Simulation of deformable objects

Maarten Everts

Supervisor:
Henk Bekker

January 18, 2006

Abstract

More and more of the world around us is simulated by computers. These simulations are performed at different levels. Engineers for example use complex, slow, but precise simulations for analyzing car-crashes or optimizing a ship's hull shape. But for creating realistic effects in games and animations it is more important that good-looking simulations can be created as fast as possible. For the aforementioned purpose, creating visual effects, we have worked on methods for simulating soft, deformable objects like bread, foam, etc.

We started out with a new method for constraining the volume of deformable objects. This method is based on an algorithm originally used in the simulation of molecules and atoms for constraining bond lengths.

However, the important part of this research is the development of a promising new method for modeling deformable objects in simulations. In this method material is represented by a collection of particles. The forces on the particles as a results of deformation of the object are calculated using theory of a branch of physics called Continuum Mechanics that studies the properties of material. Because of this it is possible to simulate a particular material with predictable material constants. This is the advantage it has over another popular particle-based method called the mass-spring model where the particles are interconnected by springs. However, finding proper spring-constants for these springs is rather hard.

This new method for modeling material is implemented and we tested it on a number of different shapes. The simulated material appears to behave like one would expect and the simulations result in a number of nice images and animations. Furthermore, the forces inside the material are good. All in all, this new method has shown to have potential and work on it will continue.

Samenvatting

Meer en meer van de wereld om ons heen wordt gesimuleerd door computers. Deze simulaties gebeuren op verschillende niveaus. Zo gebruiken ingenieurs complexe, langzame, maar precieze simulaties voor het nabootsen van autobotsingen of het optimaliseren van de vorm van de boeg van een schip. Voor andere doeleinden als creëren van realistische visuele effecten in spellen en animaties is het juist belangrijker dat goed-lijkende simulaties snel gemaakt kunnen worden. Voor het laatstgenoemde doeleinde, het creëren van visuele effecten, hebben wij gewerkt aan methoden voor het simuleren van zachte vervormbare objecten als brood, schuimrubber etc.

We zijn begonnen met een nieuwe methode voor het constant houden van het volume van vervormbare objecten in simulaties. Deze methode is gebaseerd op een algoritme dat oorspronkelijk gebruikt wordt om in simulaties van moleculen en atomen de afstanden tussen atomen constant te houden. Deze nieuwe methode bleek goede resultaten te hebben.

Echter, het belangrijkste gedeelte van dit onderzoek is de ontwikkeling van een veelbelovende nieuwe methode voor het modelleren van vervormbare objecten in simulaties. In deze methode wordt materiaal gerepresenteerd door een verzameling deeltjes. Na vervorming van het object worden de resulterende krachten op de deeltjes uitgerekend met behulp van theorie van een tak van de natuurkunde genaamd Continuum Mechanica dat de eigenschappen van materialen onderzoekt. Daardoor is het mogelijk om met deze methode bepaalde materialen te simuleren met voorspelbare materiaalconstanten. Dit in tegenstelling tot een ander veelgebruikt deeltjesmethode genaamd het massa-veer model. In dit simpele model worden de deeltjes onderling verbonden met veren, maar het is erg lastig om de veerconstantes van de veren zo te kiezen dat een bepaald materiaal gesimuleerd kan worden.

Deze nieuwe methode is geïmplementeerd en getest op een aantal verschillende vormen. In de simulaties lijkt het materiaal zich te gedragen zoals men zou verwachten en dit levert een aantal aardige plaatjes en animaties op. Daarnaast zijn ook de krachten in het materiaal zoals verwacht. Deze methode heeft dus potentie en er zal dan ook verder aan gewerkt worden.

Contents

1 Introduction

Deformable objects are a part of every day life: the warm mattress one wakes up on, the yellow rubber duck in the bathtub, the fresh clean clothes from the closet and the breakfast sandwiches with soft cheese. And of course there is the human body itself; most human tissue (for example muscle) is easily deformable. For that reason at lot of work has been done on the simulation of deformable objects. Not only for explaining and predicting material behaviour but also (or, in particular) for visual effects (computer graphics). This Master's thesis is about the simulation of deformable objects for that last purpose, visual effects. But let us first take a look at properties of material.

1.1 Deformable material

Most materials can deform, even the the hardest steel will bend when enough force is applied to it. But as the actual deformation is so small, this is not the kind of material we are interested in simulating. We are looking at more easily deformable materials like foam and rubber.

These materials differ in how and under what force they deform. To start with, some materials are more easily deformed than others. If a rubber duck is squeezed, it deforms and when it is released the original shape of the duck returns. But if a piece of dough or clay is squeezed, it is permanently deformed, the original shape will not return (this is called plasticity). Then there is material that is easily compressed in one direction, but hard to compress in an other direction.

The branch of physics that studies these and other material properties is called Continuum Mechanics. More on this in chapter 4.

1.2 Applications

Simulation of deformable objects can be split in two domains: "perfect" simulation of material (Computational Mechanics) and visually pleasing simulations (Computer Graphics). The latter is what this Master's Thesis is about.

Simulation of deformable objects has various applications in Computer Graphics. To begin with, most animation films nowadays are created with computers, not only for rendering and such, but also for realistic looking behaviour of deformable objects like cloth. Well known examples of computer animated movies are "Shrek" and "Finding Nemo".

In interactive virtual environments people can interact with virtual worlds. Adding deformable objects to interact with will enhance the immersion. One special application in virtual environments is virtual surgery. The goal of virtual surgery is to allow people (e.g. medical students) practice surgical procedures without the need to cut open an animal or human corpse. Figure 1.1 shows an example. For virtual surgery it is of course important that the material being cut

behaves realistically, but in virtual environments like these instantaneous response is just as important. In other words, the physics must be right and it should work real-time.

In 3D games like "First Person Shooters" deformable objects can also be used to enhance immersion. However, here physically realistic behaviour is less important, it just has to look good.

(a) (b)

Figure 1.1: Virtual surgery examples (from `http://www.cmis.csiro.au/mediapics02.htm`)

1.3 Modelling material

Over time, various approaches have been tried for modelling deformable material, all with different intentions and applications in mind. Let us start with simulations where realism is the primary goal. This is the realm of Computational mechanics. In this mathematical field the partial differential equations provided by Continuum Mechanics are broken down and approximated using Numerical Mathematics techniques like finite differences and finite elements. These simulations can be very computationally intensive and it can take days to simulate just a few seconds.

On the other side of the spectrum are applications where speed is more important. One of the most widely used physics-based methods in this application field is the mass-spring method. In this relatively simple method deformable material is represented by a collection of mass-points connected by springs. We will discuss this method in more detail in chapter 2. There are however also methods not based on physics that are more focused on good visual results with the least amount of computation time. The geometrically motivated method discussed in [MHTG05] is an interesting example this.

Because there are applications (like virtual surgery) where both speed as realistic behaviour are important, various attempts have been made to strike a middle-ground. The method described in [Hau04] uses Computational Mechanics as a starting point. A collection of partial differential equations derived from Continuum Mechanics is simplified to ordinary differential equations by discretising the material into small-volumes, e.g. finite elements. These equations are then solved by special implicit integration methods. This method is still computationally intensive, but interactive speeds can be obtained.

Another notable contribution is [EGS03] where Continuum Mechanics theory is used to derive a (rather complex) particle system for deformable material. This article focuses on 2-dimensional material, e.g., cloth, but the authors state that it should be fairly easy to extend this method to three dimensions.

In [MKN+04] an interesting mesh-free particle system is introduced that can simulate a wide variety of materials (including plastic material).

The aforementioned methods are physically correct, however they are relatively complex. We will propose in this document a new particle based model for simulating deformable material that is both based on Continuum Mechanics and easy to grasp.

1.4 Research process

Developing a new model for deformable material was not the original goal of this project. It started out with an idea to add constant volume simulations to the mass-spring method by using a modified constraint dynamics method from Molecular Dynamics simulations (see chapter 3). After implementation of this method, it seemed to work. However, it soon became clear that mass-spring methods have one particular drawback: it is hard to choose the right spring-constants. In an effort to find a solution to this problem a new model for deformable material was born. Because this new method was so promising, the focus shifted. Over time two variations have been created (chapter 5 and 7 respectively) and a third variation is in the works.

2 Mass-spring method

2.1 Introduction

Simulation of motion is an important aspect of animation and interactive virtual environments like games. Motion is of course not limited to ordinary movement of rigid objects, but also includes deformation of objects. The most widely used method for modelling this kind of dynamic behaviour is the mass-spring method. The reason for its popularity is that this easy to grasp and easy to implement method achieves great visual results. On top of that, mass-spring systems are faster than other approaches and allow for interactive and real-time simulations, even on desktop systems.

2.2 Mass-spring principles

The basic principle is the following. An object is modelled as a collection of mass-points (particles) interconnected by weightless springs. Displacement of the particles will cause springs to exert force on these particles. A linear spring between particles i and j obeying Hooke's law results in the following force on particle i

$$\mathbf{F}_i = -K\frac{|\mathbf{r}_{ij}| - l_{ij}}{|\mathbf{r}_{ij}|}\,\mathbf{r}_{ij}.$$

Here, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the distance vector between particles i and j, l_{ij} is the rest length of the spring and K is a constant that describes the stiffness of the spring.

All spring forces and external forces (like gravity or user interaction) are accumulated each timestep of the simulation and the new positions of the particles are calculated by integration of Newton's equation of motion

$$\mathbf{F} = m\mathbf{a}.$$

Both explicit and implicit [BW98] integrators have been used for this. An explicit integrator like leap-frog is simple and stable, but has the disadvantage that very small timesteps are required to ensure stability for stiff systems. Implicit methods on the other hand allow large timesteps without loss of stability, but this also comes with a price as these methods require the solution of a (usually sparse) linear system.

2.3 Mass-spring limitations

Although mass-spring systems have been successfully used to model and simulate deformable objects, there are some limitations.

One of the main problems is the difficulty of setting parameters; there is no straightforward way to set the spring constants in such a way that a particular type of material can be simulated. The fact that techniques like neural networks [RNP98], genetic algorithms [LPC95] and simulated annealing [DKT95] have been used to find good spring constants illustrates this. Additionally, even finding spring constants to model uniform isotropic material is hard as the geometry of the springs influence the material properties.

Another issue is that a tetrahedron of springs easily collapses, or in other word, easily turns inside out. Consider a tetrahedron whose top particle is pushed down (see figure 2.1). As the tetrahedron becomes flatter, there is a sub-linear increase of force, i.e., it becomes easier to push through the tetrahedron. This results in the collapse of the tetrahedron, which can lead to instability.

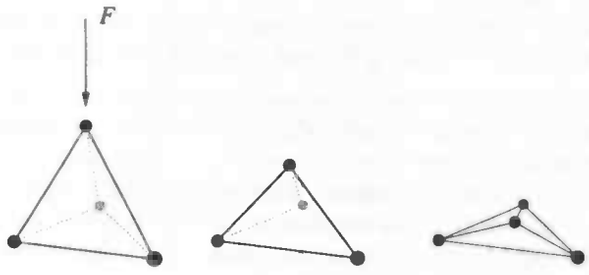


Figure 2.1: Collapse of tetrahedron

2.4 Adaptations

Several variations of the mass-spring method have been introduced in the literature. Most have not been derived from physics, but are somewhat ad hoc. Usually it involves the addition of special springs to obtain a certain effect. For example in [NT98] muscle is modelled by springs on the surface and special angular springs that keep the surface smooth.

In [BC00] a mass-spring inspired deformable model is introduced that allows anisotropy. For each volume element the user can supply the mechanical characteristics of the material along a given number of axes of interest. The internal forces will be acting along these axes instead of acting along the mesh edges. In addition, to preserve volume and prevent tetrahedron collapse special barycentric springs are added that originate from the barycenter of the tetrahedron.

Yet another approach is discussed in [JL04]. Instead of adding special diagonal springs and special shear springs, a generalized spring and oriented particle model is introduced. These springs have a additional reference vector to detect twist.

3 Volume constraint

3.1 Introduction

Most materials found in nature maintain a constant or quasi-constant volume during deformations (for example muscle-tissue), but as discussed in the previous chapter, it is hard to do a constant volume simulation using just the mass-springs technique. In this chapter an extension of the mass-spring method for constraining the volume of deformable objects is presented.

Molecular Dynamics (MD) simulations and mass-spring systems are quite similar. Both simulate a collection of mass-points that move according to Newton's law of motion. In MD simulations, the covalent interaction between two bonded atoms is usually very rigid, which means that the distance practically remains constant. Traditionally very rigid springs were used to model these covalent interaction, but this requires small time-steps to be used for the integration. For that reason most MD programs now use *constraint dynamics* to avoid the use of stiff springs for covalent interaction.

For long time, the SHAKE [RCB77] algorithm was the most widely used method for constraint dynamics. Each timestep, after an unconstrained update, this method iteratively corrects bond lengths (and angles) until a certain accuracy is reached. In [HBBF97] a faster alternative called LINCS was introduced. It was designed for bond (distance) constraints, but can be adapted to other geometrical constraints as well. Say, for example, the constraint that the sum of the distances between a number of particles must remain constant. This chapter discusses how to adapt this algorithm for a volume constraint.

3.2 Algorithm derivation

Although the derivation of the algorithm in [HBBF97] is presented in terms of matrices, no matrix multiplications are needed in the implementation. The derivation is reproduced here.

Consider a system of N particles, with positions given by a $3N$ vector $\mathbf{r}(t)$. The equations of motion are given by Newton's law

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = \mathbf{M}^{-1}\mathbf{f}, \tag{3.1}$$

where $\mathbf{f}$ is the $3N$ force vector and $\mathbf{M}$ is a $3N \times 3N$ diagonal matrix, containing the masses of the particles. Let us now assume that the system is constrained by K time-independent constraint equations

$$g_i(\mathbf{r}) = 0 \quad i = 1, \ldots, K. \tag{3.2}$$

Equations 3.1 and 3.2 may be solved by using Lagrange multipliers. That means that the constraints are added as a zero term to the potential $\mathbf{V}(\mathbf{r})$, multiplied by Lagrange multipliers $\lambda_i(t)$

$$-\mathbf{M}\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = \frac{\partial}{\partial \mathbf{r}}(\mathbf{V} - \boldsymbol{\lambda} \cdot \mathbf{g}). \tag{3.3}$$

A new notation is introduced for the gradient matrix of the constraint equations, which appears on the right-hand side of

$$B_{hi} = \frac{\partial g_h}{\partial r_i}. \tag{3.4}$$

Notice that $\mathbf{B}$ is a $K \times 3N$ matrix. It contains the directions of the constraints. Eq. (3.3) can now be simplified to give

$$-\mathbf{M}\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} + \mathbf{B}^T\boldsymbol{\lambda} + \mathbf{f} = 0. \tag{3.5}$$

Because the constraint equations are zero, the first and second derivatives of the constraints are also zero, so

$$\frac{\mathrm{d}\mathbf{g}}{\mathrm{d}t} = \mathbf{B}\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = 0, \tag{3.6}$$

$$\frac{\mathrm{d}^2\mathbf{g}}{\mathrm{d}t^2} = \mathbf{B}\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}^2t} + \frac{\mathrm{d}\mathbf{B}}{\mathrm{d}t}\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = 0. \tag{3.7}$$

To solve for $\boldsymbol{\lambda}$, left-multiply eq. (3.5) with $\mathbf{BM}^{-1}$ and use eq. (3.7) to get

$$\frac{\mathrm{d}\mathbf{B}}{\mathrm{d}t}\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} + \mathbf{BM}^{-1}\mathbf{B}^T\boldsymbol{\lambda} + \mathbf{BM}^{-1}\mathbf{f} = 0. \tag{3.8}$$

Thus, the constraint forces are

$$\mathbf{B}^T\boldsymbol{\lambda} = -\mathbf{B}^T(\mathbf{BM}^{-1}\mathbf{B}^T)^{-1}\mathbf{BM}^{-1}\mathbf{f} - \mathbf{B}^T(\mathbf{BM}^{-1}\mathbf{B}^T)^{-1}\frac{\mathrm{d}\mathbf{B}}{\mathrm{d}t}\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t}. \tag{3.9}$$

Substituting this into eq. (3.5) gives the constrained equations of motion. Using the abbreviation $\mathbf{T} = \mathbf{M}^{-1}\mathbf{B}^T(\mathbf{BM}^{-1}\mathbf{B}^T)^{-1}$ these are

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = (\mathbf{I} - \mathbf{TB})\mathbf{M}^{-1}\mathbf{f} - \mathbf{T}\frac{\mathrm{d}\mathbf{B}}{\mathrm{d}t}\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t}. \tag{3.10}$$

A straightforward discretization in a leap-frog method uses a half-timestep difference between the discretization of the first and last derivatives in eq. (3.10),

$$\frac{\mathbf{v}_{n+\frac{1}{2}} - \mathbf{v}_{n-\frac{1}{2}}}{\Delta t} = (\mathbf{I} - \mathbf{T}_n\mathbf{B}_n)\mathbf{M}^{-1}\mathbf{f}_n - \mathbf{T}_n\frac{\mathbf{B}_n - \mathbf{B}_{n-1}}{\Delta t}\mathbf{v}_{n-\frac{1}{2}}, \tag{3.11}$$

$$\frac{\mathbf{r}_{n+1} - \mathbf{r}_{n-1}}{\Delta t} = \mathbf{v}_{n+\frac{1}{2}}. \tag{3.12}$$

The discretization of the last term in eq. (3.11) at $t_{n-\frac{1}{2}}$ is the actual linearization of the problem. Because the right-hand side of eq. (3.11) does not depend on $t_{n+\frac{1}{2}}$, the new velocities can be calculated directly. If in eq. (3.11), the term $\mathbf{B}_{n-1}\mathbf{v}_{n-\frac{1}{2}}$ is zero, the equation simplifies to

$$\mathbf{v}_{n+\frac{1}{2}} = (\mathbf{I} - \mathbf{T}_n\mathbf{B}_n)(\mathbf{v}_{n-\frac{1}{2}} + \Delta t\, \mathbf{M}^{-1}\mathbf{f}_n), \tag{3.13}$$

and because $\mathbf{I} - \mathbf{T}_n\mathbf{B}_n$ is the projection matrix that sets the constrained coordinates to zero, $\mathbf{B}_n\mathbf{v}_{n+\frac{1}{2}} = 0$. Thus if $\mathbf{B}_0\mathbf{v}_{\frac{1}{2}} = 0$, eq. (3.13) can be used instead of eq. (3.11). Substituting eq. (3.13) into eq. (3.12) gives the constrained new positions

$$\begin{aligned}\mathbf{r}_{n+1} &= (\mathbf{I} - \mathbf{T}_n\mathbf{B}_n)(\Delta t\, \mathbf{v}_{n-\frac{1}{2}} + (\Delta t)^2\mathbf{M}^{-1}\mathbf{f}_n + \mathbf{r}_n) \\ &= \mathbf{r}_{n+1}^{\mathrm{unc}} - \mathbf{M}^{-1}\mathbf{B}_n^T(\mathbf{B}_n\mathbf{M}^{-1}\mathbf{B}_n^T)^{-1}\mathbf{B}_n(\mathbf{r}_{n+1}^{\mathrm{unc}} - \mathbf{r}_n).\end{aligned} \tag{3.14}$$

Here $\mathbf{r}_{n+1}^{\text{unc}}$ are the new positions of the particles after an unconstrained update. So each timestep there is an unconstrained update of the positions followed by a correction of the particle positions based on eq. (3.14).

3.3 Volume constraint

The volume of an object can be constrained at different levels. One option is to constrain the volume of each sub-element (tetrahedron) of the object and the other is to constrain the total volume.

3.3.1 Sub-element volume constraint

A volume constraint equation g_V will be of the form

$$V(\mathbf{r}) - V_0 = 0, \tag{3.15}$$

where V_0 is the initial volume and $V(\mathbf{r})$ is the volume at a certain point in time.

For a constraint on the volume of a single tetrahedron, an expression for the volume of a tetrahedron is needed. Let a tetrahedron be specified by its polyhedron vertices at (x_i, y_i, z_i) where $i = 1, \ldots, 4$. Then the volume is given by

$$V_{tet} = \frac{1}{6} \begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix}. \tag{3.16}$$

A mass-spring system with sub-volume constraints could also prevent the collapse of tetrahedra (see section 2.3). But there will be as many constraints as there are tetrahedra, so B will become a $T \times 3N$ matrix, where T is the number of tetrahedra in the model. This would require the matrix inverse of a $T \times T$ matrix, which is too computationally intensive for models of hundreds of tetrahedra. For that reason we will focus on a total volume constraint.

3.3.2 Total volume constraint

The most straightforward way to obtain an expression for the total volume of an object that is subdivided into tetrahedra is to sum the volumes of all individual tetrahedra. However, there is an easier way to calculate the total volume of an object.

Depending on the position and order of the vertices, the determinant of eq. (3.16) can be both positive and negative. Although a negative volume is an odd concept, this notion can be used to explain how to calculate the volume of a tetrahedrized object using only the particles on the surface mesh.

Consider a (not necessarily convex) object W and an arbitrary point p in space (see figure 3.1). The surface of W is described as a triangular mesh. Each triangle face (three points ordered with regard to the surface normal) combined with point p defines a tetrahedron. Many of these

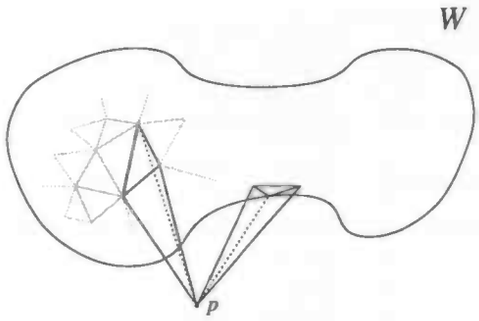


Figure 3.1: Volume calculation from surface mesh

tetrahedra will of course overlap, but it can be easily shown that the positive and negative volumes of the tetrahedra compensate for this; the sum of all the volumes is equal to the actual volume of object W.

Point p is arbitrary, so one might as well use $(0,0,0)$. Replacing (x_4, y_4, z_4) in eq. (3.16) with $(0,0,0)$ gives the 'contribution' of a face to the total volume:

$$V_{part} = \frac{1}{6}(x_1 y_2 z_3 - x_1 y_3 z_2 - x_2 y_1 z_3 + x_2 y_3 z_1 + x_3 y_1 z_2 - x_3 y_2 z_1). \tag{3.17}$$

Now $V(\mathbf{r})$ can be defined as

$$V(\mathbf{r}) = \frac{1}{6}\sum_j (x_{j,1}y_{j,2}z_{j,3} - x_{j,1}y_{j,3}z_{j,2} - x_{j,2}y_{j,1}z_{j,3} + x_{j,2}y_{j,3}z_{j,1} + x_{j,3}y_{j,1}z_{j,2} - x_{j,3}y_{j,2}z_{j,1}) \tag{3.18}$$

where $(x_{j,i}, y_{j,i}, z_{j,i})$ is the position of vertex i of face j.

3.4 Implementation

As stated above, no actual matrix multiplications are needed in the implementation. This section will discuss some of these and other implementation details. Let us first see how to calculate the gradient matrix B. Recall the definition of B:

$$B_{hi} = \frac{\partial g_h}{\partial r_i}.$$

If there is only one constraint, B is a $1 \times 3N$ matrix with the partial derivatives of the volume constraint g_V. Eq. (3.18) showed that the volume constraint g_V is a simple sum of products of three elements. As a result, the partial derivatives in B are also sums of products. Table 3.1 shows the partial derivatives per face. Calculating B comes down to

1. Initialize all elements of B to zero

	partial derivative
x_1	$\frac{1}{6}(y_2 z_3 - y_3 z_2)$
y_1	$\frac{1}{6}(x_3 z_2 - x_2 z_3)$
z_1	$\frac{1}{6}(x_2 y_3 - x_3 y_2)$
x_2	$\frac{1}{6}(y_3 z_1 - y_1 z_3)$
y_2	$\frac{1}{6}(x_1 z_3 - x_3 z_1)$
z_2	$\frac{1}{6}(x_3 y_1 - x_1 y_3)$
x_3	$\frac{1}{6}(y_1 z_2 - y_2 z_1)$
y_3	$\frac{1}{6}(x_2 z_1 - x_1 z_2)$
z_3	$\frac{1}{6}(x_1 y_2 - x_2 y_1)$

Table 3.1: Partial derivatives of equation eq. (3.17)

2. For each face
   - calculate partial derivatives according to table 3.1
   - add these values to the corresponding elements of B

Now that we have calculated $\mathbf{B}_n$, let us see how to calculate the other terms in eq. (3.14). Eq. (3.14) contains a matrix inverse:

$$(\mathbf{B}_n \mathbf{M}^{-1} \mathbf{B}_n^T)^{-1}.$$

Fortunately, for a total volume constraint, B is a $1 \times 3N$ matrix and $\mathbf{B}_n \mathbf{M}^{-1} \mathbf{B}_n^T$ is a simple 1×1 matrix, whose inverse is trivial. Calculating $\mathbf{B}_n \mathbf{M}^{-1} \mathbf{B}_n^T$ itself is also trivial, it is a simple summation:

$$\sum_i^{3N} \frac{(B_{1i})^2}{m_i}$$

where m_i is the ith diagonal element of M. The operations in the rest of eq. (3.14) are simple vector operations (additions and dot-products).

3.5 Correction term

In [HBBF97] the authors state that the derived algorithm is correct, but not stable. Numerical errors can accumulate, which leads to drift, because the second derivatives of the constraints were set to zero instead of the constraints themselves. For that reason a correction term is added to eq. 3.13:

$$\mathbf{v}_{n+\frac{1}{2}} = (\mathbf{I} - \mathbf{T}_n \mathbf{B}_n)(\mathbf{v}_{n-\frac{1}{2}} + \Delta t \mathbf{M}^{-1} \mathbf{f}_n) - \frac{1}{\Delta t} \mathbf{T}_n (\mathbf{B}_n \mathbf{r}_n - \mathbf{d}) \tag{3.19}$$

where d is a vector with the bond lengths of the constraints. This term should be zero, as it is the real distance between bounded atoms minus the prescribed bond lengths. Of course, in a simulation, this term is almost never exactly zero, so adding this term eliminates accumulation of numerical errors and makes the method stable.

The correction term above was a feedback mechanism specifically for bond length constraints and we would like to do something similar for the volume constraint. For a total volume constraint it turns out that the expression $\mathbf{B}_n \mathbf{r}_n$ is related to the total volume of the simulated object.

It is three times the actual volume. This can be explained as follows. The expression $\mathbf{B}_n\mathbf{r}_n$ is basically the partial derivative of eq (3.18) multiplied by $\mathbf{r}$. But each product of eq. (3.18) shows up in $\mathbf{B}$ three times; the partial derivative with respect to each element of the product. That is the reason why $\mathbf{B}_n\mathbf{r}_n$ is three times the actual volume. Now with correction term eq. (3.14) becomes:

$$\mathbf{r}_{n+1} = \mathbf{r}_{n+1}^{\text{unc}} - \mathbf{M}^{-1}\mathbf{B}_n^T(\mathbf{B}_n\mathbf{M}^{-1}\mathbf{B}_n^T)^{-1}\,\gamma(\mathbf{B}_n\mathbf{r}_{n+1}^{\text{unc}} - 3\mathbf{V}_0) \tag{3.20}$$

where γ is a parameter that influences the effect of the feedback term.

4 Continuum Mechanics

4.1 Introduction

A simulation of a deformable 3D object should look realistic and for that, the physics must be right. Continuum Mechanics is a branch of physics that studies the properties and behaviour of materials without regarding the atomic structure. Instead, the material is viewed as a continuum: a continuous distribution of matter for which physical properties can be handled in the infinitesimal limit. This allows differential equations to be used to solve problems in Continuum Mechanics.

Continuum Mechanics has theory for both fluids and solids, but as our goal is to simulate deformable objects, this discussion of Continuum Mechanics will focus on solids. For solids, different classes of materials can be distinguished. Elastic material for example returns to its initial shape after deformation, but plastic material (for instance, dough) permanently deforms after large enough applied stress. For viscoelastic material on the other hand, the force required for deformation depends on the velocity of the deformation.

4.2 Stress

An important concept in Continuum Mechanics is *stress*, which is a way to specify the interaction and forces inside a material body. Say B is a material continuum and S a closed surface within B. The material within the volume enclosed by the surface S interacts with the material outside of this volume.

To express this interaction we will look at a small surface element of area ΔS on the surface S. Let $\mathbf{n}$ be a unit outward normal to ΔS and let $\Delta \mathbf{F}$ be the resultant force exerted across ΔS. The force $\Delta \mathbf{F}$ depends on the location and size of the area and the orientation of the normal. The *stress principle of Euler and Cauchy* states that, if ΔS tends to zero, the ratio $\Delta \mathbf{F}/\Delta S$ tends to a definite limit $d\mathbf{F}/dS$, and that the moment of the force acting on the surface ΔS about any point within the area vanishes in the limit. The vector $d\mathbf{F}/dS$ is called the *stress vector* $\mathbf{t_n}$.

The stress vector can be obtained from the surface normal $\mathbf{n}$ and a stress tensor σ_{ij}.

$$t_i = \sum_j \sigma_{ij} n_j \tag{4.1}$$

A tensor is a matrix-like mathematical object and a stress tensor defines the state of stress at a certain point. The stress tensor is defined by nine scalars, three for each axis (figure 4.3 illustrates this). The matrix representation of the stress tensor can be found in figure 4.2. Assuming that the net torque is zero, it can be shown that the stress tensor is symmetric:

$$\tau_{12} = \tau_{21}, \quad \tau_{13} = \tau_{31}, \quad \tau_{23} = \tau_{32}.$$

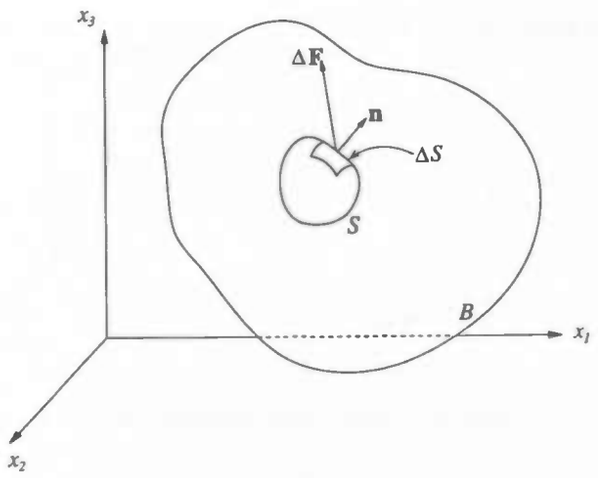


Figure 4.1: Stress

So there are only six independent components: the three *normal stresses* (τ_{11},τ_{22} and τ_{33}) on the diagonal and three *shearing stresses*. Each of these components has the dimension of force per unit area.

$$[\sigma_{ij}] = \begin{bmatrix} \tau_{11} & \tau_{12} & \tau_{13} \\ \tau_{21} & \tau_{22} & \tau_{23} \\ \tau_{31} & \tau_{32} & \tau_{33} \end{bmatrix}$$

Figure 4.2: Stress tensor

4.3 Strain

Stress inside a material is the result of a change of size and/or shape. This state of deformation is called *strain* and like stress, strain can be defined at a certain point in space with a tensor. For Hookean elastic solids, the stress tensor is linearly proportional to the strain tensor.

$$\sigma_{ij} = \sum_{k,l} c_{ijkl}\, e_{kl} \tag{4.2}$$

Here e_{kl} is the strain tensor and c_{ijkl} is the tensor of elastic constants. The latter is a four-dimensional tensor and specifies the properties of the material at a point. This tensor has $3^4 = 81$ elements, but because of symmetry there are only 36 independent constants.

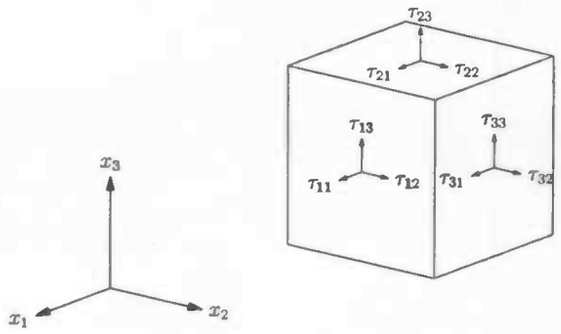


Figure 4.3: Stress tensor components

4.4 Tensor

In one of the previous sections we stated that a tensor is a matrix-like object. But there is more to it than that. A tensor can in fact be looked at as the generalization of vectors and matrices. It is used to represent geometric or physical quantities relative to a given frame of reference and has the property that it transforms according to certain rules under a change of coordinates. Because we will use only one coordinate system, this tensor property is not significant for our purpose.

Tensor calculus is rather complex and not very intuitive; it is hard to imagine what a tensor exactly represents. Fortunately there is also a more engineering approach to Continuum Mechanics that looks at simple isotropic material without using tensors. This way of describing material properties is what we will use in later chapters to define a model for simulating deformable objects.

4.5 Isotropic material

An important simplification in Continuum Mechanics is to look at isotropic material, i.e., material whose mechanical properties do not depend on the direction. For isotropic material the number of independent constants in c_{ijkl} can be reduced to two: the Lamé constants λ and μ. These constants relate to two important parameters for isotropic material: the modulus of elasticity E (Young's modulus) and the shear modulus G.

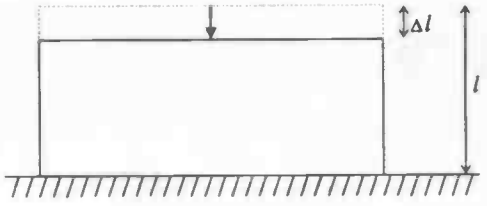
Consider a rectangular block of isotropic linear elastic material aligned to the axes. As a result of a force applied in the y direction, this block has become smaller by Δl. See figure 4.4(a). There is a linear relation between the resulting normal stress σ and the relative change of length $\epsilon = \frac{\Delta l}{l}$:

$$\sigma = E\epsilon. \tag{4.3}$$

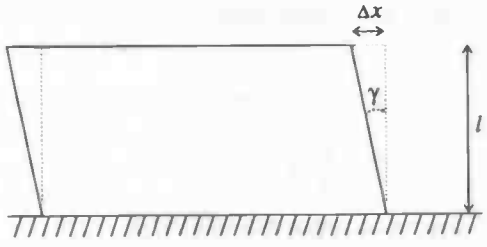
Similarly, as a result of force applied in the x direction to the top of the block, the top has been shifted by Δx with respect to the bottom. See figure 4.4(b). There is a linear relation between shear stress τ and the angle of shearing γ:

$$\tau = G\gamma.. \tag{4.4}$$

The dimension of both σ and τ is force per unit area. The normal stress is perpendicular to the plane and the shear stress is parallel to the plane.



(a)



(b)

Figure 4.4: Normal and shear stress

4.6 Poisson's ratio

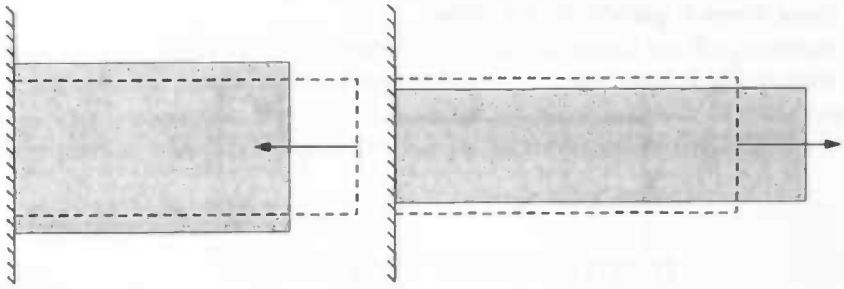
When pressed upon in one direction, many materials will expand in the two other directions (lateral extension, figure 4.5(a)) and when stretched they become thinner (transverse contraction, figure 4.5(b)). So the volume remains somewhat constant. The magnitude of this effect is expressed by the Poisson's ratio ν of the material. This is the ratio between the relative change of length ($\Delta l/l$) and the relative change of thickness ($\Delta d/d$),

$$\nu = -\frac{\Delta l/l}{\Delta d/d}.$$

As the relative change of thinkness occurs in both opposite directions, a perfectly incompressible material (like rubber) has a Poisson's ratio of 0.5, which is also the maximum. For most materials ν has a value between 0.0 and 0.5.

There is a relationship between the two constants Young's modulus E and shear modulus G introduced in the previous section, and Poisson's ratio ν:

$$G = \frac{E}{2(1+\nu)}. \tag{4.5}$$



(a) Lateral extension (b) Transverse contraction

Figure 4.5: Compress and stretch of a block of material

From $0 \leq \nu \leq 0.5$ follows that:

$$\frac{E}{3} \leq G \leq \frac{E}{2}. \tag{4.6}$$

For material whose volume remains constant the following holds:

$$E = 3G.$$

5 Continuum Mechanics Particle Model 1

5.1 Introduction

As discussed in chapter 2 using mass-spring systems for modelling deformable objects has some problems. Most notably the difficulty of finding good spring constants. This chapter will introduce a promising alternative. Its most important advantage is that by using Continuum Mechanics theory it allows to model a particular material with predictable material constants.

As with mass-spring systems, the material is modelled as a set of particles with pair interactions. The difference between this new method and mass-spring systems is that for the conventional approach springs there are central interactions, i.e. the forces are in the direction of the springs. In our new method this is not the case.

Voronoi diagrams play an important role in this method. Later another method was developed that does not need Voronoi diagrams. It is discussed in chapter 7.

5.2 Prerequisites

5.2.1 Barycentric coordinates

When a deformed object is moved to another position, the stress forces will of course remain the same with respect to the object. In other words, internal forces are relative to the position and orientation of the body. A technique that can assist in expressing relative position and orientation is the use of barycentric coordinates. With barycentric coordinates the position of a point in space can be expressed in terms of the positions of other points. For dimension $n-1$, vertex $\mathbf{r}$ can be written as the weighted sum of n points v_n:

$$\mathbf{r} = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i. \tag{5.1}$$

The weights λ_i, the barycentric coordinates, sum to 1:

$$\sum_{i=1}^{n} \lambda_i = 1. \tag{5.2}$$

In three dimensions, the position of a point can be expressed in four other points, so then $n = 4$. Obtaining the weights for a point r given four other points is a matter of solving a system of (in this case) four equations.

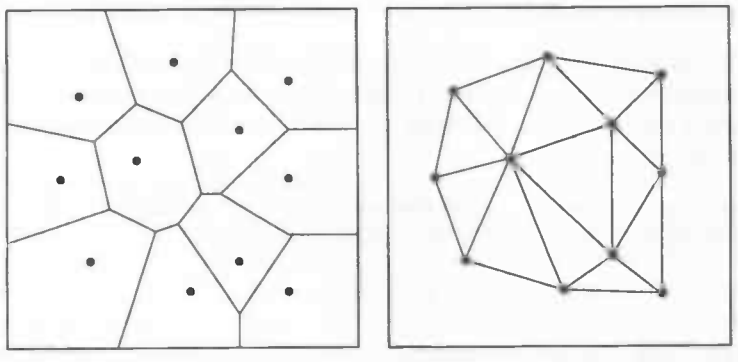
5.2.2 Voronoi diagram

A Voronoi diagram is a special kind of subdivision of space based on the positions of a set of points. It is explained here for the 2-dimensional case. Consider a set S of n points in a plane. Each point $p \in S$ can be assigned a region in which each point is closer to p than to any other point in S. This region is called the Voronoi region V_p. A more precise definition:

$$V_p = \{x \in \mathbb{R}^2 | ||x - p|| \leq ||x - q||, \forall q \in S\}.$$

Figure 5.1(a) illustrates this.

The dual of the Voronoi diagram is the Delaunay triangulation. It can be obtained by connecting points in S if their Voronoi regions share an edge. This is illustrated in figure 5.1(b)



(a) (b)

Figure 5.1: 2D Voronoi diagram (left) and Delaunay triangulation (right) of the same point-set

Both concepts, the Voronoi Diagram and the Delaunay triangulation, can be translated to three dimensions. Figure 5.2 illustrates a three dimensional Voronoi region, also called a Voronoi cell. The dual of a 3D Voronoi diagram consists of tetrahedra instead of triangles. It is constructed by connecting two points by a line if their Voronoi regions share a face.

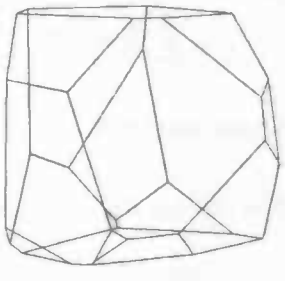


Figure 5.2: 3D Voronoi cell example

5.3 Overview

We will start with the general idea of this method for modelling deformable material. A deformable body B is represented by a set of particles. It is assumed that at $t = 0$ there is no stress inside the body. After $t = 0$ force is exerted on the particles and (some of) the particles will start to move. The body B is now deformed and this leads to stress inside the material.

Now we let each particle represent a piece of material. With the theory of chapter 4 on Continuum Mechanics, we can say that the total force on a piece of material bounded by the surface S, is equal to the forces exerted over the surface S. In other words, the sum of all the forces exerted over infinitesimal patches on S:

$$F = \int_S \mathbf{t}\, dS.$$

This notion is what will be used to calculate the forces on a particle. The piece of material belonging to a particle is bounded by a number of faces and over these faces the stress forces will be calculated. A natural way to assign space to particles, is to use the Voronoi diagram of the particles. That way every particle P_i has a corresponding Voronoi cell V_i which represents the material belonging to particle P_i. Figure 5.2 shows an example of a 3D Voronoi cell.

5.4 Model

We will now look in more detail at how to use the concept described above to calculate the forces on the particles. A Voronoi cell consists of faces, edges and vertices. Over these faces, which we will call Voronoi faces, the stress forces will be calculated. A Voronoi face is always shared between two Voronoi cells, for that reason a Voronoi face is denoted by $V_{i,j}$: the face shared by Voronoi cell V_i and Voronoi cell V_j. A Voronoi face consists of a number of vertices, which we will call Voronoi vertices and are denoted by $V_{i,j,k}$: vertex k of face $V_{i,j}$.

The Voronoi diagram is calculated at $t = 0$, when body B is still at rest. At time $t > 0$, particles have moved and the original Voronoi diagram is no longer correct. Recalculating the Voronoi diagram for the new particle positions could result in a Voronoi diagram with a combinatorial structure differing form the one at $t = 0$. In other words, faces could disappear or have more/less vertices.

To maintain a constant combinatorial structure, the Voronoi diagram is instead *reconstructed* for the new particle positions. A Voronoi diagram is built from its vertices $V_{i,j,k}$. The position of $V_{i,j,k}$ is determined by the position of four particles. Thus $\mathbf{V}_{i,j,k}$ can be expressed as the barycentric coordinates of these four particles. By using the barycentric coordinates to find the new positions of the Voronoi vertices, the Voronoi diagram effectively moves with the particles. Of course the resulting deformed Voronoi diagram is not a real Voronoi diagram and the Voronoi faces are also no longer flat (see figure 5.3). We call the reconstructed Voronoi cell at time t $V_i(t)$ and the reconstructed Voronoi face at time t $V_{i,j}(t)$.

Now the stress forces over the Voronoi faces can be calculated. Consider a face $V_{i,j}$ between P_i and P_j. We assume that the material is isotropic, so the theory of section 4.5 can be used. This means that to calculate the stress forces over this face the relative change of length and the angle of shearing are needed.

The relative change of length can be approximated by

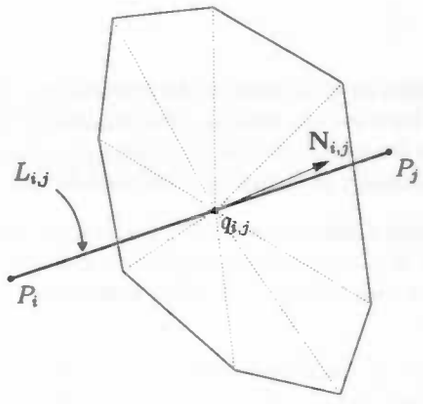


Figure 5.3: Deformed Voronoi face

$$\epsilon = \frac{d_{i,j}(t) - d_{i,j}(0)}{d_{i,j}(0)}$$

where $d_{i,j}(t)$ is the distance between particle P_i and particle P_j at time t.

The reconstructed Voronoi face $V_{i,j}(t)$ is an indication how the material between P_i and P_j was oriented at $t = 0$. As indicated above, this face is not necessarily flat anymore. In order to obtain an approximate normal, a point $q_{i,j}$ is introduced. It is the midpoint of the line $L_{i,j}$ between P_i and P_j. By connecting every $V_{i,j,k}(t)$ of $V_{i,j}$ to $q_{i,j}$, the face is subdivided into triangles. The approximate normal $\mathbf{N}_{i,j}$ is then obtained by summation of the normals of the triangles.

The angle of shearing γ is the angle between $N_{i,j}$ and $L_{i,j}$.

Using eq. (4.3), the elastic stress force can now be defined as:

$$|\mathbf{F}_{\text{elastic},i,j}| = \frac{d_{i,j}(t) - d_{i,j}(0)}{d_{i,j}(0)} \; A_{i,j} \; E$$

where $A_{i,j}$ is the area of the face $V_{i,j}(0)$. The direction of $\mathbf{F}_{\text{elastic},i,j}$ is same as $\mathbf{N}_{i,j}$. Using eq. (4.4), the shear stress force can be defined as

$$|\mathbf{F}_{\text{shear},i,j}| = \gamma \; A_{i,j} \; G$$

where γ is the angle between $\mathbf{N}_{i,j}$ and $L_{i,j}$. $\mathbf{F}_{\text{shear},i,j}$ is perpendicular to $\mathbf{N}_{i,j}$.

Figure 5.4 illustrates the direction of the forces.

5.5 Particle masses

In addition to the internal material forces, the masses of the particles are also required to integrate Newton's equation of motion,

$$F = ma.$$

The Voronoi diagram, which is already computed for the force calculation, assigns a Voronoi cell to each particle. A Voronoi cell indicates what material a particle represents and by combining a density value and the volume of a Voronoi cell, the mass of the particle can be obtained.

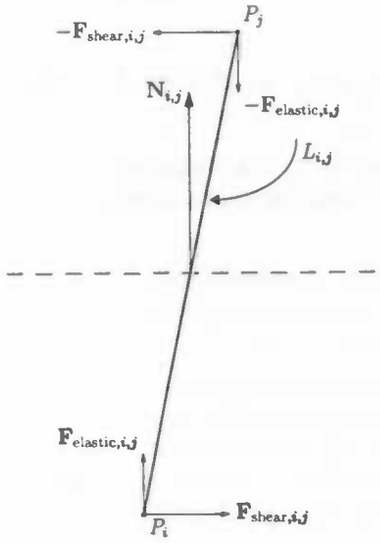


Figure 5.4: Direction of the forces

6 Model generation

6.1 Introduction

For both the mass-spring method as the method discussed in the previous chapter a three dimensional shape needs to be discretised into sub-volumes. A natural way to do this is to divide the shape into tetrahedra.

6.2 Constrained Delaunay and Conforming Delaunay

Delaunay tetrahedrization of a set of points is discussed in section 5.2.2. A Delaunay tetrahedrization is a suitable option for mesh generation because the smallest angle in the resulting tetrahedra is maximized.

Usually a three dimensional model is given as a surface mesh. A *Constrained Delaunay tetrahedrization* (CDT) is a Delaunay tetrahedrization of the input points of the shape with the added constraint that the surface mesh must be part of the tetrahedrization.

In most applications of tetrahedrization well-shaped, i.e., non-flat tetrahedra are preferred as this for instance can increase numerical stability. A way to measure the quality of a tetrahedron is to consider its radius-edge ratio [Si]. A tetrahedron t has a unique circumsphere. Let $R = R(t)$ be the radius of this circumsphere and $L = L(t)$ be the length of the shortest edge. The radius-edge ratio $Q = Q(t)$ is measured by taking the ratio, that is:

$$Q = R/L$$

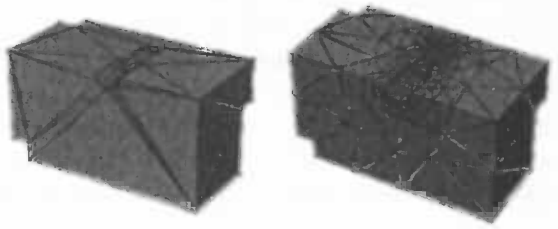
For well-shaped tetrahedra this ratio is usually small (e.g. smaller than 2.0) and can thus be used to measure the quality of tetrahedra.

A quality or conforming Delaunay tetrahedrization is obtained by adding vertices to the CDT to obtain tetrahedra of a higher quality. Figure 6.1(b) shows an example of a quality tetrahedrization.

Several (free) programs are available that can generate 3D meshes, for example Gmsh [GR] and Tetgen [Si]. The latter is discussed in more detail in the next section.

6.3 Tetgen

Tetgen [Si] is a free, open-source program for generating tetrahedral meshes. It can generate exact constrained Delaunay tetrahedrization and quality (conforming Delaunay) meshes.



(a) CDT mesh

(b) Conforming Delaunay mesh

Figure 6.1: Constrained and Conforming Delaunay tetrahedrization examples

As input it takes several different formats that describe the surface of the model. It can then generate a constrained Delaunay tetrahedrization or a conforming Delaunay mesh (figure 6.1). If a quality tetrahedrization is requested, the program makes sure that all tetrahedra have a radius-edge ratio smaller than 2.0. The quality of the mesh can also be tuned by providing an alternative radius-edge ratio.

Sometimes it is useful to have a more or less uniform distribution of tetrahedra (and particles) in the model. For this situation it is possible to impose a maximum volume bound on all tetrahedra of the conforming Delaunay tetrahedrization.

In addition, Tetgen can produce a list of neighbors for each tetrahedron. This proved useful for constructing a Voronoi diagram.

6.4 Voronoi diagram

For the method discussed in the previous chapter, the Voronoi diagram is needed to define the region of influence for each particle. Tetgen already proved to be a very suitable program for 3D mesh generation, but unfortunately Tetgen does not have an option to generate a Voronoi diagram along with the tetrahedrization.

Qhull [Qhu], another open-source computational geometry program, does have an option to generate 3D Voronoi diagrams. Unfortunately it can only do this for a set of points and it cannot bound the Voronoi Diagram (some Voronoi regions extend into infinity). Therefore it was examined if there is a way to obtain the Voronoi diagram based on the output of Tetgen.

6.4.1 Voronoi diagram from tetrahedra

Tetgen produces a list of points, a list of tetrahedra, a list of neighbors and a list of faces. For each point we would like a (bounded) Voronoi cell. A simple, intuitive, yet unproven algorithm was developed and it is discussed in this section.

A Voronoi diagram is built from its Voronoi vertices. Every Voronoi vertex is the circumcenter of a tetrahedron. This can be calculated with the code found on [She96] and [She98]. These circumcenters then have to form a Voronoi face. The next assumption is that every edge in the tetrahedrization has one corresponding Voronoi face. Now consider an edge between particle P_i and particle P_j. This edge is the edge of a number of tetrahedra, as illustrated in figure 6.2. The circumcenters of these tetrahedra form a Voronoi face. The tetrahedra need to be sorted around the edge P_iP_j to get a proper face. As noted above, Tetgen can produce a list of neighboring tetrahedra for each tetrahedron. This can be used to find a a proper sequence of tetrahedra.

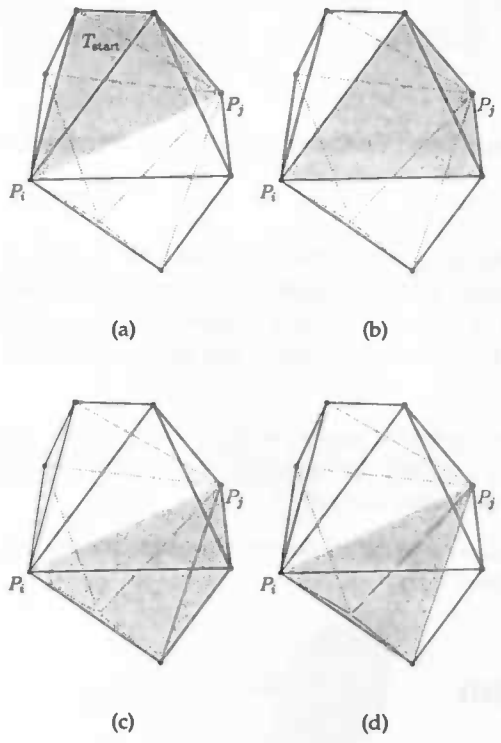


Figure 6.2: Process of finding tetrahedra around edge

To find the Voronoi face through the edge between P_i and P_j, we do the following. First, find a tetrahedron that has P_iP_j as an edge. We will call this tetrahedron T_{start} (see figure 6.2(a)). It has two neighboring tetrahedra that also have P_iP_j as an edge. One is chosen (figure 6.2(b)). This tetrahedron has one neighbor tetrahedron that has P_iP_j as an edge and has not been seen before. This is repeated until T_{start} reached. Now we have a sorted list of tetrahedra around edge P_iP_j whose circumcenters define a Voronoi face.

In that way the Voronoi faces inside the body can be found, but the Voronoi faces at the surface of the shape require some special care. The Voronoi cells of particles at the boundary of the model extend into infinity and the faces of these Voronoi cells need to be 'cut' at the surface.

We assume that the surface is a triangular mesh. Consider an edge P_iP_j on the surface. It is the edge of two triangle faces. Again, the tetrahedra around this edge need to be found. The

difference with the procedure described above for internal edges is that T_{start} is not an arbitrary tetrahedron that has P_iP_j as an edge, but one of the two tetrahedra that have a face on the surface. The "walking" around the edge in this case continues until the other tetrahedron T_{end} at the surface is reached. It is assumed that the Voronoi face for edge P_iP_j has exactly two edges that cut through the surface. One of these edges will start at the circumcenter of T_{start} and the other at the circumcenter of T_{end}. To get a proper bounded Voronoi face, these circumcenters need to be connected with some additional points.

The points of intersection of the infinite edges and the surface are exactly the centers of the circles through the two the triangles adjacent to edge P_iP_j. At these intersection points additional "Voronoi vertices" are added. Let us call these vertices a and b. If a and b are on the same side of P_iP_j (figure 6.4), then the Voronoi face can be constructed by simply connecting a and b. But this only gives good results if the two triangles have (almost) the same orientation. In the other case, if a and b are not on the same side of P_iP_j, an extra point $q_{i,j}$, the midpoint of the edge P_jP_j, is added between a and b (see figure 6.3).

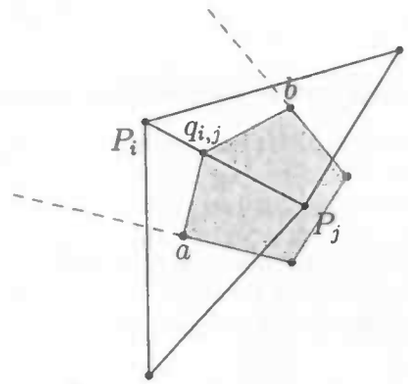


Figure 6.3: Surface triangles adjacent to P_iP_j

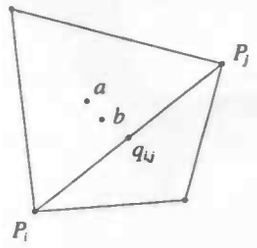
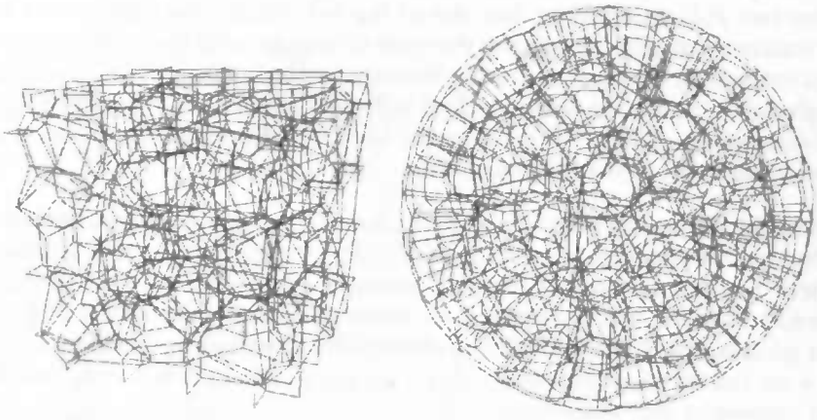


Figure 6.4: Triangle circumcenter in same triangle

6.4.2 Results

The method describe above was implemented and tested on a few simple shapes. Figure 6.5 shows some examples.



(a) Cube (b) Sphere

Figure 6.5: Results of the Voronoi Diagram algorithm on simple objects

6.4.3 Problems

The intuitive algorithm described above seemed to give good results for simple objects like a cube or a sphere (figure 6.5). But for more complex objects there are some irregularities. For example the (flat) flaps of the dolphin model in figure 6.6.

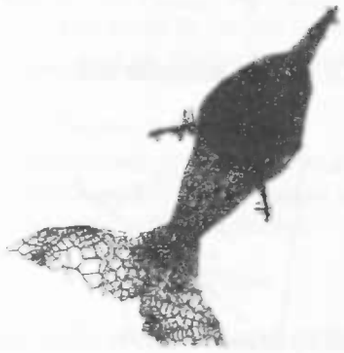


Figure 6.6: Irregularities in Voronoi diagram of a complex model

Also it raised some questions. We mostly use conforming Delaunay tetrahedrization. Is there a Voronoi diagram for that anyway? If so, does the above algorithm produce it? What happens if an edge has a corresponding Voronoi face that is (partly) outside the input shape? Figure 6.6 shows a possibility. In [Ede01] the dual of a constrained Delaunay triangulation, the extended Voronoi diagram, is discussed and it is rather complex. How complex will it be in three dimensions, for a constrained Delaunay tetrahedrization? These issues and questions spurred the development of yet another alternative for mass-springs systems which does not need a Voronoi diagram and is discussed in the next chapter.

7 Continuum Mechanics Particle Model 2

7.1 Introduction

As the results in chapter 9 will show, the method described in chapter 5 seems to work. An important aspect of this method is the Voronoi diagram. However, chapter 6 showed that obtaining a suitable Voronoi Diagram for a constrained Delaunay tetrahedrization is not always straightforward. In an effort to remove the need of a Voronoi diagram a new method was developed.

7.2 Overview

The general idea is similar. Again a deformable body B is modelled by a set of particles and each particle represents a piece of material. The idea is still that the forces on a particle are computed by calculating the force over the faces that bound the material belonging to the particle. The difference lies in the way the material of the body is divided among the particles. In chapter 5 Voronoi cells were used for this and the variation described in this chapter uses a different approach.

7.3 Model

Finding Voronoi diagrams for tetrahedrizations generated by Tetgen proved problematic (see chapter 6). For that reason another way for distributing material among particles was created that only uses the tetrahedrization.

Assume a body B is subdivided into tetrahedra. A particle P_i in B is the vertex of say N tetrahedra. With six small sub-faces we now subdivide each tetrahedron into 4 sub-volumes, one for each particle (illustrated in figure 7.1). Thus the part of B that belongs to particle P_i consists of N sub-volumes.

For a more precise definition of the sub-volumes, consider a tetrahedron T in B. It is defined by the positions of four points $p_i, i = 0 \ldots 3$. In T some additional points are defined:

C	barycenter of $p_0 \ldots p_3$ (center of the tetrahedron)
$M_{i,j,k}$	barycenter of p_i, p_j and p_k (center of a triangle face)
$M_{i,j}$	barycenter of p_i and p_j (midpoint of an edge)

With these points, six sub-faces can be defined which subdivide the tetrahedron into four sub-volumes. Figure 7.1 illustrates this. So the material belonging to a particle that is the vertex of N tetrahedra is bounded by $3N$ sub-faces.

For quality tetrahedrizations (see section 6.3) the shape of the all the sub-faces around a particle will not be very different from a the Voronoi cell of that particle.

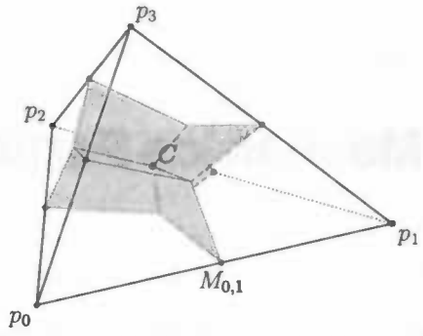


Figure 7.1: Sub-faces in tetrahedron

7.4 Force calculation

So the internal material forces will be calculated over the sub-faces. How this can be done is discussed next. Consider a sub-face between point p_0 and p_1 in tetrahedron T. At $t = 0$ the orientation of this face is stored. For this the normal of the sub-face is calculated (the cross-product of $M_{0,1,3} - M_{0,1}$ and $M_{0,1,2} - M_{0,1}$). This vector is normalized to unit length and added to one of the points of T, say p_0. This position is then expressed as the barycentric coordinates (see section 5.2.1) of $p_0 \dots p_3$.

At $t > 0$, the positions of the particles have changed. The deformed normal $\mathbf{N}_d$ is reconstructed from the barycentric coordinates and the new positions of the particles. The geometric normal $\mathbf{N}_g$ is again calculated from the cross-product of $M_{0,1,3} - M_{0,1}$ and $M_{0,1,2} - M_{0,1}$. With these two vectors the relative change of length and the angle of shearing can be calculated for this sub-face. Figure 7.2 shows how the vectors $\mathbf{v}_{\text{elastic}}$ and $\mathbf{v}_{\text{shear}}$ can be found. The vector $\mathbf{v}_{\text{elastic}}$ represents the direction of the elastic force and its length is proportional to the relative change of length for this sub-face. The elastic force on the particle associated with point p_i can then be calculated with this vector:

$$\mathbf{F}_{\text{elastic}} = A_{i,j}\mathbf{v}_{\text{elastic}},$$

where $A_{i,j}$ is the area of the sub-face between point p_i and point p_j. The vector $\mathbf{v}_{\text{shear}}$ represents the direction of the shear force and its length is proportional to the angle of shearing γ for this sub-face. The shear force can be calculated with this vector:

$$\mathbf{F}_{\text{shear}} = A_{i,j}\mathbf{v}_{\text{shear}}.$$

The above process is done for each of the six sub-faces of all the the tetrahedra.

7.5 Problems

During implementation and exploration of this method some doubts arose about its correctness. Egbert van der Es, a fellow student who is also working on this project, analyzed using Matlab the forces that would occur during deformation in a single tetrahedron when applying this method. Two types deformations are analyzed:

1. Moving the top vertex up and down (compression)

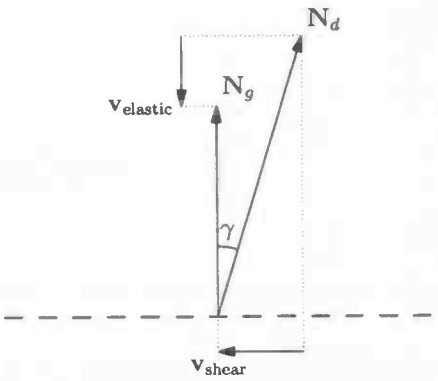
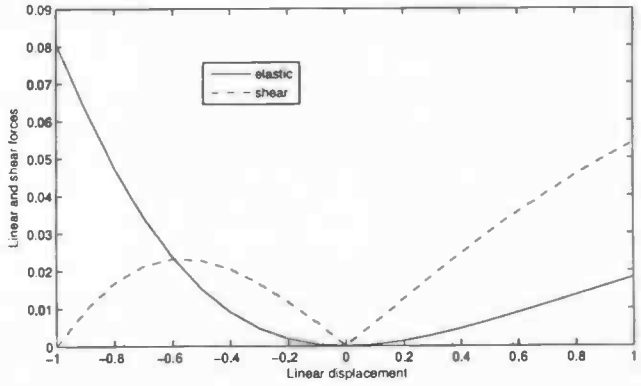


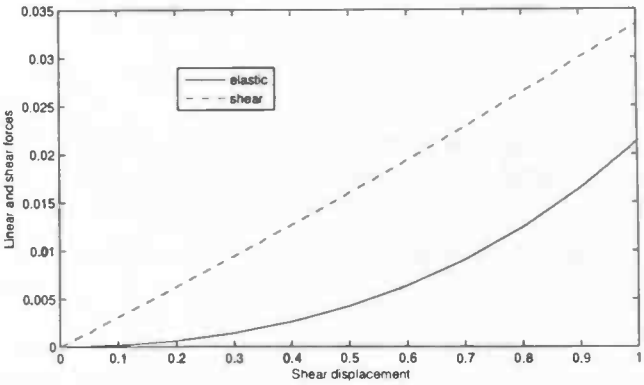
Figure 7.2: Force calculation

2. Moving the top vertex sideways (shear)

He created some graphs (depicted in figure 7.3) in Matlab that show the total force in a tetrahedron as the result of simple deformation. In both graphs the derivative of the linear or elastic force is zero at the origin. This is not what one would expect, especially for compression. This means that it was likely that this model is not correct. Later it was realized that the above model contains an assumption on how sub-faces would deform upon deformation of the tetrahedron. This is where the model fails; one cannot make assumptions on the deformation of material changes inside a deformed tetrahedron based on the four vertices of the tetrahedron.



(a) Compression



(b) Shear

Figure 7.3: Force inside a deformed tetrahedron (courtesy of Egbert van der Es)

8 Implementation

8.1 Introduction

In this chapter we will discuss the implementation of the models and concepts of the previous chapters. Let us start with some general information. Simulations of physical phenomena are usually quite computationally intensive. In particular when the number of simulated parts (in our case particles) is large. For that reason C++ was chosen as the implementation language. It strikes a good balance between speed and maintainability.

Development was done on a Linux workstation and the most obvious choice for the compiler is gcc. Both gcc version 3.3 and 4.0 have been used.

All libraries used are cross-platform, so in theory it should be fairly easy to port the software to the Windows platform.

8.2 Overview

The software is subdivided in several programs and static libraries. They will be mentioned here and are discussed in more detail in the following sections.

libStructure A static library with the data-structure for 3D models (structures). It also has code for saving to and reading from files.

libSimulator A static library that contains all simulation code. Uses `libStructure`. Discussed in section 8.3.

SimGUI A Graphical User Interface (GUI) that allows the user to control the simulation and to visualize the results of the simulation. Uses `libSimulator`. Discussed in section 8.5

SimConsole A simple 'head-less' console application that uses libSimulator to run a simulation and generates output for later processing. Uses `libSimulator`.

StructGen Program that generates a 3D model (structure) for use with SimGUI and SimConsole. The Tetgen library is used for tetrahedrization. Uses `libStructure`. Discussed in section 8.6.

8.3 Simulator

The Simulator library contains all the code for running a simulation of a particle system. Right from the start it was clear that some flexibility would be useful to facilitate experiments with

different approaches. For that reason a flexible object-oriented design was chosen. Of course it evolved somewhat over time, but the general idea remained the same.

Figure 8.1 shows the overall design of the Simulator library. It consists of three classes: Simulation, Action and Structure. To explain this design we need the following notion. A *simulation* for the most part consists of a number of "things" being done to the particles each and every timestep. These "things" are modeled by the abstract class *Action*. An Action-class derivative has two special functions: `init` and `doStep`. The `init` function can for example be used to initialize data-structures specific for the Action. The `doStep` function is called each timestep. Examples of Actions are "Gravity" and the integration method "LeapFrog".

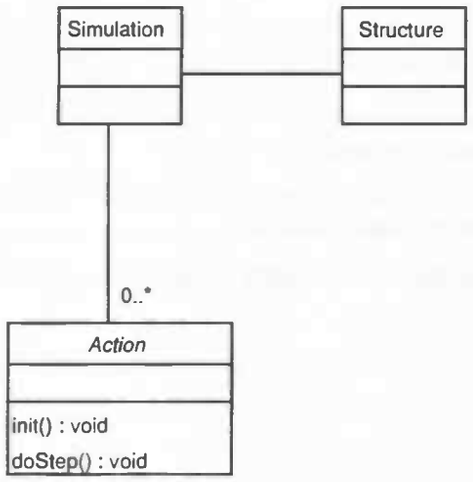


Figure 8.1: Design of libSimulator

In the *Simulation* class information about the simulation is stored. This includes the timestep, the number timesteps to perform, the Structure and a set of Action-derivatives which specify what action to perform on the structure each timestep.

The SimulationEngine class does the actual work, but what it needs to do is actually quite simple. In pseudo-code:

```
Simulation sim
foreach Action a of sim do
   a.init()
end

while not done do
  foreach Action a of sim do
    a.doStep()
  end
end
```

So the above design allows us to specify an experiment by adding Action derivatives to the Simulation class. For each new model or integrator we can simply make a new Action derivative without changing code elsewhere.

With this in place, something is needed that allows the user of the software to actually specify an experiment without recompiling the program. One option would be to use a configuration file of some sort, but instead was chosen for the added flexibility an embedded language. The next section will elaborate on this.

8.3.1 Lua

Lua [Lua] is a small but powerful programming language specifically designed for embedding in applications. Lua is implemented as a small library of C functions, written in ANSI C and thus compiles on many platforms. Embedding this library in the Simulator library allows the user to specify their simulation experiments as a simple Lua script. In the Lua script the Actions are added to the simulations, listing 8.1 shows an example Lua script that illustrates this. With the parameters of the constructors the Actions can be configured.

The classes of the Simulator need to be exposed to the to the Lua interpreter. There are several ways to do this, but one of the easiest is to use toLua++ [toL]. This program can be given a list of C++ header files[1] with the classes that need to be exposed to Lua and from that it generates glue-code between C++ and Lua. This means that when a new Action derivative is created only the toLua++ program needs to be run and the new Action is instantly available for the Lua scripts. There is no need to update a parser for a configuration file.

Listing 8.1: Example lua code

```
1 -- This is a comment
2 sim = Simulation:new()
3
4 -- Set simulation parameters
5 sim:setSaveSkip(4);
6 sim:setNumTimesteps(80001);
7 sim:setStepSize(0.0001)
8
9 -- Load a 3D model and set the density to 1200 kg/m3
10 sim:setStructure(Structure:new("balkmetgat.str",1200))
11
12 -- Add actions to the simulation
13 sim:addAction(ClearForces:new())
14 sim:addAction(Gravity:new(10.0))
15 sim:addAction(MaterialForces:new(3000000.0,1000000.0))
16 sim:addAction(LeapFrog:new())
17 --sim:addAction(Ground:new(0.0))
18 sim:addAction(GlobalDamping:new(0.99))
19 --sim:addAction(SimpleVolumeConstraint:new())
20 --sim:addAction(MoveSteady:new(0,0.0,2,0.0005,40000,true))
21
22 setupSimulation(sim)
```

[1] Which have some special tolua++ comments.

8.4 Available Actions

Over time, several Action-derivatives have been created. They will be presented in this section.

Calculate Flux An Action class derivative that calculates the flux of material forces through a plane in the material. It does this by summing all the forces over the edges that go through the specified plane. The plane is always perpendicular to a coordinate axis and its positions is specified by a real on this axis.

ClearForces An Action class that sets the forces of all particles to zero (initialization).

EventForces An Action class that handles Event objects. Event objects can be used to apply a force on a particle at a particular timestep. It is mostly obsoleted by other Action derivatives like MoveBlock and MoveSteady.

GlobalDamping An Action class that dampens the system by multiplying the velocity of all particles with a factor that can be specified.

Gravity Gravity action. Applies gravity to the object. The gravity constant can be specified.

Ground Ground is an action that makes particles bounce of the ground. The level of the ground can be specified.

LeapFrog An Action class that implements the leap-frog integration method.

MaterialForces The MaterialForces class implements the mass-spring alternative based on Continuum Mechanics that was discussed in chapter 5. It calculates the forces on the particles of the the simulated object.

MaterialForces2 The MaterialForces class implements the mass-spring alternative based on Continuum Mechanics that was discussed in chapter 7. This one does not use a Voronoi Diagram. It calculates the forces on the particles of the the simulated object.

MoveBlock With the MoveBlock class a number of particles in a block can be specified and these particles will be moved in a certain direction (one of the axes) with a certain speed.

MoveSteady With the MoveSteady class a number of particles in a plane can be specified and these particles will be moved in a certain direction (one of the axes) with a certain speed. This is used for example to push down the top of a cube.

Rotate With the Rotate class a number of particles in a plane can be specified and these particles will be rotated with a certain speed.

SimpleVolumeConstraint The SimpleVolumeConstraint class tries to enforce a volume constraint. It is the implementation of the volume constraint method discussed in chapter 3. This action should be placed after the integration method (LeapFrog).

SineMovement With the SineMovement class a number of particles in a plane can be specified and these particles will be moved in a sine-like way.

Tabletop This action represents a tabletop that can be used together with the Gravity action to obtain the effect of objects falling/draping over table. The height and the dimension of the top can be specified.

8.5 SimGUI

SimGUI is a Graphical User Interface (GUI) that allows the user to control the simulation and to visualize the results of the simulation. The interface consists of a number of elements. The numbers in the following list correspond to the numbers in figure 8.2, a screenshot of SimGUI.

1. Visualization viewport. The user can control the view by clicking and dragging the mouse cursor in the viewport.
2. Frame information. Shows information about the currently displayed frame: timestep, time, volume and volume difference.
3. Frame selection. Allows the user to select a different frame to display.
4. Simulation control. Here the simulation can be started and paused.
5. Visualization control. With it, the user can enable or disable the following options:
   - Show faces.
   - Show edges of the tetrahedra (the interaction pairs).
   - Show all Voronoi faces, i.e. show the Voronoi Diagram. For example used to generate figures 6.5 and 6.6.
   - Highlight one particular Voronoi face.
   - Highlight one particular Voronoi cell.
6. Simulation information.

The interface is created with the wxWidgets GUI library. The most import reasons for this choice were:

- Open-source, free-as-in-speech.
- Cross-platform: available for Unix, Windows, and Mac OS X. Uses the native toolkit.
- It is a C++ framework.
- Previous experience.

8.6 StructGen

StructGen is a simple console program that generates 3D models for use with SimGUI and SimConsole. The Tetgen [Si] library is embedded in the program and handles the tetrahedrization. StructGen has the same command-line options as the Tetgen binary:

```
structgen <tetgenswitches> <input> <output>
```

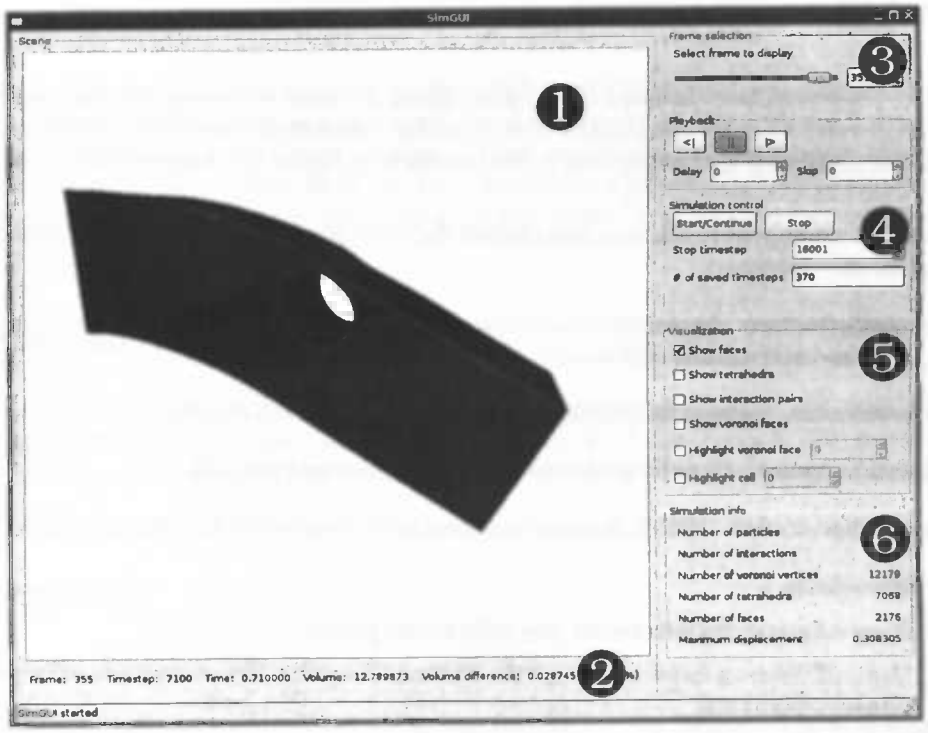


Figure 8.2: Screenshot of SimGUI

For example to generate a 3D model with quality tetrahedra whose volumes are at most 0.01 (see also section 6.3) the following command must be issued:

```
structgen nqa0.01 input.mesh output.str
```

where `input.mesh` is the input-file and `output.str` is the output str file (discussed in the next section).

StructGen lets the Tetgen library read the input file and generate a tetrahedrization. From Tetgen's data-structure the Voronoi diagram is generated according to the procedure described in section (6.4.1). Then with the help of `libStructure` the output file is generated.

8.7 File-format

The files that describe the 3D model to be simulated are called `str`-files. The name comes from "structure"; it describes the structure of the material. This file-format was created to be able to store the information about an object that was needed for the two methods discussed in chapter 5 and 7 respectively. This included besides the tetrahedrization a description of the Voronoi diagram. The goal of this file-format was simplicity: simple to read and simple to write. For that purpose all the data is stored in one file, as opposed to a separate file for each data type, which is the approach Tetgen uses.

A `str`-file consists of blocks. Each block holds one type of data. The first line of a block states the type of data in this block. The next line states the number of data items. The rest of the block consists of the data items, one per line. All data-types are implicitly numbered and numbering starts at zero, e.g. the first particle position read belongs to particle 0, the second to particle 1, etc.

Table 8.1 shows what block types are available and how the data-items are formatted. This table is mostly self-explanatory, perhaps except for `voronoiridges`. First of all, a Voronoi ridge is the same as a Voronoi face, it is the term that Qhull uses. Secondly, the data format is explained as follows. The Voronoi face has `<num vv>` Voronoi vertices, `<vv0> <vv1> ... <vvn>` and it lies between particles `<p0>` and `<p1>`.

Block type	data format
`particles`	`<x> <y> <z>`
`tetrahedra`	`<p0> <p1> <p2> <p3>`
`neighbors`	`<t0> <t1> <t2> <t3>`
`faces`	`<p0> <p1> <p2>`
`voronoivertices`	`<x> <y> <z>`
`dependencies`	`<p0> <p1> <p2> <p3>`
`voronoiridges`	`<p0> <p1> <num vv> <vv0> <vv1> ... <vvn>`
`voronoicells`	`<num vf> <vf0> <vf1> ... <vvn>`

Table 8.1: Block types in str-file

8.8 Complexity

We will look at the complexity of the following parts of the implementation:

- `SimpleVolumeConstraint`, the constraint method discussed in chapter 3.
- `MaterialForces`, the method discussed in chapter 5.
- `MaterialForces2`, the method discussed in chapter 7.
- `StructGen`, the program that generates the models.

8.8.1 SimpleVolumeConstraint

The SimpleVolumeConstraint class implements a volume constraint for the total volume. Each timestep B is computed and with this vector eq. (3.14) is calculated. Computing B consists of $O(F)$ operations, where F is the number of triangle faces in the model. Calculating eq. (3.14) consists of a number of $O(N)$ operations, where N is the number of particles. As the number of faces is in general smaller than the number of particles, the complexity per timestep is $O(N)$.

8.8.2 MaterialForces

In the MaterialForces class the forces are calculated per Voronoi face, or in other words, per edge. This calculation involves an approximation of a normal with Voronoi vertices around the edge. In quality tetrahedrizations the number of Voronoi vertices (i.e., the number of tetrahedra) around an edge is small (at most 10 to 15), this calculation is practically $O(1)$. With this we can conclude that the complexity per timestep is $O(E)$, where E is the number of edges in the model.

8.8.3 MaterialForces2

In the MaterialForces2 class the forces are calculated per tetrahedron. Each tetrahedron has exactly six sub-faces, so the complexity of the calculation of the forces in one tetrahedron is $O(1)$. This brings the complexity per timestep to $O(T)$, where T is the number of tetrahedra in the model.

8.8.4 StructGen

StructGen first lets Tetgen tetrahedralize a mesh and from Tetgen's data-structure the Voronoi diagram is created. Tetgen's complexity can be found in Tetgen's documentation. For finding the Voronoi diagram a data-structure is created that allows StructGen to find edges, faces and tetrahedra. For this a map-data structure is used. Filling this data-structure is $O(E \log E)$ operations, where E is the number of edges.

Finding the Voronoi faces is done per edge and as noted above the number of tetrahedra around an edge is usually small. This process of constructing a Voronoi face can include a look-up in the aforementioned data-structure, which has a complexity of $O(\log E)$. Constructing the whole Voronoi diagram has thus a complexity of $O(E \log E)$, where E is the number of edges in the model.

8.9 Accuracy

In the implementation double precision floating point numbers are used. This should be precise enough as the approximations in the simulations transcend possible numerical errors due to lack in precision.

9 Results

9.1 Introduction

In the previous chapters, one constraint method and two models for deformable objects have been discussed. This chapter will present some results. Unless noted otherwise, the following material properties are used in the experiments described in this chapter:

Material	rubber
Density	1.2×10^3 kg m^{-3}
Young's modulus (E)	3.0×10^6 Pa
Shear modulus (G)	1.0×10^6 Pa

For most experiments we will look at both methods discussed in chapter 5 and 7. In the text they will be called CMPM1 and CMPM2 respectively.

9.2 Hanging beam

We will start with a visually oriented experiment that was used to see how the simulated material behaves. The object being simulated is a beam of rubber (with a hole in it) that is on one side attached to something immovable. At $t = 0$ gravity is turned on and as a result the unconnected part of the beam is pulled down. Without damping this part will eventually spring back up. This is illustrated by figure 9.2. This sequence of shots is created with CMPM1. Repeating the experiment for CMPM2 gives similar results, except that the lowest point of the beam is higher (see figure 9.1), i.e., using CMPM2 instead of CMPM1 appears to result in stiffer material. More on this in the next section. For this experiment CMPM2 also appears to be more stable. In the case of CMPM1 the beam goes down, goes up, goes down again and then explodes, but in the case of CMPM2 the beam just keeps on going up and down.

The original model was found on Tetgen's website [Si] and the tetrahedrized model has 1679 particles, 7068 tetrahedra and 9835 edges. The timestep used for both methods was 0.0001 s.

9.3 Compressing material

The simulated object in this experiment is a cube of rubber of $1 \times 1 \times 1$ meters aligned with the coordinate axes. During simulation two opposing faces are pulled away or pushed towards from each other. This is done by moving the particles in the faces in the right direction each timestep (this is implemented in the `MoveSteady` Action). The particles are free to move in the other directions. At a certain point in time the movement of the particles is stopped.

(a) CMPM1 (b) CMPM2

Figure 9.1: Lowest point of the beam for both methods

During simulation the flux of force in the cube is calculated. We look at the flux in a plane through the cube perpendicular to the movement of the two faces. The flux is calculated by taking the sum of the forces that are exerted over the edges that cut through the plane. The resulting value for the flux can then be compared with the expected value. The stretching stops when the faces have moved 0.16m. At that point in time the relative change in length ϵ is 0.16 and thus the expected elastic force flux is (according to section 4.5):

$$F_{\text{elastic flux}} = A\epsilon E = 1.0 \times 0.16 \times 3 \times 10^6 = 4.8 \times 10^5 \text{ N}.$$

Figures 9.3 and 9.4 show the resulting graphs for CMPM1 and CMPM2 respectively. The elastic and shear force flux are both shown in the graphs. During the simulation the angles inside the material also change, so there will also be shear forces. The graphs show that this indeed occurs.

For both CMPM1 and CMPM2 it seems that compressing material results in larger elastic force flux than for stretching material, yet the relative change of length is the same.

Looking at the elastic force flux for $\epsilon = 0.16$, one can see that in all cases F_{elastic} is in the neighborhood of the expected value. In particular for CMPM2 when stretching the material. Also, for CMPM2 the elastic force flux is larger than for CMPM1 and this supports the observation of the previous section that using CMPM2 results in stiffer material.

Equation 4.3 of 4.5 indicates there should be a linear relation between the relative change of length and the force flux. The graphs of figures 9.3 and 9.4 indeed show a linear relationship.

Some additional information about the simulation. The timestep used is $0.0001s$. The tetrahedrized cube has 150 particles, 522 tetrahedra and 787 edges.

9.4 Shearing material

The object being simulated is the same cube as in the previous section. Only in this experiment the material is sheared by moving two opposing faces away from each other (illustrated by figure 9.6). Again the force flux is calculated, only now the plane is parallel to the movement of the two faces. Shearing stops when the angle of shearing γ is 0.57. The expected shear force flux for this angle is

$$F_{\text{shear flux}} = A\gamma G = 1.0 \times 0.57 \times 1.0 \times 10^6 = 5.7 \times 10^5 \text{ N}.$$

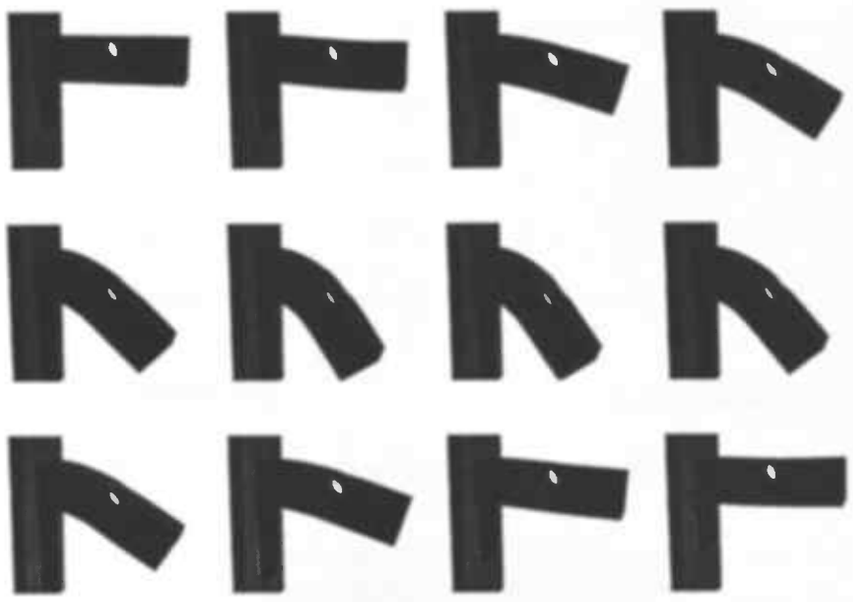


Figure 9.2: Sequence of shots of a simulation with CMPM1

The shear force flux for both methods is again not far from the expected value. And once more, the value for CMPM2 is closer to the expected value than the value for CMPM1.

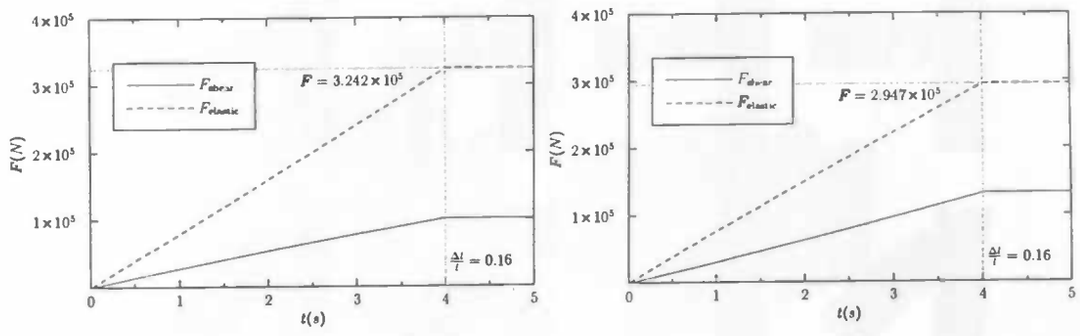
The model and the timestep are the same as in the previous section.

9.5 Fine vs. coarse tetrahedrization

One way to lower the computational costs of a simulation is to lower the number of particles or volume elements. The question is how much this will influence the properties of the simulated material. To test this the following experiment is done.

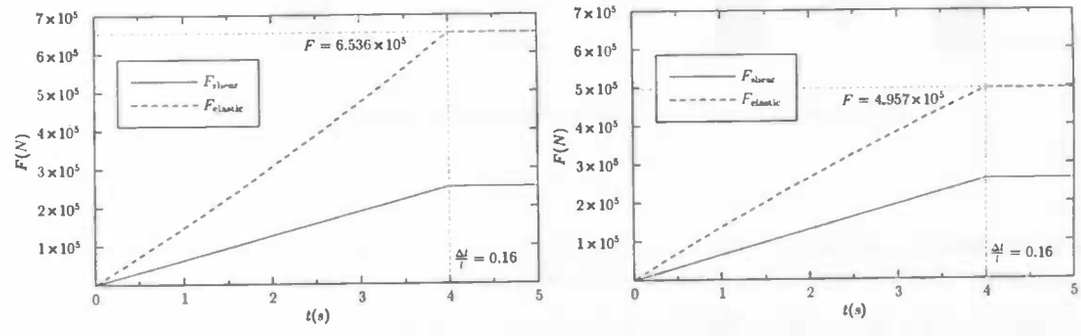
First two models of a $1 \times 1 \times 1$ cube are created: one with a coarse tetrahedrization and one with a finer tetrahedrization. This is done by using the maximum volume bound option of StructGen/Tetgen. In the coarse cube the tetrahedra have a volume of at most $0.005\ m^3$ and in the fine cube the tetrahedra have a volume of at most $0.0004\ m^3$. This results in two cube models with 124 and 1035 particles respectively. Figure 9.7 shows a cut-through of both models.

The actual simulation is similar to the one discussed in section 9.3. Two faces of the cube are pulled away from each other and again force flux is calculated. This is done for both models and both methods (CMPM1 and CMPM2) and the resulting force flux values can then be compared. The graphs of figure 9.8 and 9.9 show that a finer tetrahedrization results in smaller elastic force flux and in a larger shear force flux. What these graphs also show is that CMPM2 does a better job than CMPM1 at keeping a more or less constant force flux for different levels of coarseness.



(a) Compressing material (b) Stretching material

Figure 9.3: Force flux for CMPM1



(a) Compressing material (b) Stretching material

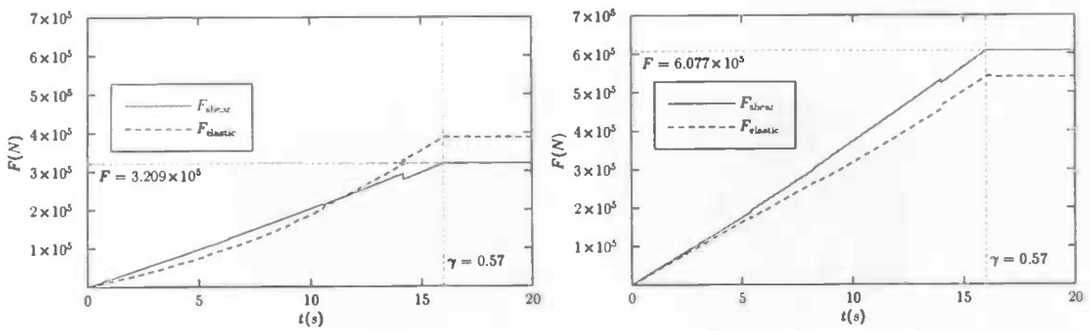
Figure 9.4: Force flux for CMPM2

For CMPM1 the difference is about 10% and for CMPM2 the difference is about 1%. This is acceptable for the computer graphics applications these methods are aiming for.

9.6 Volume

Although the focus of this Master's Thesis shifted towards the methods CMPM1 and CMPM2, the volume constraint method of chapter 3 does work. The experiment described in this section will illustrate this.

The setup is similar to the one in section 9.3. Two faces of a rubber cube are pushed towards each other, until $\frac{\Delta l}{l} = 0.24$. The difference is that the volume of the cube is constrained. Again the flux is calculated and these values are incorporated in the graphs of figure 9.10. These two graphs show that for this experiment the volume constraint results in a lower elastic force flux and a higher shear force flux. So with a volume constraint a block of material is easier to



(a) CMPM1 (b) CMPM2

Figure 9.5: Force flux as the result of shearing the material

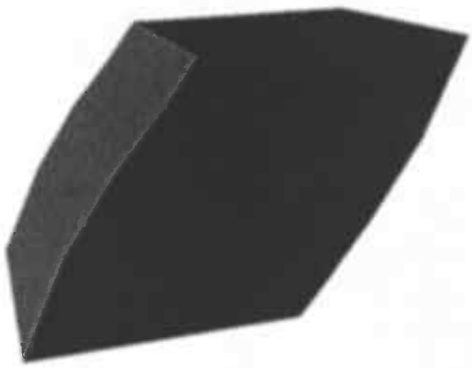


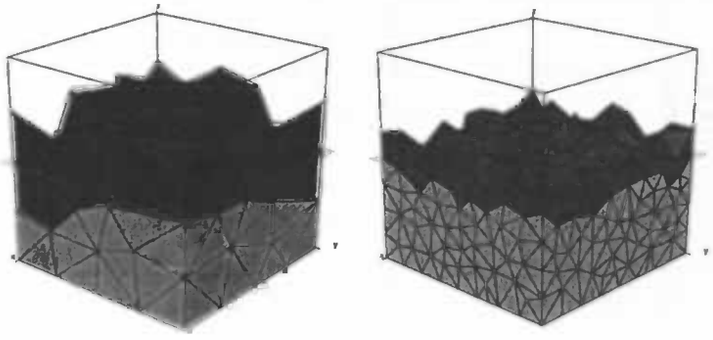
Figure 9.6: Shearing a cube

compress. This is somewhat counter-intuitive and could be explained as follows. The constraint method LINCS adds forces to the system that make the object extend in the lateral directions and these same forces perhaps make it easier to compress the object.

The material properties Young's modulus (E) and shear modulus (G) have been chosen such that $E = 3G$ holds. Based on the theory discussed in section 4.6 this would mean that the material is incompressible. Figure 9.11 however shows that the unconstrained simulated material clearly is not incompressible: a relative change of length of 0.24 results in a 24% volume difference.

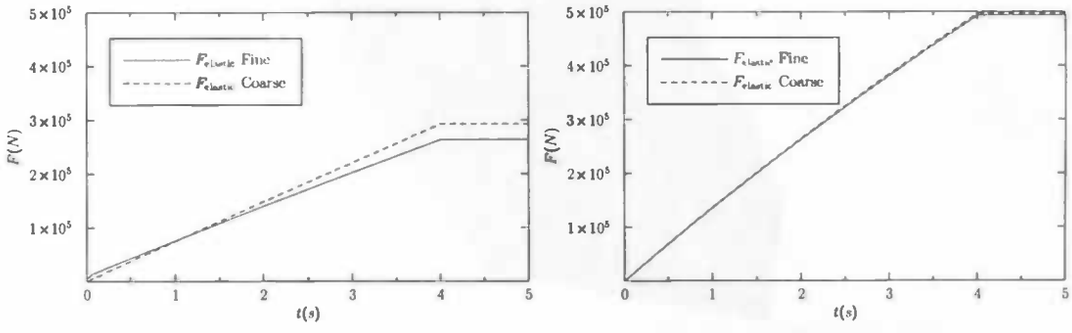
What figure 9.11 also shows is that the volume constraint method succeeds in constraining the volume of the cube while this cube is being compressed. During simulation with the volume constraint, the largest volume difference is $4.6 \times 10^{-5}\ m^3$, which is 0.46% of the total volume. The graphs in figure 9.11 are are created from CMPM1 data, but the results CMPM2 are similar.

An interesting observation is that with the volume constraint lateral extension does occur. Figure 9.12 illustrates this.



(a) Coarse (124 particles) (b) Fine (1034 particles)

Figure 9.7: Fine and coarse tetrahedrization



(a) CMPM1 (b) CMPM2

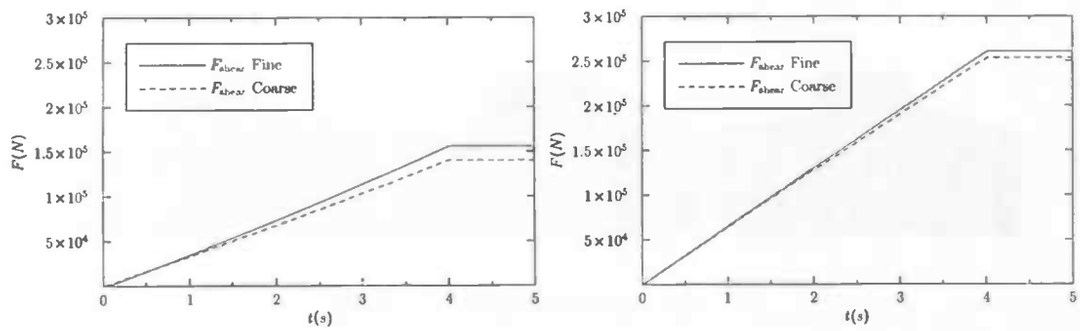
Figure 9.8: Influence of tetrahedrization on elastic force flux

9.7 Miscellaneous visual experiments

What follows is a number of experiments with the focus on the visual effect. Graphs like in the previous sections will not be produced for these experiments.

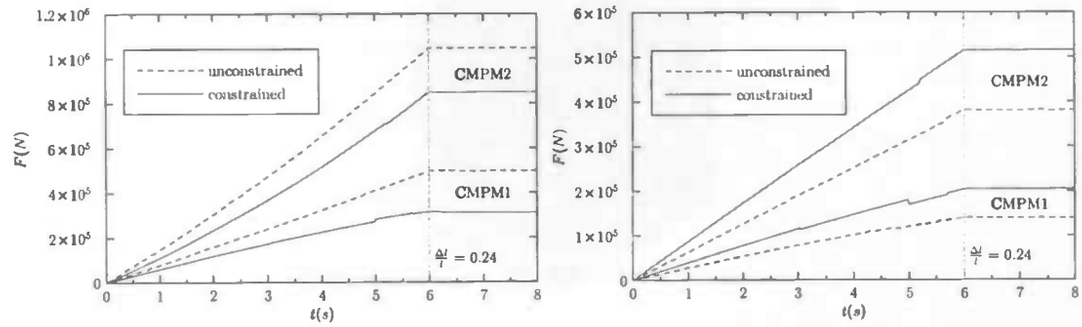
9.7.1 Twists

Again the rubber cube is subject of an experiment. This time it is twisted: the bottom particles are 'glued' to the ground and the particles in the upper face are slowly rotated. Figure 9.13 shows what this looks like. Eventually the top particles are released and the cube "wobbles" back into its original shape.



(a) CMPM1 (b) CMPM2

Figure 9.9: Influence of tetrahedrization on shear force flux



(a) Elastic force flux (b) Shear force flux

Figure 9.10: Influence of volume constraint on elastic and shear force flux

9.7.2 Sine movement

In all the previous experiments there was only simple movement in one direction. In this experiment one end of a thin beam is continuously moved up and down in a sine-like way. At first the thin beam shows regular sine waves, but eventually it becomes more and more irregular. The sequence of shots in figure 9.14 illustrate this.

For this thin beam a rather fine tetrahedrization was used; 4854 particles, 19268 tetrahedra and 27439 edges.

9.7.3 Mattress

The above sine movement can also be applied to a mattress-like object. The result can be seen in figure 9.15.

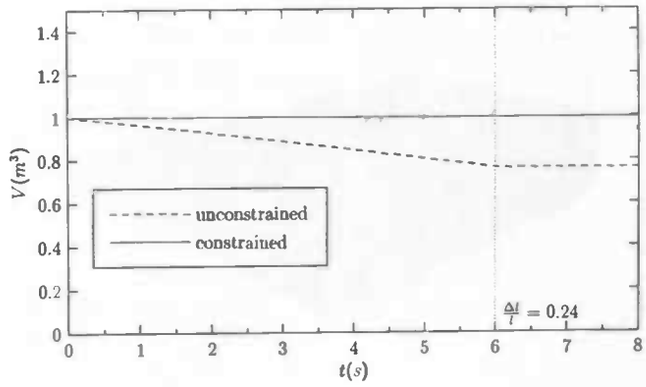


Figure 9.11: Volume of the cube with and without volume constraint

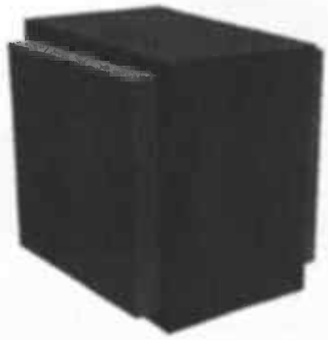


Figure 9.12: Lateral extension occurs when volume constraint is used

This same mattress-like object is also used in a simple experiment where it is draped over a block or tabletop. Figure 9.16 illustrates this.

9.8 Complex object

We will conclude with a slightly more complex object: a bunny. This 3D model was found on `http://www-c.inria.fr/Eric.Saltel/download/`, which has a collection of more than 30000 free 3D meshes. After tetrahedrization it has 7222 particles, 26692 tetrahedra and 39351 edges.

Unfortunately CMPM1 did not work on this model, it resulted in a unstable simulation. Most likely this is due to problems with the generation of the constrained Voronoi diagram (see section 6.4.3). CMPM 2 does work on this model and the sequence of figure 9.17 illustrates a simple experiment where the particles in the paws of the bunny are being pulled down. The paws are first a bit stretched and then the rest of the body follows. Note how the bunny's ears flap down in the last shot of the sequence of figure 9.17.

Figure 9.13: Twist simulation sequence

Figure 9.14: Sine movement of a thin beam

9.9 Speed

One of the goals for these kind simulations of deformable objects is real-time interaction. The simulations presented here are far from real-time. For example it takes 53 seconds on a reasonably fast machine[1] to do 16000 timesteps (1.6 simulated seconds) in the simulation of the hanging beam with its 1679 particles. That is for CMPM1, CMPM2 is even slower: a little more than 2 minutes to simulate the same 1.6 seconds.

There are two ways to remedy this situation. First of all, the current implementation was not designed for speed, but for flexibility. This flexibility probably produces some overhead, but it allowed us to easily test new models and approaches. A specialized implementation that, for example, takes advantage of SIMD instructions of modern processors will be much faster. Secondly, the integrator used for the experiments is leap-frog. Leap-frog is an explicit integrator and in particular for stiff materials small timesteps are required for stable simulations. For implicit integrators larger timesteps can be used. A fellow MSc. student, Egbert van der Es, is working on this and has implemented the implicit integration method of [BW98] for CMPM1 and with this integrator real-time simulations can be achieved on relatively new hardware.

[1] A workstation with a AMD Athlon64 3000+ processor

Figure 9.15: Sine movement applied to a mattress-like object

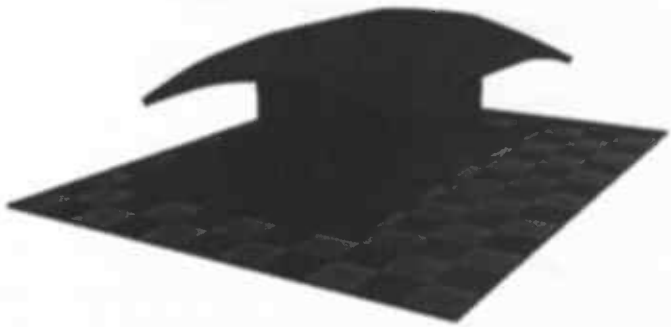


Figure 9.16: Mattress resting on a solid block

9.10 Summary of the results

To begin with, the experiments in section 9.6 have shown that the volume constraint method of chapter 3 succeeds in constraining the volume for particle-based models like CMPM1 and CMPM2.

The results from sections 9.3 and 9.4 show that the forces inside the simulated objects are well within limits for visual applications. Furthermore, the coarseness of the tetrahedrization seems not to material behaviour too much, he difference is acceptable for computer graphics applications. This means coarse models, which are less computationally intensive to simulate, will also show acceptable behaviour. The last few more visual experiments show that the simulated deformable objects also give good visual results. The objects respond to movement and interaction like one would expect.

All in all, the results have shown that this new approach to modelling deformable material has potential.

Figure 9.17: A rubber bunny being pulled down

10 Conclusion and Future work

This Master's thesis project started with an idea for extending mass-spring systems with a volume constraint. During implementation of this idea, it became clear that mass-spring systems have one important drawback: finding proper spring constants is difficult. In an effort to remedy this, an alternative particle model was developed. This model is based on Continuum Mechanics theory and allows to model a particular material with predictable material constants. Because of promising results the focus of this project was shifted from Constant volume simulation to this new model.

Voronoi diagrams play an in important role in this method. This turned out to be also one of the method's disadvantages. Even though good results were obtained for simple shapes, there were problems finding the bounded Voronoi diagram for constrained Delaunay tetrahedrizations of more complex shapes. To overcome this problem another model was developed that has the same approach, but does not need a Voronoi diagram. Yet this model was also set aside because in it assumptions were made that were not (necessarily) true. A third variation is being developed that tries to incorporate the good points of the previous two attempts. Egbert van Es, another MSc. student, is working on this.

Despite the problems of the first two variations, the results they produced were promising. To begin with, the forces inside the material in response to compressing and shearing of the object are well within the limits for applications in computer graphics. Also, the coarseness of the tetrahedrization did not influence the properties of the simulated material too much and this means coarser models can be used to speed up simulations. And perhaps even more importantly, the simulated deformable objects visually respond to interaction and movement like one would expect.

All experiments described in this document have been done with the simple leap-frag integration method. To obtain a stable simulation small timesteps had to be used, which results in high computational cost. For real-time simulations , the timestep needs to be increased, and one way to do this is to look at other integration methods. Egbert is working on this as well and has already implemented the implicit integration method of [BW98] for the first variation of our method with promising results.

So far we have simulated simple homogeneous isotropic material. Adding support for inhomogeneous material is not that hard and the third version of this method that is currently under development has the added advantage that it is fairly easy to incorporate anisotropy in it.

Finally, for real-time interaction with material in virtual environments techniques like collision detection need to be added. And of course for a virtual surgery application the question of how to implement cutting is an interesting subject to look into.

Bibliography

[BC00] David Bourguignon and Marie-Paule Cani. Controlling anisotropy in mass-spring systems. In *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, Springer Computer Science, pages 113–123. Springer-Verlag, aug 2000. Proceedings of the 11th Eurographics Workshop, Interlaken, Switzerland, August 21–22, 2000.

[BW98] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 43–54. SIGGRAPH, 1998.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[DKT95] Oliver Deussen, Leif Kobbelt, and Peter Tücke. Using simulated annealing to obtain good nodal approximations of deformable objects. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 30–43, 1995.

[Ede01] Herbert Edelsbrunner. *Geometry and topology for mesh generation*. Cambridge University Press, New York, NY, USA, 2001.

[EEH00] Bernhard Eberhardt, Olaf Etzmuß, and Michael Hauth. Implicit-explicit schemes for fast animation with particle systems. pages 137–154, 2000.

[EGS03] Olaf Etzmuss, Joachim Gross, and Wolfgang Strasser. Deriving a particle system from continuum mechanics for the animation of deformable objects. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):538–550, 2003.

[GR] Christophe Geuzaine and Jean-François Remacle. Gmsh website, `http://www.geuz.org/gmsh`.

[Hau04] Michael Hauth. *Visual Simulation of Deformable Models*. PhD thesis, Wilhelm-Schickard-Institut für Informatik, University of Tübingen, Germany, July 2004.

[HBBF97] B. Hess, H. Bekker, H.J.C Berendsen, and J.G.E.M. Fraaije. Lincs: A linear constraint solver for molecular simulations. *Journal of Computational Chemistry*, 18(12):1463–1472, 1997.

[JL04] Il-Kwon Jeong and Inho Lee. An oriented particle and generalized spring model for fast prototyping deformable objects. The Eurographics Association 2004, 2004. Short Texts, `http://eg04.inrialpes.fr/Programme/ShortPresentation/PDF/short38.pdf`.

[LPC95] Jean Louchet, Xavier Provot, and David Crochemore. Evolutionary identification of cloth animation models. In Dimitri Terzolpoulos and Daniel Thalmann, editors, *Computer Animation and Simulation '95*, LNCS, pages 44–54, Maastricht, Netherlands, 2-3 September 1995. Springer-Verlag. Proceedings of the Eurographics Workshop.

[Lua] Lua programming language, website: `http://www.lua.org`.

[MHTG05] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM Trans. Graph.*, 24(3):471–478, 2005.

[MKN+04] Matthias Müller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa. Point based animation of elastic, plastic and melting objects. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, Aug 2004.

[NT98] L. P. Nedel and D. Thalmann. Real time muscle deformations using mass-spring systems. In *CGI '98: Proceedings of the Computer Graphics International 1998*, pages 156–165, Washington, DC, USA, 1998. IEEE Computer Society.

[Qhu] Qhull. Qhull website, `http://www.qhull.org/`.

[RCB77] J.P. Rijckaert, G. Ciccotti, and H.J.C. Berendsen. Numerical integration of the cartesion equations of motion of a system with constraints: Molecular dynamics of *n*-alkanes. *Journal of Computational Physics*, 23:327–341, 1977.

[RNP98] A. Radetzky, A. Nurnberger, and D. Pretschner. The simulation of elastic tissues in virtual medicine using neuro-fuzzy systems. In *Proc. of Medical Imaging 1998*, volume 3335 of *SPIE Proceedings*, 1998.

[She96] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust predicates for computational geometry. website: `http://www.cs.cmu.edu/~quake/robust.html`, 1996.

[She98] Jonathan R Shewchuk. The geometry junkyard archive on circumcenters. website: `http://www.ics.uci.edu/~eppstein/junkyard/circumcenter.html`, 1998.

[Si] Hang Si. Tetgen website, `http://tetgen.berlios.de/`.

[toL] tolua++, website: `http://www.codenix.com/~tolua/`.