is commonly solved in the field of computer graphics. The next section will discuss an alternative solution approach that is particularly well suited for parallel implementations such as GPUs. For more detailed information, e.g., on the derviation of the equations, there are many good text books on the topic. We can, e.g., recommend [KC04]. Note that a very detailed description of how to solve the NS equations directly can be found in the notes of the SIGGRAPH course on fluid simulation [RB07].

The NS equations are basically two conservation equations, that have to be fulfilled in order to give the motion of an incompressible fluid. The first one is very simple - it ensures the conservation of mass in the velocity field. Assuming a constant density of the fluid, this can be written as

$$\nabla \mathbf{v} = 0 \ . \tag{10.1}$$

It simply means that when some quantity is advected in the velocity field **v**, a smoke density, for instance, the overall mass of this quantity will not change (assuming that we make no error computing this advection). For fluids it is of course highly important to keep the mass constant - otherwise it will look like the fluid evaporates over time, or that drops disappear while flying through the air. The second, and more complicated part, of the NS equations

[1]The origins of the well established Navier-Stokes equations reach back to Isaac Newton, who, around 1700, formulated the basic equations for the theoretical description of fluids. These were used by L. Euler half a century later to develop the basic equations for momentum conservation and pressure. Amongst others, Louis M. H. Navier continued to work on the fluid mechanic equations at the end of the 18th century, as did Georg G. Stokes several years later. He was one of the first to analytically solve fluid problems for viscous media. The NS equations could not be practically used up to the middle of the 20th century, when the numerical methods, that are necessary to solve the resulting equations, were developed.

ensures that the momentum of the fluid is preserved. It can be written as

$$\underbrace{\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)}_{\text{advection}}+\underbrace{\nabla P}_{\text{pressure}}=\underbrace{\nu\triangle\mathbf{u}}_{\text{viscosity}}+\underbrace{\rho\mathbf{g}}_{\text{forces}}. \tag{10.2}$$

This equation is a lot more complicated, but can be understood by identifying its individual terms. The first one, the advection terms, deals with just that - the advection of the velocity field of the fluid with itself. The pressure term involves the gradient of the pressure, so it essentially makes sure the fluid tries to flow away from regions with high pressure. The viscosity term, where ν is a parameter to account for the different ”thicknesses” of fluids (honey, e.g., has a much higher viscosity than water), involves the second derivates of the velocities, and as such, performs a smearing out of the information in the velocity field. As we are usually interested in computing fluids with a very low viscosity, such as water and air, this term is completely left out (the NS equations without this term are then called the Euler equations). We can do this because the method to solve the equations normally introduces a fair amount of error, which is noticeable as a viscosity. Likewise, the more accurately the equations are solved, the more lively (and less viscous) the fluid will look.

Luckily, the different parts of this equation can be computed separately, which significantly simplifies the algorithm. As mentioned before, the viscosity term can be left out. Adding the forces (this could, e.g., include gravity) to the velocities is typically also very easy. Computing the advection is somewhat more difficult. A very elegant and stable way to solve this is to use a semi-Lagrangian scheme [Sta99]. This step will be described in more detail later for solving the shallow water equations. It can be computed very efficiently and is unconditionally stable. This is very useful, as the nonlinear terms of the advection are usually very problematic for stability when using other schemes such as finite differences.

The last, and most complex term to handle in the equations is the pressure term. Given some the velocity field computed from the previous steps, we compute the pressure values in a way that ensures that (10.1) holds. It turns out that a Poisson equation with corresponding boundary conditions has to be solved for the pressure. This is a well known equation in mathematics: it is a partial differential equation that requires a system of linear equations to be solved. Typically, this is done with an iterative method, such as a conjugate gradient solver. It is usually by far the most expensive part of the algorithm. Once a correct pressure is computed, the velocity field can be adjusted to be divergence free, and describes the state of the fluid for the next time step.

These steps: adding forces, advecting the velocities, and computing a pressure correction are repeated for each time step. They are used to compute the motion of a single fluid phase, and can be used to, e.g., simulate swirling smoke. For effects such as liquids, it is in addition to the basic algorithm necessary to compute the motion and behavior of the interface between the liquid and its surrounding (usually air). We will only cover two-phase flows in the form of the shallow water equations in Chapter 11. In the next section we will explain a different approach to compute the motion of a fluid that is based on a simpler algorithm.

10.2 Lattice Boltzmann Methods

This section will describe the basic LBM algorithm. The lattice Boltzmann formulation is also a grid-based method, but originated from the field of statistical physics [2]. Although it comes from a descriptions on the molecular level, the method can still be used to compute fluids on a larger scale. For simulations with LBM, the simulation region is typically represented by a Cartesian and equidistant grid of cells. Each cell only interacts with cells in its direct neighborhood. While conventional solvers directly discretize the NS equations, the LBM is essentially a first order explicit discretization of the Boltzmann equation in a discrete phase-space. It can also be shown, that the LBM approximates the NS equations with good accuracy. A detailed overview of the LBM and derivations of the NS equations from LBM can be found in [WG00] and [Suc01], among others.

The LBM has several advantages, that make it interesting for real-time applications. First, a timestep with LBM only requires a single pass over the computational grid. So it maps very well to parallel architectures such as GPUs. On the other hand, as the LBM performs an explicit timestep, the allowed velocities in the simulation are typically limited to ensure stability. Another interesting aspect is that each cell in LBM contains more information than just the velocity and pressure or density. This makes it possible to handle complex boundaries with high accuracy even on relatively coarse grids. The following section will describe the standard LBM with simple boundary conditions, and a turbulence model to ensure stability.

10.3 The Basic Algorithm

The basic lattice Boltzmann (LB) algorithm consists of two steps, the stream-step, and the collide-step. These are usually combined with no-slip boundary conditions for the domain boundaries or obstacles. The simplicity of the algorithm is especially evident when implementing it, which, for the basic algorithm, requires roughly a single page of C-code. Using a LBM, the particle movement is restricted to a limited number of directions. Here, a three-dimensional model with 19 velocities (commonly denoted as *D3Q19*) will be used. Alternatives are models with 15 or 27 velocities. However, the latter one has no clear advantages over the 19 velocity model, while the model with 15 velocities has a decreased stability. The D3Q19 model is thus usually preferable as it requires less

[2]The Boltzmann equation itself has been known since 1872. It is named after the Austrian scientist Ludwig Boltzmann, and is part of the classical statistical physics that describe the behavior of a gas on the microscopic scale. The LBM follows the approach of cellular automata to model even complex systems with a set of simple and local rules for each cell [Wol02]. As the LBM computes macroscopic behavior, such as the motion of a fluid, with equations describing microscopic scales, it operates on a so-called "mesoscopic" level in between those two extremes.

Historically, the LBM evolved from methods for the simulation of gases that computed the motion of each molecule in the gas purely with integer operations. In [HYP76], there was a first attempt to perform fluid simulations with this approach. It took ten years to discover that the isotropy of the lattice vectors is crucial for a correct approximation of the NS equations [FdH$^+$87]. Motivated by this improvement, [MZ88] developed the first algorithm that was actually called LBM by performing simulations with averaged floating point values instead of single fluid molecules. The third important contribution to the basic LBM was the simplified collision operator with a single time relaxation parameter [BGK54, CCM92, QdL92].

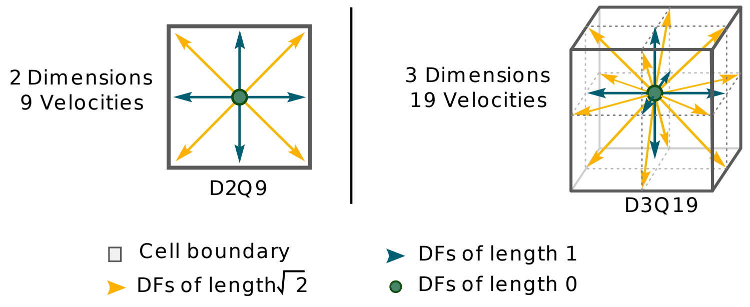


Figure 10.1: The most commonly used LBM models in two and three dimensions.

memory than the 27 velocity model. For two dimensions the D2Q9 model with nine velocities is the most common one. The D3Q19 model with its lattice velocity vectors $\mathbf{e}_{1..19}$ is shown in Fig. 10.1 (together with the D2Q9 model). The velocity vectors take the following values: $\mathbf{e}_1 = (0,0,0)^T$, $\mathbf{e}_{2,3} = (\pm1,0,0)^T$, $\mathbf{e}_{4,5} = (0,\pm1,0)^T$, $\mathbf{e}_{6,7} = (0,0,\pm1)^T$, $\mathbf{e}_{8..11} = (\pm1,\pm1,0)^T$, $\mathbf{e}_{12..15} = (0,\pm1,\pm1)^T$, and $\mathbf{e}_{16..19} = (\pm1,0,\pm1)^T$. As all formulas for the LBM usually only depend on the so-called particle distribution functions (*DFs*), all of these two-dimensional and three-dimensional models can be used with the method presented here. To increase clarity, the following illustrations will all use the D2Q9 model.

For each of the velocities, a floating point number $f_{1..19}$, representing a blob of fluid moving with this velocity, needs to be stored. As the LBM originates from statistical physics, this blob is thought of as a collection of molecules or particles. Thus, in the D3Q19 model there are particles not moving at all (f_1), moving with speed 1 ($f_{2..7}$) and moving with speed $\sqrt{2}$ ($f_{8..19}$). In the following, a subscript of $\tilde{i}$ will denote the value from the inverse direction of a value with subscript i. Thus, f_i and $f_{\tilde{i}}$ are opposite DFs with inverse velocity vectors $\mathbf{e}_{\tilde{i}} = -\mathbf{e}_i$. During the first part of the algorithm (the stream step), all DFs are advected with their respective velocities. This propagation results in a movement of the floating point values to the neighboring cells, as shown in Fig. 10.2. Formulated in terms of DFs the stream step can be written as

$$f_i^*(\mathbf{x}, t+\Delta t) = f_i(\mathbf{x}+\Delta t\ \mathbf{e}_{\tilde{i}}, t). \tag{10.3}$$

Here, Δx denotes the size of a cell and Δt the time step size. Both are normalized by the condition $\Delta t/\Delta x = 1$, which makes it possible to handle the advection by a simple copying operation, as described above. These post-streaming DFs f_i^* have to be distinguished from the standard DFs f_i, and are never really stored in the grid. The stream step alone is clearly not enough to simulate the behavior of incompressible fluids, which is governed by the ongoing collisions of the particles with each other. The second part of the LBM, the collide step, amounts for this by weighting the DFs of a cell with the so called *equilibrium distribution functions*, denoted by f_i^{eq}. These depend solely on the density and velocity of the fluid. Here, the incompressible model from [HL97] is used, which alleviates compressibility effects of the standard model by using a modified equilibrium DF and velocity calculation. The density and velocity can be computed by summation of all the DFs for one cell

$$\rho = \sum f_i \qquad \mathbf{u} = \sum \mathbf{e}_i f_i \quad . \tag{10.4}$$

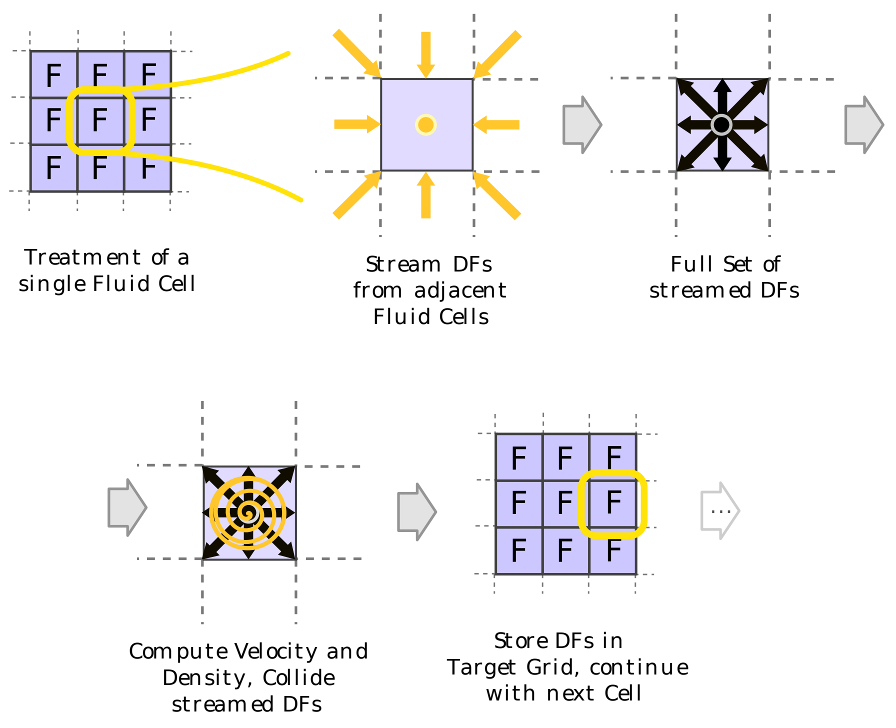

F
Treatment of a single Fluid Cell
Stream DFs from adjacent Fluid Cells
Full Set of streamed DFs
Compute Velocity and Density, Collide streamed DFs
Store DFs in Target Grid, continue with next Cell
...

Figure 10.2: This figure gives an overview of the stream and collide steps for a fluid cell.

The standard model, in contrast to the one used here, requires a normalization of the velocity with the fluid density. For a single direction i, the equilibrium DF f_i^{eq} can be computed with

$$\begin{aligned} f_i^{eq} &= w_i \left[\rho + 3\mathbf{e_i}\cdot\mathbf{u} - \frac{3}{2}\mathbf{u}^2 + \frac{9}{2}(\mathbf{e_i}\cdot\mathbf{u})^2 \right], \quad \text{where} \\ w_i &= 1/3 \quad \text{for } i = 1, \\ w_i &= 1/18 \quad \text{for } i = 2..7, \\ w_i &= 1/36 \quad \text{for } i = 8..19. \end{aligned} \tag{10.5}$$

The equilibrium DFs represent a stationary state of the fluid. However, this does not mean that the fluid is not moving. Rather, the values of the DFs would not change, if the whole fluid was at such an equilibrium state. For very viscous flows, such an equilibrium state (equivalent to a *Stokes* flow) can be globally reached. In this case, the DFs will converge to constant values. The collisions of the molecules in a real fluid are approximated by linearly relaxing the DFs of a cell towards their equilibrium state. Thus, each f_i is weighted with the corresponding f_i^{eq} using:

$$f_i(\mathbf{x}, t+\Delta t) = (1-\omega) f_i^*(\mathbf{x}, t+\Delta t) + \omega f_i^{eq}. \tag{10.6}$$

Here, ω is the parameter that controls the viscosity of the fluid. Often, $\tau = 1/\omega$ is also used to denote the lattice viscosity. The parameter ω is in the range of $(0..2]$, where values close to 0 result in very viscous fluids, while values near 2 result in more turbulent flows. Usually these are also visually more interesting. However, for values close to 2, the method can

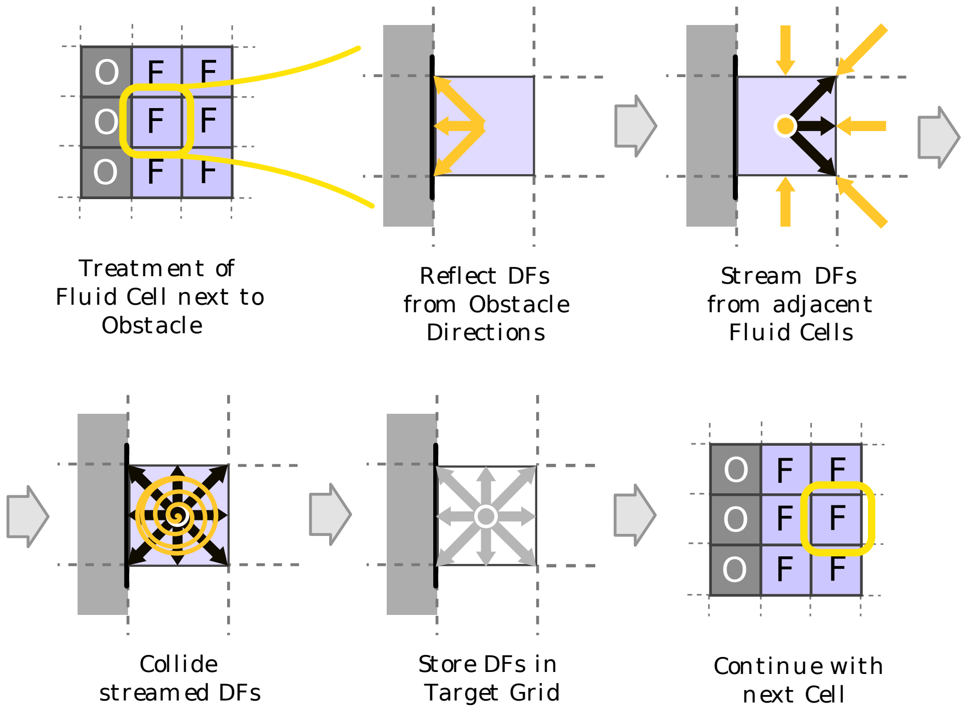

O
F
Treatment of Fluid Cell next to Obstacle
Reflect DFs from Obstacle Directions
Stream DFs from adjacent Fluid Cells
Collide streamed DFs
Store DFs in Target Grid
Continue with next Cell

Figure 10.3: This figure gives an overview of the stream and collide steps for a fluid cell next to an obstacle.

become instable. In Section 10.5, a method to stabilize the computations with a turbulence model will be explained. This alleviates the instabilities mentioned above. The parameter ω is given by the kinematic viscosity of a fluid. The values computed with Eq. (10.6) are stored as DFs for time $t+\Delta t$. As each cell needs the DFs of the adjacent cells from the previous time step, two arrays for the DFs of the current and the last time step are usually used.

The easiest way to implement the no-slip boundary conditions is the link bounce back rule that results in a placement of the boundary halfway between fluid and obstacle cells. If the neighboring cell at $(\mathbf{x}+\Delta t\ \mathbf{e}_i)$ is an obstacle cell during streaming, the DF from the inverse direction of the current cell is used. Thus, Eq. (10.3) changes to

$$f_i^*(\mathbf{x},t+\Delta t) = f_{\tilde{i}}(\mathbf{x},t). \tag{10.7}$$

Fig. 10.3 illustrates those basic steps for a cell next to an obstacle cell.

10.4 Implementation

A 2D implementation of the algorithm described so far might consist of a flag field g to distinguish fluid ($g = FLUID$) and obstacle cells ($g = OBS$), and two arrays of single-precision floating point variables, f and f', with 9 values for each cell in the grid. For a grid with n_x and n_y cells along the x- and y-direction, respectively, a typical initialization step might be to set the boundaries of the domain to be obstacles, and initialize a resting fluid volume:

Init-lbm

```
(1) for $j = 0$ to $n_y$ do
(2)   for $i = 0$ to $n_x$ do
(3)     if$(i == 0 || i == n_x - 1 || j == 0 || j == n_y - 1)$ do
(4)       $g[i,j] = OBS$
(5)     else
(6)       $g[i,j] = FLUID$
(5)     endif
(7)     for $l = 0$ to 9 do
(7)       $f[i,j,l] = f'[i,j,l] = w_l$;
(7)     endfor
(8)   endfor
(9) endfor
```

The code above and below requires some globals arrays: the weights of the equilibrium DFs w, and the lattice vectors. For the D2Q9 model, these are

Lbm D2Q9 Globals

```
(1) $w_l[9] = \{1/3,\ 1/18, 1/18, 1/18, 1/18,\ 1/18, 1/18, 1/18, 1/18\}$
(2) $e_x[9] = \{0,\ 1, -1, 0, 0,\ 1, -1, 1, -1\}$
(3) $e_y[9] = \{0,\ 0, 0, 1, -1,\ 1, 1, -1, -1\}$
```

During a loop over all cells in the current grid, each cell collects the neighboring DFs according to Eq. (10.3) or Eq. (10.7), for adjacent fluid and obstacle cells, respectively.

Stream

```
(1) for $j = 0$ to $n_y$ do
(2)   for $i = 0$ to $n_x$ do
(3)     for $l = 0$ to 8 do
(4)       $l_{inv} = \text{invert}(l)$
(5)       if $g[i + e_x[l_{inv}], j + e_y[l_{inv}]] == OBS$ do
(6)         $f'[i,j,l] = f[i,j,l_{inv}]$
(7)       else
(8)         $f'[i,j,l] = f[i + e_x[l_{inv}], j + e_y[l_{inv}], l]$
(9)       endif
(10)    endfor
(11)  endfor
(12) endfor
```

Here, e_x and e_y denote the x- and y-components of a lattice vector $\mathbf{e}_i$, while invert(l) is a function to return the index of the lattice vector opposite to $\mathbf{e}_i$. The pseudo code above performs the normal streaming operation for all fluid neighbors, and uses a bounce-back boundary conditions for obstacle cells.

The next step is to compute the collisions of the streamed DFs. For this, the density and velocity are computed and used to calculate the equilibrium DFs. These are weighted with the streamed DFs according to the viscosity parameter ω.

Collide

```
(1)  for $j = 0$ to $n_y$ do
(2)    for $i = 0$ to $n_x$ do
(3)      if $g[i, j] == OBS$ do continue; endif
(4)      $(\rho, u_x, u_y) = \text{getDensityAndVelocity}(i, j, k)$
(5)      for $l = 0$ to 8 do
(6)        $a = e_x[l]u_x + e_y[l]u_y$
(7)        $f^{eq} = w(l)(\rho + \frac{3}{2}(u_x^2 + u_y^2) + 3a + \frac{9}{2}a^2)$
(8)        $f'[i, j, l] = (1 - \omega)f'[i, j, l] + \omega f^{eq}$
(9)      endfor
(10)   endfor
(11) endfor
```

Here, *getDensityAndVelocity*(i, j) is a function to compute the density and velocity for a single cell. As described above, these values can be computed by summation over all the DFs of a cell:

getDensityVelocity(i, j)

```
(1) $\rho = u_x = u_y = u_z = 0$
(2) for $l = 0$ to 8 do
(3)   $\rho += f'[i, j, l]$
(4)   $u_x += e_x[l] \cdot f'[i, j, l]$
(5)   $u_y += e_y[l] \cdot f'[i, j, l]$
(6) endfor
(9) return$(\rho, u_x, u_y)$
```

Once the Stream and Collide steps are completed, the f' grid contains the updated state of the fluid for the next time step. Typically, the two grids are now swapped, and subsequent time steps alternate in streaming and colliding the DFs from one grid array to the other. Performing these two very simple steps already computes a solution of the Navier-Stokes equations. Note that using Eq. (10.7) the DFs for obstacle cells are never touched. For a 3D implementation, it is only necessary to loop over a third component, and include all 19 DFs in the inner loops over l (instead of only 9). The operations themselves are the same in 2D and 3D.

In contrast to a standard finite-difference NS solver, the implementation is much simpler, but also requires more memory. A typical NS solver as described in Section 10.1 usually requires at least 7 floating point values for each grid point (pressure, three velocity components, plus three temporary variables), but for some cases it might need higher resolutions to resolve obstacles with the same accuracy. Using a more sophisticated LB implementation with *grid compression* [PKW$^+$03], the memory requirements can be reduced to almost half of the usual requirements. Furthermore, using an adaptive time step size is common practice for a NS solver, while the size of the time step in the LBM is, by default, fixed to 1 (however, [TPR$^+$06] explains how to achieve the effect of a changing time step for a LBM solver). As the maximum lattice velocity may not exceed $1/3$, in order for the LBM to remain stable, it might still need several time steps to advance to the same time a NS solver would reach in a single step. However, each of these time steps usually requires a significantly smaller amount of work, as the LBM can be computed very efficiently on modern CPUs. Moreover, it does not require additional global computations such as the pressure correction step.

10.5 Stability

In order to simulate turbulent flows with the LBM, the basic algorithm needs to be extended, as its stability is limited once the relaxation parameter τ approaches $1/2$ (which is equivalent to ω being close to 2). Here, the Smagorinsky sub-grid model, as used in, e.g., [WZF$^+$03, LWK03], will be applied. Its primary use is to stabilize the simulation, instead of relying on its ability to accurately model subgrid scale vortices in the simulation. The model requires slightly more computations per cell than the standard LBM, but significantly increases stability which makes it a very useful addition. Note that in recent years, a variety of collision operators with increased accuracy and stability have been developed (e.g., multi relaxation time models and cascaded lattice boltzmann), but we have have found that the Smagorinsky model is easy to use and gives visually plausible results.

The sub-grid turbulence model applies the calculation of the local stress tensor as described in [Sma63] to the LBM. The computation of this tensor is relatively easy for the LBM, as each cell already contains information about the derivatives of the hydrodynamic variables in each DF. The magnitude of the stress tensor is then used in each cell to modify the relaxation time according to the so called *eddy viscosity*. For the calculation of the modified relaxation time, the Smagorinsky constant C is used. For the simulations in the following, C will be set to 0.03. Values in this range are commonly used for LB simulations, and were shown to yield good modeling of the sub-grid vortices [YGL05]. The turbulence model is integrated into the basic algorithm that was described in Section 10.3 by adding the calculation of the modified relaxation time after the streaming step, and using this value in the normal collision step.

The modified relaxation time τ_s is calculated by performing the steps that are described in the following. First, the non-equilibrium stress tensor $\Pi_{\alpha,\beta}$ is calculated for each cell with

$$\Pi_{\alpha,\beta} = \sum_{i=1}^{19} \mathbf{e}_{i_\alpha} \mathbf{e}_{i_\beta} \left(f_i - f_i^{eq} \right), \tag{10.8}$$

using the notation from [HSCD96]. Thus, α and β each run over the three spatial dimensions, while i is the index of the respective velocity vector and DF for the D3Q19 model.

The viscosity correction uses an intermediate value S which is computed as

$$S = \frac{1}{6C^2}\left(\sqrt{\nu^2 + 18C^2\sqrt{\Pi_{\alpha,\beta}\Pi_{\alpha,\beta}}} - \nu\right) . \tag{10.9}$$

Now the modified relaxation time is given by

$$\tau_s = 3(\nu + C^2 S) + \frac{1}{2}. \tag{10.10}$$

From Eq. (10.9) it can be seen that S will always have a positive value – thus the local viscosity will be increased depending on the size of the stress tensor calculated from the non-equilibrium parts of the distribution functions of the cell to be relaxed. This effectively removes instabilities due to small values of τ. We have found that with values of C around 0.03 the LBM simulations are removed of all stability problems, if it is ensured that the velocities lie in the allowed range ($|\mathbf{v}| < 1/3$).

Chapter 11

Shallow Water Equations

Nils Thuerey, Peter Hess

11.1 Introduction

The *shallow water equations* (SWE) are a simplified version of the more general *Navier-Stokes* (NS) equations, which are commonly used to describe the motion of fluids. The SWE reduce the problem of a three-dimensional fluid motion to a two-dimensional description with a height-field representation. From now on, we will use the following notation (it is also illustrated in Fig. 11.1):

- h denotes the height of the fluid above zero-level.
- g is the height of the ground below the fluid (above zero-level).
- η denotes the height of the fluid above ground, $\eta = h - g$.
- $\mathbf{v}$ the velocity of the fluid in the horizontal plane.

A basic version of the SWE can be written as

$$\frac{\partial \eta}{\partial t} + (\nabla \eta)\mathbf{v} = -\eta \nabla \cdot \mathbf{v} \tag{11.1}$$

$$\frac{\partial \mathbf{v}}{\partial t} + (\nabla \mathbf{v})\mathbf{v} = a_n \nabla h\,, \tag{11.2}$$

where a_n denotes a vertical acceleration of the fluid, e.g., due to gravity. This formulation can be derived from the NS equations by, most importantly, assuming a hydrostatic pressure along the direction of gravity. Interested readers can find a detailed derivation of these euqations in Section A.

In the following sections we will first explain how to solve these equations with a basic solver, and then extend this solver with more advanced techniques to handle open boundaries, or free surfaces.

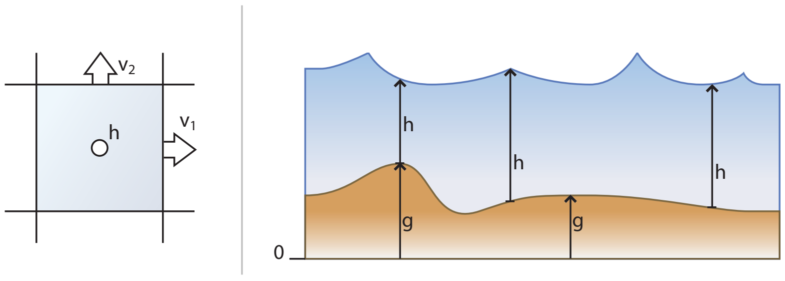


Figure 11.1: A fluid volume is represented as a heightfield elevation in normal direction **n**. The fluid velocity **v** has two components in horizontal directions.

11.2 A basic Solver

A basic solver of the SWE has to compute the change of the height and velocity values of the water surface over time. According to Eq. (11.2) and Eq. (11.1) the equations for the water height η and the velocity $\mathbf{v} = (v_1, v_2)$ can be written as:

$$\begin{aligned} \partial\eta/\partial t + (\nabla\eta)\mathbf{v} &= -\eta\nabla\cdot\mathbf{v} \\ \partial v_1/\partial t + (\nabla v_1)\mathbf{v} &= a_n\nabla h \\ \partial v_2/\partial t + (\nabla v_2)\mathbf{v} &= a_n\nabla h\ . \end{aligned} \tag{11.3}$$

In this form, the two distinct parts of the equations can be identified: the left side accounts for the advection within the velocity field **v**, while the right side computes an additional acceleration term. Here, we will use an explicit time integration scheme, as this makes the solution of the SWE signficantly more simple. Alternatively, implicit schemes, as described in [LvdP02] could be used. These, however, require a system of linear equations to be solved for the update, and, with simpler methods such as implicit Euler, introduce a significant amount of damping.

To compute a solution for these equations, we first discretize the domain with n_1 cells in x-direction, and n_2 cells in y-direction. For simplicity, we assume in the following that the cells have a square form with side length Δx. The gravity force a_n is assumed to act along the z-axis, e.g. $a_n = 10$, and the size of a single time step is given by Δt. The overall height of the water, and the strength of the gravity will later on determine the speed of the waves travelling on the surface. To represent the three unknowns with this grid we use a staggered grid. This means that the pressure is located in the center of a cell, while the velocity components are located at the center of each edge, as shown in Fig. 11.1. The staggered grid is commonly used for fluid solvers, and prevents instabilities that would result from a discretization on a co-located grid. An update step of the shallow water solver consisits of the following parts: first all three fields are advected with the current velocity field. Afterwards, the acceleration terms are computed for the height and velocity fields. The following pseudo-code illustrates a single step of the simulation loop of a simple shallow water solver:

```
Shallow-water-step($\eta$, $\mathbf{v}$, $g$)
(1) $\eta = \text{Advect}(\eta, \mathbf{v})$
(2) $v_1 = \text{Advect}(v_1, \mathbf{v})$
(3) $v_2 = \text{Advect}(v_2, \mathbf{v})$
(4) Update-height($\eta$, $\mathbf{v}$)
(5) $h = \eta' + g$
(6) Update-velocities($h, v_1, v_2$)
```

Note that the three calls of the *Advect*(...) function return a value that is assigned back to the original input grid (e.g., in line 1 η is a parameter of the call, and used in the assignment). This should indicate that the advection requires a temporary array, to which the advected values are written, and which is copied back to the original grid after finishing the advection step.

To compute the advection, we can use the semi-Lagrangian method [Sta99] to compute a solution without having to worry about stability. This algorithm computes the advection on a grid by essentially performing a backward trace of an imaginary particle at each grid location. Given a scalar field s to be advected, we have to compute a new value for a grid cell at position $\mathbf{x}$. This is done by tracing a particle at this position backward in time, where it had the position $\mathbf{x}^{t-1} = \mathbf{x} - \Delta t \mathbf{v}(\mathbf{x})$. We now update the value of s with the value at $\mathbf{x}^{t-1}$, so the new value is given by $s(\mathbf{x})' = s(\mathbf{x}^{t-1})$. Note that although $\mathbf{x}$ is either the center or edge of a cell in our case, $\mathbf{x}'$ can be located anywhere in the grid, and thus usually requires an interpolation to compute the value of s there. This is typically done with a bi-linear interpolation, to ensure stability. It guarantees that the interpolated value is bounded by its source values from the grid, while any form of higher order interpolation could result in larger or smaller values, and thus cause stability problems. The advection step can be formulated as

```
Advect($s$, $\mathbf{v}$)
(1) for $j = 1$ to $n_2 - 1$
(2)     for $i = 1$ to $n_1 - 1$
(3)         $\mathbf{x} = (i \cdot \Delta x, j \cdot \Delta x)$
(4)         $\mathbf{x}' = \mathbf{x} - \Delta t \cdot v(\mathbf{x})$
(5)         $s'(i, j) = \text{interpolate}(s, \mathbf{x}')$
(6)     endfor
(7) endfor
(8) return($s'$)
```

Note that, due to the staggered grid, the lookup of $\mathbf{v}(\mathbf{x})$ above already might require an averaging of two neighboring velocity components to compute the velocity at the desired position. This is also, why the three advection steps cannot directly performed together - each of them requires slightly different velocity interpolations, and leads to different offsets in the grid for interpolation.

The divergence of the velocity field for the fluid height update can be easily computed with finite differences on the staggered grid. So, according to Eq. (11.3), the height update is given by

```
Update-height($\eta, \mathbf{v}$)
(1)  for $j = 1$ to $n_2 - 1$
(2)      for $i = 1$ to $n_1 - 1$
(3)          $\eta(i,j) -= \eta(i,j) \cdot \left( \frac{(v_1(i+1,j) - v_1(i,j))}{\Delta x} + \frac{(v_2(i,j+1) - v_2(i,j))}{\Delta x} \right) \Delta t$
(5)      endfor
(6)  endfor
(7)  return($\eta'$)
```

In contrast to the advection steps, adding the accelerations can be directly done on the input grids. Similarly, the acceleration term for the velocity update is given by the gradient of the overall fluid height. Note that in this case, the total height above the zero-level is used instead of the fluid height above the ground level. This is necessary, to, e.g., induce an acceleration of the fluid on an inclined plane, even when the fluid height itself is constant (all derivatives of η would be zero in this case). The parameter a for the velocity update below is the gravity force.

```
Update-velocities($h, v_1, v_2, a$)
(1)  for $j = 1$ to $n_2 - 1$
(2)      for $i = 2$ to $n_1 - 1$
(3)          $v_1(i,j) += a(\frac{h(i-1,j) - h(i,j)}{\Delta x})\Delta t$
(5)      endfor
(6)  endfor
(7)  for $j = 2$ to $n_2 - 1$
(8)      for $i = 1$ to $n_1 - 1$
(9)          $v_2(i,j) += a(\frac{h(i,j-1) - h(i,j)}{\Delta x})\Delta t$
(10)     endfor
(11) endfor
```

This concludes a single step of a basic shallow water solver. Note that the steps so far do not update the values at the boundary of the simulation domain, as we cannot compute any derivatives there. Instead, special boundary conditions are required at the border, and can be used to achieve a variety of effects. These will be the topic of the next section.

11.3 Boundary Conditions

In the following, we will describe different types of boundary conditions: reflecting and absorsorbing boundaries, as well as a form of free surface boundary conditions. The former can be used to model a wall that reflects incoming waves. The second type can be used to give the effect of an open water surface, as waves will simply leave the computational domain. Free surface boundary conditions can be used once the fluid should, e.g., flow through a landscape. Although the boundary conditions will be described to handle the outermost region of the computational domain, they can likewise be used to, e.g., create a

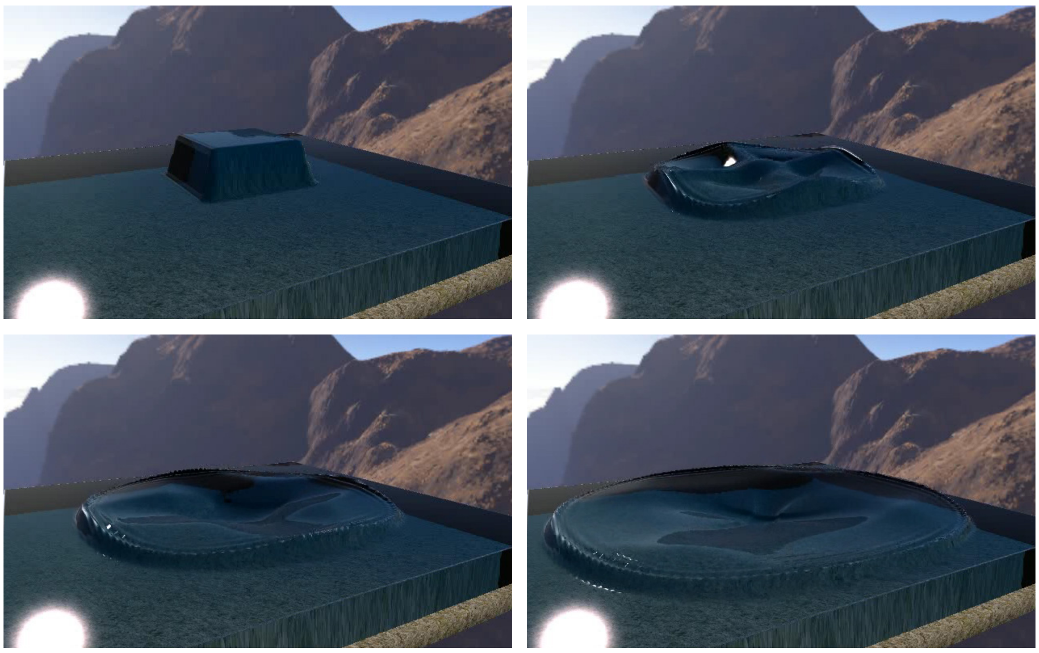


Figure 11.2: Example of a wave spreading in a basic shallow water simulation.

wall in the middle of the domain. We will not consider periodic boundary conditions here. They are commonly used in engineering applications, but give the visually unnatural effect that a wave leaving the domain at the right side re-enters it at the left. The results shown in this section where created by Peter Hess [Hes07].

Reflecting Boundaries

In the following, we will, without loss of generality, consider the boundary conditions for cells at the left boundary. Reflecting boundary conditions are achieved by setting the velocities at the boundary to zero (after all, there should be no flux through the wall). In addition, we mirror the height of the fluid in the outermost layer. We thus set:

$$\begin{aligned} h(0,j)' &= h(1,j) \\ v_1(1,j)' &= 0 \\ v_2(0,j)' &= 0\,. \end{aligned} \tag{11.4}$$

Note that we do not modify the y-component v_2 of the velocity field. The fluid is thus allowed to move tangentially to a wall. Theoretically, we could also enforce different behaviors for the tangential velocities, but in practice this does not make a noticeable difference. Also note, that we only set $v_1(1,j)$, as $v_1(0,j)$ is usually never accessed during an computation step.

Absorbing Boundaries

Surprisingly, it is more difficult to achieve absorbing boundaries than reflecting ones. The problem of boundaries simulating an infinite domain is already known for a long time (see, e.g., [Dur01] for details). A commonly used method to achieve this, is the *perfectly matched layer* introduced by [Ber94], requires an additional layer of computations around the actual domain.

This is why we chose to use the Higdon boundary conditions [Hig94] which are less accurate but can be more efficiently computed than PML. Below is the p$^{\text{th}}$ order Higdon boundary condition, where the velocities c_j are chosen to span the range of incoming wave velocities.

$$\left(\prod_{j=1}^{p}\left(\frac{\partial}{\partial t}+c_j\frac{\partial}{\partial x}\right)\right)h=0 \tag{11.5}$$

This boundary condition can be problematic for higher order approximations, but as the wave propagation speed in shallow water is known to be $c=\sqrt{g\eta}$, this allows us to use to use the 1$^{\text{st}}$ order boundary condition

$$\left(\frac{\partial}{\partial t}+c\frac{\partial}{\partial x}\right)h=0\ . \tag{11.6}$$

This boundary condition actually requires temporal derivatives, so we assume the current heightfield is given by h^t, while the heights of the previous step are stored in h^{t-1}. Hence, we can set the boundary values to:

$$\begin{aligned} h(0,j)' &= \frac{\Delta x\, h(0,j)^{t-1}+\Delta t\ c(1,j)^t h(1,j)^t}{\Delta x+\Delta t\ c(1,j)^t} \\ v_1(1,j)' &= v_1(1,j)^{t-1}-a\frac{h(1,j)^t-h(0,j)^t}{\Delta x}\Delta t \\ v_2(1,j)' &= 0 \end{aligned} \tag{11.7}$$

Note that the update of v_1 is essentially the same acceleration term on the left hand side of Eq. (A.16). To further suppress any residual reflections at the boundary, we can apply a slight damping of the height field in a layer around the boundary.

Fig. 11.3 shows the effect of these boundary conditions compared to reflecting ones.

For boundaries where fluid should flow into or out of the domain, we can reuse the two types above. Inflow boundary conditions can be achieved by specifiying reflecting ones, with an additional fixed normal velocity. For outflow boundary conditions, absorbing ones with free normal velocities are more suitable.

Free Surfaces

Often, shallow water simulations assume a completely fluid domain, since this makes solving the SWE quite straightforward. Once applications like a river, or fluid filling an arbitrary terrain are needed, this is not sufficient anymore. Such applications require a distinction between areas filled with fluid, and empty or dry areas. An example can be seen in Fig. 11.4. In the following we will consider this as a problem similar to free surface handling for full fluid simulations. Shallow water simulations naturally have an interface, and thus a free

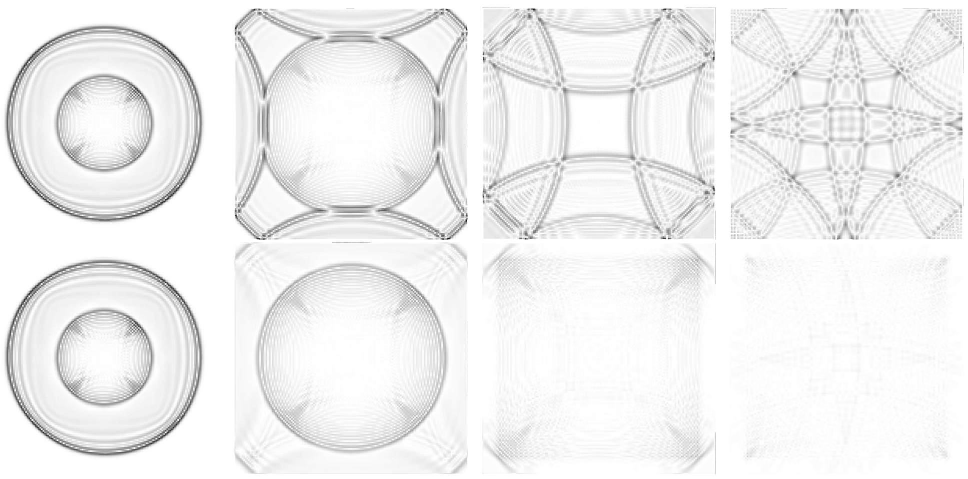


Figure 11.3: A comparison between reflecting boundary conditions (upper row of pictures), and absorbing ones (lower row).

surface, in the simulation plane between fluid below and a second gas phase above the fluid. In addition, we will now prescribe boundary conditions with such a free surface within the simulation plane itself. From the mathematical point of view, a distinction between fluid and dry would not be necessary since the SWE still work if η is zero. Distinguishing fluid and dry cells, however, brings some advantages. Foremost, computational time can be saved if large parts of the domain are dry. Therefore, we introduce cell flags $f(i,j)$, that determine the type of each cell, and update them once per time step after updating the heights. This allows us to quickly identify wet and dry cells. Besides the computational advantage, controlling the transition between wet and dry cells also gives us some control over the spreading velocity. Without free surface tracking, the fluid boundary would expand exactly one cell per time step, regardless of cell size and time step length. The height of this advancing boundary would be very small, but this behavior is usually not desired. In addition, we will compute a fill value r for each cell, as this allows us to track a smoothly moving surface line between the fluid and empty cells.

To determine the cell's flag f we have to compute the minimal and maximal ground level h_{min} and h_{max} as well as the maximal fluid depth η_{max} on the cell's edges.

$$h_{min}(i,j) = \frac{h(i,j) + \min\ h(\mathbf{p})}{2} \qquad \mathbf{p} \in \mathcal{N}(i,j) \tag{11.8}$$

$$h_{max}(i,j) = \frac{h(i,j) + \max\ h(\mathbf{p})}{2} + \varepsilon_H \qquad \mathbf{p} \in \mathcal{N}(i,j) \tag{11.9}$$

$$\eta_{max}(i,j) = \frac{\eta(i,j) + \max\ \eta(\mathbf{p})}{2} \qquad \mathbf{p} \in \mathcal{N}(i,j) \tag{11.10}$$

where $\mathcal{N}(i,j)$ is the set of the four direct neighbors of cell (i,j). Note that we add a small value ε_h to h_{max} to prevent h_{min} to be equal to h_{max} in flat areas. With these three values we can now determine f as well as the fill ratio r which indicates the cell's fill level in dependence of the local ground topology $h_{min}(i,j)$ and $h_{max}(i,j)$. r can be used to compute

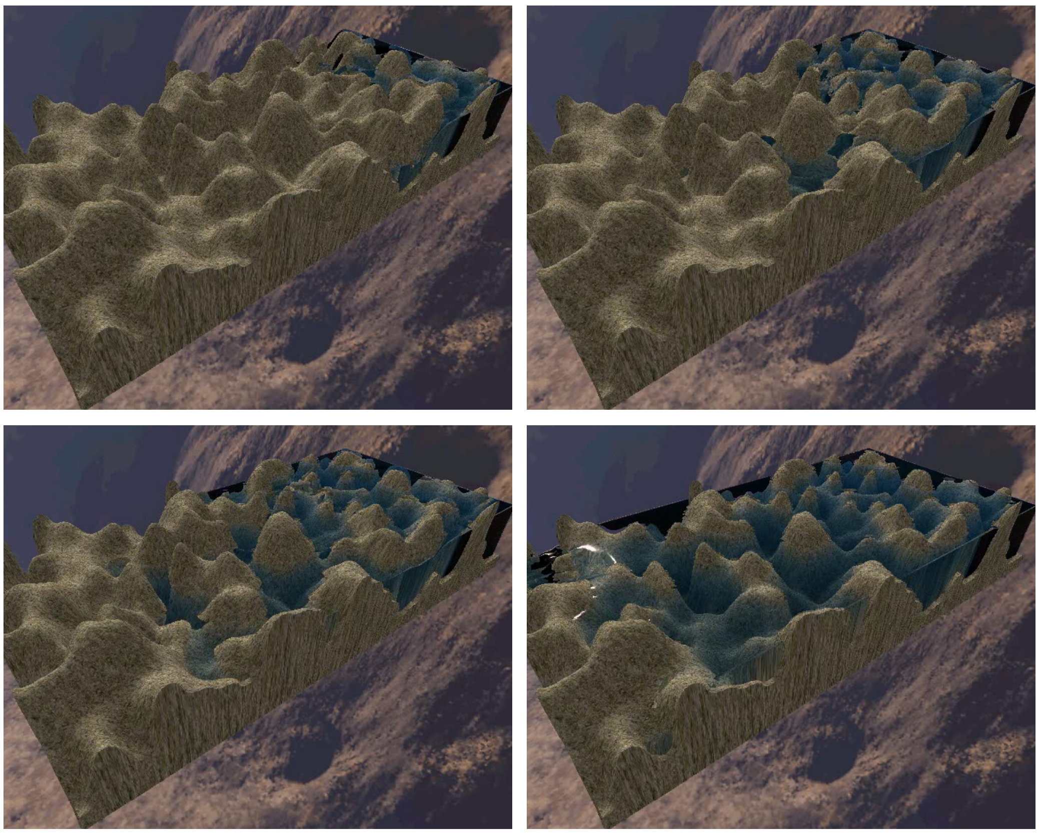


Figure 11.4: A shallow water simulation with free surface boundary conditions fills a terrain.

an isoline which defines the border of the rendered fluid surface for rendering the water surface. The following pseudo code shows how f and r are calculated:

```
Compute-flags$(i,j)$
(1)  if $h(i,j) \leq h_{min}(i,j)$ and $\eta_{max}(i,j) < \varepsilon_{\eta_{max}}$
(2)      $f(i,j) = DRY$
(3)      $r(i,j) = 0$
(4)  else if $h(i,j) > h_{max}$
(5)      $f(i,j) = FLUID$
(6)      $r(i,j) = 1$
(7)  else
(8)      $f(i,j) = FLUID$
(9)      $r(i,j) = \Big(h(i,j) - h_{min}(i,j)\Big) / \Big(h_{max}(i,j) - h_{min}(i,j)\Big)$
(10) endif
```

A cell is marked as dry if its surface height is not higher than the lowest ground value in the cell and if there is no neighbor cell from which fluid could flow into this cell. The fill

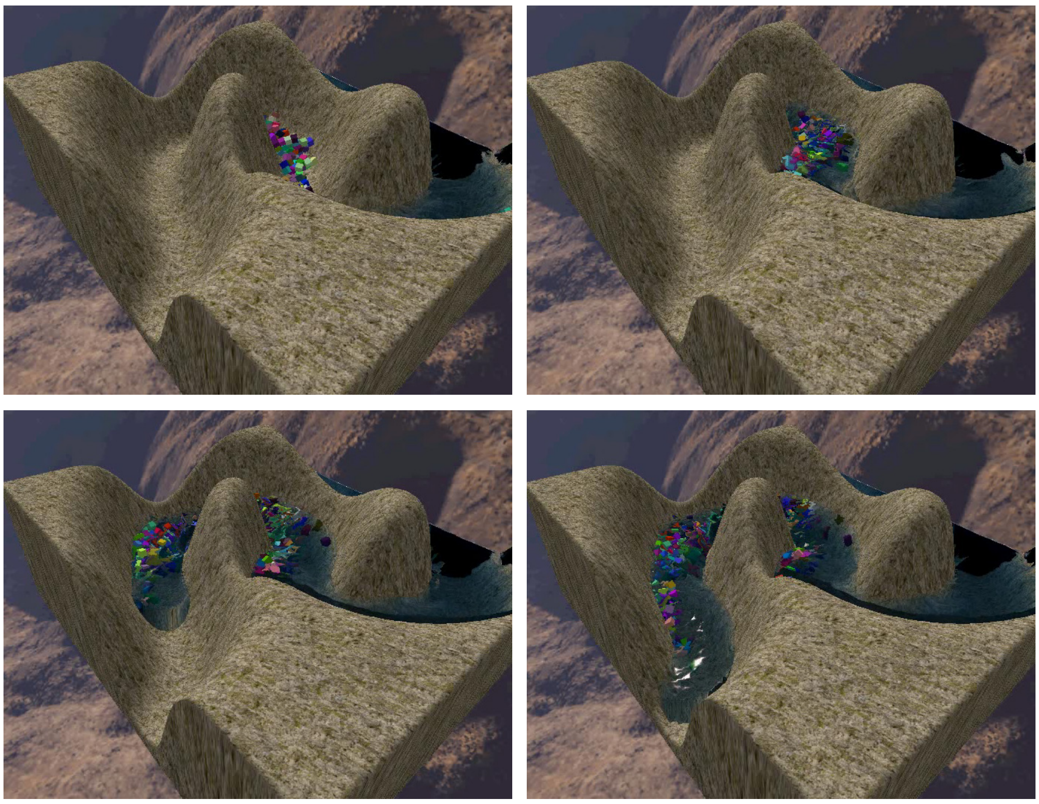


Figure 11.5: A wave flows through an S-shaped river bed, and pushes a large number of rigid bodies along with the flow.

ratio is then set to zero. ε_η can be seen as a threshold which allows inflow from a neighbor cell only if this neighbor has a large enough amount of fluid. This effectively limits the spread of thin layers of fluid. So this could be seen as a simple way of simulating surface tension. A cell is completely filled if its surface height is higher than the ground at any position in the cell. The fill ratio is then set to one. The cell is also marked as fluid if the surface height is only in parts higher than the ground level. In this case however the fill ratio is the ratio between minimal ground level, fluid surface height and maximal ground level.

Note, that with this definition cells may have negative depth values η even if they are marked as fluid. There are cases were the cell center itself is dry, so the value of η is negative at this point, while the whole cell still contains fluid at the edges of a cell.

Part I

Appendix

Appendix A

Derivation of the Shallow Water Equations

Nils Thuerey, Roland Angst

In this chapter we go through a detailed derivation of a more general form of the *shallow water equations* (SWE) from the underlying *Navier-Stokes* (NS) equations. It is based on [Ang07].

A.1 Integral Form

The SWE are a specialized and simplified version of the Navier-Stokes equations. The Navier-Stokes equations describe the dynamics of a fluid much like elasticity theory describes the dynamics of deformable solids. The Navier-Stokes equations relate the dynamics of the density field, the temperature field and the velocity field of a fluid to each other. The full Navier-Stokes equations in three dimensions are a system of partial differential equations with five equations and unknowns (density, velocity and temperature). The Navier-Stokes equations as well as the SWE are derived from three balance equations which prescribe three conservation laws, namely the conservation of mass, momentum and energy. In addition, we will make use of the material specific constitutive relations for the derivation. A common simplification is to assume that the fluid flow is incompressible which means that the fluid density is spatially constant. Likewise, we assume that the viscosity coefficient, and the thermal conductivity are uniform.

This assumption has important implications:

- The density is no longer an unknown of the equation system. Instead, the pressure becomes an unknown.
- The pressure and the velocity field are decoupled from the energy conservation. This means that we can solve for the pressure and the velocity ignoring the temperature field. We thus assume an isotermal fluid.

These first assumptions reduce the Navier-Stokes equations to four equations in four unknowns: three velocity components and the pressure.

Balance Equations

The balance equations describe fundamental physical laws and are valid for any system described with continuum mechanics. Hence, they are the same for the Navier-Stokes equations and for the SWE. Since we assume that the fluid is incompressible and isothermal, the conservation of mass and the conservation of momentum are the only balance equations required for the derivation. The continuity equation or conservation of mass for a domain Ω with the boundary $\partial\Omega$ with density ρ reads like

$$\frac{d}{dt}\iiint_{\Omega^3} \rho dV + \iint_{\partial\Omega^3} \rho \mathbf{v}\cdot\mathbf{n} dA = 0\ . \tag{A.1}$$

It means the temporal change of the fluid mass in the domain, which is computed by the integral of the density over the fluid volume (the first term), is given by the flux along its boundary (the second term). Similarly, the conservation of momentum looks like

$$\frac{d}{dt}\iiint_{\Omega^3} \rho \mathbf{v} dV + \iint_{\partial\Omega^3} \rho \mathbf{v}\mathbf{v}^T\cdot\mathbf{n} dA \iint_{\partial\Omega^3} \sigma\cdot\mathbf{n} dA = \iiint_{\Omega^3} \mathbf{f} dV \tag{A.2}$$

where $\mathbf{v}$ is the fluid velocity and $\mathbf{f}$ captures sources and sinks of the fluid. This last term is also known as body force and is a force density, in $[N/m^3]$, and the actual force acting on a given volume V is thus given by the volume integral over this force density. σ denotes the stress tensor which is the sum of the pressure p and the viscous stress tensor $\mathbf{T}$. Note that viscous forces are caused by spatial velocity differences in the fluid. We will assume that there are only small velocity variations, so that the viscosity coefficient is small. In addition we will ignore the viscous stress tensor. This assumption justified by observing that water-like fluids have a very small viscosity and that the numerical integration introduces a certain amount of damping, which basically has the same effect as a viscosity larger than zero. Typically, this numerical viscosity is larger than that of water so that we rely on the solver to compute a fluid motion with a viscosity as small as possible. Essentially, we solve for an inviscid fluid.

Projection

The derivation of the SWE now proceeds by making an important assumption: the pressure is assumed to be hydrostatic, i.e.,

$$p = \rho a_n \Delta\eta \tag{A.3}$$

where a_n is the acceleration in vertical direction and $\Delta\eta$ corresponds to the vertical distance from the bottom of the fluid to its surface. Amongst others, the vertical acceleration takes care of the gravitational acceleration $g = -9.81\frac{m}{s^2}$. This hydrostatic pressure equation can be derived by assuming that the vertical velocity is much smaller than the horizontal velocity such that the momentum conservation in vertical direction is dominated by the pressure term and hence the advection of the vertical velocity v_n can safely be ignored. This assumption is justified by observing that in areas where the fluid is shallow, the vertical velocity is indeed orders of magnitude smaller than the horizontal velocity components. The implications of the hydrostatic pressure assumption are:

- The momentum equation for the vertical component is static, i.e., the vertical velocity is temporally constant and equal to zero.
- The velocity is basically a two dimensional tangential vector field which is orthogonal to the vertical (or normal) direction.
- The pressure p acts as the sole vertical variable.
- The fluid surface can be represented by a heightfield.

The assumption of a hydrostatic pressure thus allows us to further reduce the number of unknowns from four to three - the fluid can now be described by velocities in the horizontal domain, and a height of the fluid.

The tangential velocity can be thought of as averaged over the fluid depth. The momentum conservation equation therefore reads

$$\begin{aligned}\iiint_{\Omega^3}\begin{pmatrix}\mathbf{f}_\tau\\ f_n\end{pmatrix}dV \quad &= \quad \frac{d}{dt}\iiint_{\Omega^3}\rho\begin{pmatrix}\mathbf{v}_\tau\\ 0\end{pmatrix}dV\\ &\quad +\iint_{\partial\Omega^3}\rho\begin{pmatrix}\mathbf{v}_\tau\\ 0\end{pmatrix}\begin{pmatrix}\mathbf{v}_\tau\\ 0\end{pmatrix}^T\cdot\mathbf{n}dA+\iint_{\partial\Omega^3}p\mathbf{I}\cdot\mathbf{n}dA \qquad (A.4)\end{aligned}$$

where the subscript τ indicates horizontal (or tangential) components and the subscript n denotes the vertical (or normal) component. The following notation is introduced and will be used throughout all the chapters of this chapter:

- H: Height of fluid between zero-level and ground.
- h: Height of fluid above zero-level.
- η: Total fluid height, i.e. $\eta = H + h$.

To achieve a heightfield representation of the fluid we have to get rid of the vertical dimension in these equations by a projection of the integrals to the two dimensional horizontal domain. Assume the equations are computed over a fluid column with a square base area Ω^2_τ. Consider first the surface integral in the continuity equation

$$\iint_{\partial\Omega^3}\rho\mathbf{v}\cdot\mathbf{n}dA=\iint_{\Omega^2_\tau}\rho\mathbf{v}\cdot\mathbf{n}dA+\iint_{\Omega^2_n}\rho\mathbf{v}\cdot\mathbf{n}dA$$

where the surface integral was split up into two parts over the surfaces Ω^2_n vertical to the horizontal projection plane Ω^2_τ and the remaining bottom and top surface. There is no mass flux through the bottom because the bottom surface is impermeable and thus this surface integral vanishes. The slope of the fluid surface is assumed to be small. This implies that $\mathbf{v}\cdot\mathbf{n}$ is small, too and hence, the integral over the top of the fluid column is negligible. The mass flux through the vertical sides Ω^2_n is reduced to a line integral by projecting the fluid depth onto the boundaries $\partial\Omega^2_\tau$ of the horizontal projection surface

$$\iint_{\Omega^2_n}\rho\mathbf{v}\cdot\mathbf{n}dA=\int_{\partial\Omega^2_\tau}\int_{-H}^{h}\rho\mathbf{v}\cdot\mathbf{n}d\eta ds=\int_{\partial\Omega^2_\tau}\rho(h+H)\mathbf{v}\cdot\mathbf{n}ds=\int_{\partial\Omega^2_\tau}\rho\eta\mathbf{v}\cdot\mathbf{n}ds. \qquad (A.5)$$

Note that the normals of the vertical surfaces Ω_n^2 of the fluid column are equal to the normals of the boundary curve of the horizontal projection surface. The projection of the momentum advection term is analog to the projection of the mass flux, i.e., the integrals over the non-vertical areas vanish due to the same reasons.

Consider now the pressure force acting on the fluid column

$$\iint_{\partial\Omega^3} p\mathbf{I}\cdot\mathbf{n}dA = \iint_{\Omega_\tau^2} p\mathbf{I}\cdot\mathbf{n}dA + \iint_{\Omega_n^2} p\mathbf{I}\cdot\mathbf{n}dA$$

where the surface integral was again split up into a horizontal and a vertical component. The surface integral over the top of the fluid surface vanishes because the pressure at this interface is negligible compared to the pressure inside the fluid. The surface integral over the bottom leads to a source term of the momentum, provided the bottom is non-flat. This pressure force makes the fluid flow downwards. This force is computed by integrating the hydrostatic pressure $p = \rho g\Delta\eta$ over the bottom area

$$\begin{aligned}\iint_{\Omega_\tau^2} p\mathbf{I}\cdot\mathbf{n}dA &= \iint_{\Omega_\tau^2} \rho a_n(-\eta)\mathbf{n}dA \\ &= \iint_{\Omega_\tau^2} \rho a_n(-H-h)(-\nabla H)dA = \iint_{\Omega_\tau^2} \rho a_n(H+h)\nabla H dA. \end{aligned} \tag{A.6}$$

Note that the normal $\mathbf{n}$ in this equation corresponds to the normal given by the negative gradient of the bottom heightfield $-\nabla H$ rather than the normal of the horizontal projection plane. Furthermore, the normalization of the gradient and the cosine for the area foreshortening cancel each other out.

The projection of the pressure force due to the vertical sides of the fluid column transforms the surface integral over these sides again into a boundary integral over the curve $\partial\Omega_\tau^2$, which bounds the horizontal area. Thus, by integrating over the fluid depth, the fluxes through the vertical sides get concentrated at the boundary curve $\partial\Omega_\tau^2$

$$\begin{aligned}\iint_{\Omega_n^2} p\mathbf{I}\cdot\mathbf{n}dA &= \int_{\partial\Omega_\tau^2}\int_{-H}^{h} \rho a_n(\eta-h)\mathbf{n}d\eta ds \\ &= -\int_{\partial\Omega_\tau^2} \frac{1}{2}\rho a_n(-H-h)^2\mathbf{n}ds = -\int_{\partial\Omega_\tau^2} \frac{1}{2}\rho a_n\eta^2\mathbf{n}ds. \end{aligned} \tag{A.7}$$

Integrating the vertical component in the volume integrals of Eq. (A.1) and Eq. (A.4) and inserting Eq. (A.5), Eq. (A.6), and Eq. (A.7) the SWE can be restated in the integral form

$$\frac{d}{dt}\iint_{\Omega_\tau^2} \rho\eta dA + \int_{\partial\Omega_\tau^2} \rho\eta\mathbf{v}\cdot\mathbf{n}ds = 0 \tag{A.8}$$

$$\begin{aligned}\frac{d}{dt}\iint_{\Omega_\tau^2} \rho\eta\mathbf{v}dA + \int_{\partial\Omega_\tau^2} \rho\eta\mathbf{v}\mathbf{v}^T\cdot\mathbf{n}ds - \int_{\partial\Omega_\tau^2} \frac{1}{2}\rho a_n\eta^2\mathbf{n}ds \\ = -\iint_{\Omega_\tau^2} \rho a_n\eta\nabla H dA + \iint_{\Omega_\tau^2} \eta\mathbf{f}_\tau dA. \end{aligned} \tag{A.9}$$

From now on, the subscript τ will be omitted in the equations. In the literature, these equations are often given in the so called flux form

$$\frac{d}{dt}\iint_{\Omega^2} \mathbf{q}dA + \int_{\partial\Omega^2} F(\mathbf{q})\cdot\mathbf{n}ds = \iint_{\Omega^2} \psi dA \tag{A.10}$$

where the equations were divided by the constant density and

$$\mathbf{q} = \begin{pmatrix}\eta & \eta u & \eta v\end{pmatrix}^T$$
$$F(\mathbf{q}) = \begin{bmatrix}\mathbf{f}(\mathbf{q}) & \mathbf{g}(\mathbf{q})\end{bmatrix} = \begin{bmatrix} \eta u & \eta v \\ \eta u^2 - \frac{1}{2}a_n\eta^2 & \eta uv \\ \eta uv & \eta v^2 - \frac{1}{2}a_n\eta^2 \end{bmatrix}$$
$$\psi = \begin{pmatrix} 0 \\ -a_n\eta\nabla H + \frac{\eta}{\rho}\mathbf{f}_\tau \end{pmatrix}.$$

The two components of the velocity are denoted here as $\mathbf{v} = (u,v)^T$. This notation clearly shows the non-linearity and the conservative nature of the equations due to the flux across the boundary.

A.2 Differential Form

We now turn to the derivation of the differential form of the SWE, although the integral form admits more general solutions than the differential form. Solutions to the integral form may exhibit discontinuities (which must still satisfy certain conditions) like for example shocks. Such solutions are also known as weak solutions. The differential form on the other hand is written as a partial differential equation (PDE) which only admits continuous solutions because the derivation requires a continuity assumption. Solutions of the differential form are known as strong (also called pointwise) solutions. But note that the true solution of the SWE with certain boundary conditions might indeed be discontinuous and in such cases, a solution method derived from the differential form fails to converge to the correct solution. However, in computer graphics, accuracy is not the most important factor and thus, a strong solution which is similar to the true weak solution might be adequate enough.

As mentioned before, we will assume in this derivation that the solution to the SWE is continuous and differentiable. The divergence theorem can then be applied to the boundary integral of the integral form A.8 and A.9 and the time derivative can be moved inside the integral. We end up with

$$\iint_{\Omega^2} \mathbf{q}_t dA + \iint_{\Omega^2} \nabla \cdot \mathbf{F}(\mathbf{q}) dA = \iint_{\Omega^2} \psi dA.$$

This equation must be valid for any two dimensional domain Ω^2 and therefore the following PDE must hold

$$\mathbf{q}_t + \nabla \cdot \mathbf{F}(\mathbf{q}) = \mathbf{q}_t + \frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\mathbf{q}_x + \frac{\partial \mathbf{g}(\mathbf{q})}{\partial \mathbf{q}}\mathbf{q}_y = \psi. \tag{A.11}$$

The chain-rule was used to expand the divergence into a product involving the Jacobian of the flux functions **f** and **g**. Note that such a PDE is called hyperbolic if any linear combination of the two Jacobians $\frac{\partial \mathbf{f}}{\partial \mathbf{q}}$ and $\frac{\partial \mathbf{g}}{\partial \mathbf{q}}$ has real eigenvalues and can be diagonalized. This is indeed the case for the SWE.

Written out in detail, Eq. (A.11) looks like

$$\eta_t = -\nabla \cdot (\eta \mathbf{v}) \tag{A.12}$$
$$\frac{\partial \eta \mathbf{v}}{\partial dt} + \nabla \cdot (\eta \mathbf{v}\mathbf{v}^T - \mathbf{I}\frac{1}{2}a_n\eta^2) = -a_n\eta\nabla H + \frac{\eta}{\rho}\mathbf{f}_\tau. \tag{A.13}$$

This latter equation can be simplified even further. By pushing the differential operator inside the bracketed terms we get

$$\eta\mathbf{v}_t + \mathbf{v}\eta_t + \mathbf{v}\nabla\cdot(\eta\mathbf{v}) + \eta\mathbf{v}\cdot\nabla\mathbf{u} - a_n\eta\nabla\eta - \frac{1}{2}\eta^2\nabla a_n = -a_n\eta\nabla H + \frac{\eta}{\rho}\mathbf{f}_\tau.$$

The second and third term on the left hand side correspond to the continuity equation times the velocity and thus is equal to zero. Dividing by η and observing that $H-\eta=-h$ finally gives

$$\mathbf{v}_t + \mathbf{v}\cdot\nabla\mathbf{v} - a_n\nabla h - \frac{1}{2}\eta\nabla a_n = \frac{1}{\rho}\mathbf{f}_\tau \tag{A.14}$$

The previous equation clearly highlights the advectional part of the momentum conservation. The differential equation derived from the continuity equation can also be formulated using a material derivative by observing that $\nabla\cdot(\eta\mathbf{v}) = \mathbf{v}\cdot\nabla\eta + \eta\nabla\cdot\mathbf{v}$. The SWE then take the following form

$$\frac{D\eta}{Dt} = -\eta\nabla\cdot\mathbf{v} \tag{A.15}$$

$$\frac{D\mathbf{v}}{Dt} = a_n\nabla h + \frac{1}{2}\eta\nabla a_n + \frac{1}{\rho}\mathbf{f}_\tau. \tag{A.16}$$

Note that when the balance equations were projected from three dimension to two dimensions, the vertical force component has been dropped. This term can now be reinserted as a vertical acceleration component, for example in addition to a constant acceleration due to gravity $g = -9.81\frac{m}{s^2}$:

$$a_n = g + \frac{f_n}{\rho}$$

Thanks to these external force terms $\mathbf{f}_\tau$ and f_n, the SWE can more easily interact with other objects. Note that the SWE are usually derived without these terms.

Bibliography

[AN05] Alexis Angelidis and Fabrice Neyret. Simulation of smoke based on vortex filament primitives. *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 87–96, 2005.

[Ang07] Roland Angst. Control Algorithms for Interactively Animated Fluid Characters. Master Thesis, Computer Graphics Laboratory, ETH Zurich, 2007.

[ANSN06] Alexis Angelidis, Fabrice Neyret, Karan Singh, and Derek Nowrouzezahrai. A controllable, fast and stable basis for vortex based smoke simulation. *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 25–32, 2006.

[BBB07] Christopher Batty, Florence Bertails, and Robert Bridson. A fast variational framework for accurate solid-fluid coupling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 100, New York, NY, USA, 2007. ACM Press.

[Ber94] Jean-Pierre Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *J. Comput. Phys.*, 114(2):185–200, 1994.

[BGK54] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. *Phys. Rev.*, 94:511–525, 1954.

[BT07] M. Becker and M. Teschner. Weakly Compressible SPH for Free Surface Flows. *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2007.

[CCM92] Hudong Chen, Shiyi Chen, and William H. Matthaeus. Recovery of the Navier-Stokes equations using a lattice-gas Boltzmann method. *Phys. Rev. A*, 45(8):R5339–R5342, 1992.

[CdVLHM97] Jim X Chen, Niels da Vitoria Lobo, Charles E. Hughes, and J. Michael Moshell. Real-time fluid simulation in a dynamic virtual environment. *IEEE Comput. Graph. Appl.*, 1997.

[CFL$^+$07] Nuttapong Chentanez, Bryan E. Feldman, François Labelle, James F. O'Brien, and Jonathan R. Shewchuk. Liquid simulation on lattice-based tetrahedral meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

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

[CMT04] Mark Carlson, Peter John Mucha, and Greg Turk. Rigid fluid: Animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3), 2004.

[Dur01] D.R. Durran. Open boundary conditions: Fact and fiction. In *IUTAM Symposium on Advances in Mathematical Modelling of Atmosphere and Ocean Dynamics*, pages 1–18. Kluwer Academic Publishers, 2001.

[EMF02] D. Enright, S. Marschner, and R. Fedkiw. Animation and Rendering of Complex Water Surfaces. *ACM Trans. Graph.*, 21(3):736–744, 2002.

[FdH$^+$87] Uriel Frisch, Dominique d'Humières, Brosl Hasslacher, Pierre Lallemand, Yves Pomeau, and Jean-Pierre Rivert. Lattice Gas Hydrodynamics in Two and Three Dimensions. *Complex Systems*, 1:649–707, 1987.

[FF01] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proc. of ACM SIGGRPAH*, pages 23–30, 2001.

[FM96] N. Foster and D. Metaxas. Realistic Animation of Liquids. *Graphical Models and Image Processing*, 58, 1996.

[FOA03] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. In *Proc. of ACM SIGGRAPH*, pages 708–715, Aug 2003.

[FOK05] Bryan E. Feldman, James F. O'Brien, and Bryan M. Klingner. Animating gases with hybrid meshes. *ACM Trans. Graph.*, 24(3):904–909, 2005.

[GHDS03] Eitan Grinspun, Anil Hirani, Mathieu Desbrun, and Peter Schroder. Discrete shells. In *Proceedings of the ACM SIGGRAPH Symposium on Computer Animation*, 2003.

[GHF$^+$07] Rony Goldenthal, David Harmon, Raanan Fattal, Michel Bercovier, and Eitan Grinspun. Efficient simulation of inextensible cloth. *ACM Trans. Graph.*, 26(3):49, 2007.

[GM97] S. F. Gibson and B. Mitrich. A survey of deformable models in computer graphics. *Technical Report TR-97-19, MERL*, 1997.

[Hah88] James K. Hahn. Realistic animation of rigid bodies. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 299–308, New York, NY, USA, 1988. ACM.

[Hes07] Peter Hess. Extended Boundary Conditions for Shallow Water Simulations. Master Thesis, Computer Graphics Laboratory, ETH Zurich, 2007.

[Hig94] Robert L. Higdon. Radiation boundary conditions for dispersive waves. *SIAM J. Numer. Anal.*, 31(1):64–100, 1994.

[HL97] X. He and L.-S. Luo. Lattice Boltzmann model for the incompressible Navier-Stokes equations. *J. Stat. Phys.*, 88:927–944, 1997.

[HNC02] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive Animation of Ocean Waves. *Proc. of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, July 2002.

[HSCD96] Shuling Hou, James D. Sterling, Shiyi Chen, and Gary Doolen. A Lattice Boltzmann Subgrid Model for High Reynolds Number Flow. *Fields Institute Communications*, 6:151–166, 1996.

[HTK$^+$04] B. Heidelberger, M. Teschner, R. Keiser, M. Mueller, and M. Gross. Consistent penetration depth estimation for deformable collision response. pages 339–346, 2004.

[HYP76] J. Hardy, O. De Pazzis Y., and Pomeau. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Physical Review A*, 13:1949–1960, 1976.

[Jak01] T. Jakobsen. Advanced character physics the fysix engine. *www.gamasutra.com*, 2001.

[JP99] Doug L. James and Dinesh K. Pai. Artdefo, accurate real time deformable objects. In *Computer Graphics Proceedings*, Annual Conference Series, pages 65–72. ACM SIGGRAPH 99, August 1999.

[KC04] Pijush Kundu and Ira Cohen. *Fluid Mechanics*. Elsevier Academic Press, 2004.

[KFCO06] Bryan M. Klingner, Bryan E. Feldman, Nuttapong Chentanez, and James F. O'Brien. Fluid animation with dynamic meshes. *ACM Trans. Graph.*, 25(3):820–825, 2006.

[KM90] M. Kass and G. Miller. Rapid, Stable Fluid Dynamics for Computer Graphics. *ACM Trans. Graph.*, 24(4):49–55, 1990.

[Lov03] Jörn Loviscach. Complex Water Effects at Interactive Frame Rates. *Journal of WSCG*, 11:298–305, 2003.

[LvdP02] Anita T. Layton and Michiel van de Panne. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*, 18(1):41–53, 2002.

[LWK03] Wei Li, Xiaoming Wei, and Arie E. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8):444–456, 2003.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Proc. of the 2003 ACM Siggraph/Eurographics Symposium on Computer Animation*, 2003.

[MDH07] Xing Mei, Philippe Decaudin, and Baogang Hu. Fast hydraulic erosion simulation and visualization on gpu. In *Pacific Graphics*, 2007.

[MG04] Matthias Müller and Markus Gross. Interactive virtual materials. In *GI '04: Proceedings of Graphics Interface 2004*, pages 239–246, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[MHR06] M. Müller, B. Heidelberger M. Hennix, and J. Ratcliff. Position based dynamics. *Proceedings of Virtual Reality Interactions and Physical Simulations*, pages 71–80, 2006.

[Mon92] J. Monaghan. Smoothed particle hydrodynamics. *Annu. Rev. Astron. Phys.*, 30:543–574, 1992.

[MSKG05] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. *Proc. of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation*, 2005.

[MTPS04] Antoine McNamara, Adrien Treuille, Zoran Popovic, and Jos Stam. Fluid control using the adjoint method. *ACM Trans. Graph.*, 23(3):449–456, 2004.

[MZ88] Guy R. McNamara and Gianluigi Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Phys. Rev. Lett.*, 61(20):2332–2335, 1988.

[NFJ02] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728, New York, NY, USA, 2002. ACM Press.

[NMK$^+$05] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson. Physically based deformable models in computer graphics. *Eurographics 2005 state of the art report*, 2005.

[OH95] J. F. O'Brien and J. K. Hodgins. Dynamic simulation of splashing fluids. In *CA '95: Proceedings of the Computer Animation*, page 198, 1995.

[PKW$^+$03] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes in 2D and 3D. Technical Report 03–8, Department for System-Simulation, Germany, 2003.

[QdL92] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK Models for Navier-Stokes Equation. *Europhys. Lett.*, 17(6):479–484, 1992.

[RB07] Matthias Mueller-Fischer Robert Bridson. Fluid Simulation, SIGGRAPH 2007 Course, 2007.

[RNGF03] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. Smoke simulation for large scale phenomena. *ACM Trans. Graph.*, 22(3):703–707, 2003.

[Sma63] J. Smagorinsky. General circulation experiments with the primitive equations. *Mon. Wea. Rev.*, 91:99–164, 1963.

[Sta99] Jos Stam. Stable Fluids. *Proc. of ACM SIGGRAPH*, pages 121–128, 1999.

[Suc01] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, 2001.

[SY05a] Lin Shi and Yizhou Yu. Controllable smoke animation with guiding objects. *ACM Trans. Graph.*, 24(1), 2005.

[SY05b] Lin Shi and Yizhou Yu. Taming liquids for rapidly changing targets. *Proc. of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation*, 2005.

[Tes04] Jerry Tessendorf. Simulating Ocean Surfaces. *SIGGRAPH 2004 Course Notes 31*, 2004.

[THM$^+$03] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. pages 47–54, 2003.

[TKPR06] Nils Thürey, Richard Keiser, Mark Pauly, and Ulrich Rüde. Detail-Preserving Fluid Control. *Proc. of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2006.

[TKZ$^+$04] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, Laks Raghupathi, A. Fuhrmann, Marie-Paule Cani, François 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.

[TPBF87] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *Computer Graphics Proceedings*, Annual Conference Series, pages 205–214. ACM SIGGRAPH 87, July 1987.

[TPR$^+$06] Nils Thürey, Thomas Pohl, Ulrich Rüde, Markus Oechsner, and Carolin Körner. Optimization and Stabilization of LBM Free Surface Flow Simulations using Adaptive Parameterization. *Computers and Fluids*, 35 [8-9]:934–939, September-November 2006.

[TR04] Nils Thürey and Ulrich Rüde. Free Surface Lattice-Boltzmann fluid simulations with and without level sets. *Proc. of Vision, Modelling, and Visualization VMV*, pages 199–208, 2004.

[TRS06] Nils Thürey, Ulrich Rüde, and Marc Stamminger. Animation of Open Water Phenomena with coupled Shallow Water and Free Surface Simulations. *Proc. of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2006.

[TSS$^+$07] Nils Thürey, Filip Sadlo, Simon Schirm, Matthias Müller-Fischer, and Markus Gross. Real-time simulations of bubbles and foam within a shallow water framework. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 191–198, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[WG00] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.

[WLMK04] Xiaoming Wei, Wei Li, Klaus M"uller, and Arie E. Kaufman. The Lattice-Boltzmann Method for Simulating Gaseous Phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004.

[Wol02] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[WWXP06] Changbo Wang, Zhangye Wang, Tian Xia, and Qunsheng Peng. Real-time snowing simulation. *The Visual Computer*, pages 315–323, May 2006.

[WZF$^+$03] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman. Natural phenomena: Blowing in the wind. *Proc. of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, pages 75–85, July 2003.

[YGL05] H. Yu, S.S. Girimaji, and Li-Shi Luo. Lattice Boltzmann simulations of decaying homogeneous isotropic turbulence. *Phys. Rev. E*, 71, 2005.

[YUM86] Larry Yaeger, Craig Upson, and Robert Myers. Combining physical and visual simulation and creation of the planet jupiter for the film 2010. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 85–93. ACM Press, 1986.