Implementation of Deformable Objects

Micky K Christensen
micky@mkc.dk

Anders Fleron
afleron@knights.dk

Department of Computer Science
University of Copenhagen

May 1, 2004

Abstract

With in depth focus on deformable surfaces, we have successfully implemented the method of elastically deformable models presented by Terzopoulos et al. in 1987. Our implementation runs interactively, even with high particle concentrations. We have created convincing physics-based animations, supported by our generic XML scene interface. The implementation is compared against a fast relaxation based particle system, and not only exhibits a more realistic behavior of deformation, but also makes a tough competitor regarding performance. We reveal all the details on the theory and the implementation process, such as the infamous stiffness matrix.

Acknowledgements

First of all we would like to thank our places of work, friends, and especially our closest families, for giving us the space to complete this project.

A huge thank must go to the core team of the OpenTissue project group at DIKU, especially Kenny Erleben and Henrik Dohlmann for calming our frustrations from time to time, and always providing us with great insights in the world of physics.

Finally, we wish to thank our supervisor, Jon Sporring, for the time he has put into this project, for helping us gaining the needed theoretical knowledge, for reviewing the project, and for providing helpful guidelines throughout the entire process.

All Your help and feedbacks have been most appreciated.

Contents

1 Introduction 1

1.1 Goals 1
1.2 Reading Guide 1
1.2.1 Mathematical notation 2
1.2.2 Pseudo code notation 2
1.2.3 Overview of the paper 2
1.3 The Enclosed CD 2

2 Theory 3

2.1 The Elastically Deformable Model 3
2.2 Differential Geometry 4
2.2.1 Manifolds 4
2.2.2 Deformation energy 4
2.2.3 Forms 4
2.2.4 First fundamental form 5
2.2.5 Second fundamental form 8
2.3 Approximation of the Elasticity 9
2.3.1 The elastic force 10
2.3.2 Variational calculus 10
2.3.3 Deriving the elastic force 12
2.4 Finite Differences 16
2.4.1 Functions of one variable 16
2.4.2 Functions of two variables 19

3 Execution and Practice 21

3.1 Discretization 21
3.1.1 Difference operators 24
3.2 Elasticity 26
3.2.1 Intrinsic elastic force 26
3.2.2 Tension 27
3.2.3 Curvature 28
3.2.4 Alternate method 30
3.3 The Stiffness Matrix 31
3.3.1 The alpha stencil 32
3.3.2 The beta stencil 34
3.3.3 The K matrix 36
3.3.4 Numerical instabilities 38
3.4 Boundaries 38
3.4.1 Projection method 38
3.4.2 Continuous boundaries 40

3.4.3 Stiffness boundary conditions 40
3.5 External Forces 41
3.5.1 Gravity 42
3.5.2 Viscosity 43
3.5.3 Spring 43
3.5.4 User controlled forces 44
3.6 Collision 47
3.6.1 Penalty collision force 47
3.6.2 Projection 52
3.6.3 Self-intersection 53
3.6.4 Geometric collision detection 54
3.6.5 Using collision handling 54
3.7 Surface Normals 54
3.7.1 Surfaces 55
3.7.2 Solids 56
3.8 Numerical Integration 58
3.8.1 Integration schemes 58
3.8.2 Semi-implicit integration 60
3.8.3 Explicit integration 64
3.9 Visualization 66
3.9.1 Direct visualization 67
3.9.2 Particle density 68
3.9.3 Visual debug 70
3.9.4 Texture coordinates 71
3.9.5 Material descriptors 71

4 Implementation 73

4.1 Program Description 73
4.1.1 Class hierarchy 73
4.1.2 Class associations 73
4.1.3 Simulation procedure 75
4.2 Compiling and Building 77
4.2.1 3rd party dependencies 77
4.2.2 User controlled behavior 77
4.3 User Guide 80
4.4 Scene Configurations 81
4.4.1 Common principles 82
4.4.2 Material container 83
4.4.3 External forces 84
4.4.4 Rigid objects 86
4.4.5 Deformable bodies 87
4.5 Examples 89
4.5.1 Models 89
4.5.2 Objects 92

5 Comparison 94

5.1 Standard of Reference 94
5.1.1 Constraint types 94
5.1.2 Comparison models 95
5.2 Comparison Cases 97
5.2.1 Maintenance 98
5.2.2 Accuracy 99
5.2.3 Stability 99
5.2.4 Performance 101

6 Results 104

6.1 Visual Test Results 104
6.1.1 Fundamental principles 104
6.1.2 Deformable open surfaces 106
6.1.3 Deformable semi-closed surfaces 108
6.1.4 Deformable Solids 110
6.2 Evaluation 111
6.2.1 Physical properties 111
6.2.2 Contribution 112
6.2.3 The process 113

7 Future Work 114

7.1 Work in Progress 114
7.1.1 Fixing curvature problems 114
7.1.2 Instabilities in deformable solids 114
7.1.3 Self intersection 114
7.1.4 Interface of the implementation 114
7.2 Ideas 115
7.2.1 Collision between deformable models 115
7.2.2 Hybrid model 115
7.2.3 Generic mesh structure 115

8 Conclusion 116

References 117

Appendices 119

A.1 Comparison Material 119
A.2 Results Gallaria 127

1 Introduction

The idea of simulating non-rigid objects is not an option that should be avoided, because of either difficult mathematics or the heavy computational nature of the subject. As a matter of fact the only place a rigid object-only environment can exist, without anyone being suspicious about the lack of realism, is within a virtual world, e.g. in computer games.

Working with a dynamic tissue simulator, it would not be much of a real life experience for possible surgeons, if all organs were made of rigid bodies. This is one of the good reasons why soft and deformable bodies are important, when trying to simulate the real world. Equally important is the way in which we control the non-rigid bodies. We are not willing to offer everything at the expense of great realism. We want to be able to simulate non-rigid bodies at a minimum maintenance level. Terzopoulos et al. [21] wrote a breakthrough article in 1987 about elastically deformable models. It is our intent to interpret and understand the contents of this article, in order to enable us to implement the described method. Our implementation will benefit the ongoing OpenTissue project [6]. In spring 2003 we attended an introductory course on simulation of deformable objects, based on [21], that triggered our interests doing the implementation.

Last, but not least, we will discuss our interpretation of the method, pitfalls in the implementation, along with evaluations of some alternatives. This paper is meant as a help to other students, and alike, when they someday decide to begin the journey of implementing Terzopoulos' deformable objects. We will also explain theoretical issues in greater detail, which hopefully gives others easier access to the understanding of this method.

1.1 Goals

We will consider our work a success, if we are able to understand the method about elastically deformable models [21], to a degree which grants us the ability to implement it. Our implementation must result in a running application, demonstrating various effects of this particular method. Comparison against another implementation that addresses the same areas will be performed. The criteria that the comparison will be based upon are ones such as realism, performance, and maintenance. It is our expectation that our implementation will do well in these areas.

If problems arise during the implementation, we must be able to address and solve them correctly, to work around them, or at least to argue adequately why we were not able to find a solution. In the end the reader is to judge if our interpretation of the article is a success.

1.2 Reading Guide

There are several ways to read this paper, depending on the level of understanding that the reader wants of the subject. If a more thorough understanding of the underlying topics is needed, the reader should start with the main theory in chapter 2 *Theory*, and then move on to the implementation related topics. If the reader is only interested in the method of implementation, chapter 4 *Implementation* could be used as a starting point. A description of the simulation loop is presented in this chapter, and references are made from within the loop to the relevant parts of the paper.

The reader should possess mathematical maturity, and also have a good understanding of linear algebra and mathematical analysis, to be able to get the most out of the theory that is discussed. It also helps to have a healthy interest in physically-based modeling.

1.2.1 Mathematical notation

Different authors use different notations for the same things. The table below lists the different mathematical notations, which we have used throughout the paper. All functions will be formatted as the dimension of the range, e.g. $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ and $\mathbf{f} : \mathbb{R}^2 \rightarrow \mathbb{R}^3$.

Notation	Identification	Description
$\mathbf{v}$, $\mathbf{p}$	Lowercase, bold	Vector, position
$\mathbf{M}$, $\boldsymbol{\alpha}$	Roman, uppercase, bold Greek, lowercase, bold	Matrix, Tensor
t	Lowercase, italic	Scalar
$[a_1, a_2, .., a_n]$	Comma separated scalars in brackets	n -dimensional grid index
$\underline{\mathbf{v}}$	Lowercase, bold, underlined	Vector collection unwrapped from grid
$\|\mathbf{v}\|_2$		l_2 norm (vector norm)
f_x		The function f differentiated with respect to x
f_{xx}		The function f differentiated twice with respect to x

1.2.2 Pseudo code notation

The notation in the pseudo code examples uses the ANSI C/C++ syntax, where complex expressions and functionalities are replaced by symbols relating to the actual text of the section. The pseudo code in the listings has been syntax highlighted for easy recognition.

1.2.3 Overview of the paper

Chapter 2 *Theory* goes through the mathematics of the method that the reader is not assumed to posses, which include Differential Geometry and Variational calculus. The chapter also gives a survey on the concept of finite differencing that is used to estimate derivates.

Chapter 3 *Execution and Practice* takes the reader through all the areas related to the understanding of the method.

Chapter 4 *Implementation* presents our implementation, including a section on how our simulation loop has been implemented, and how to manually build the implementation. The last section gives examples of how some of the implemented generalizations have been produced.

Chapter 5 *Comparison* compares the implementation against another particle system, which is able to simulate cloth and soft bodies. During the comparison we will also perform some tests of our implementation.

Chapter 6 *Results* shows many of the results that we have produced with our implementation, both the great ones, but also the more unpleasant ones. We finalize the chapter with an evaluation of our work.

Chapter 7 *Future Work* states both our work in progress and some interesting ideas to extend the project.

1.3 The Enclosed CD

We have included a supplemental CD with the project. The CD has been created to be accessible under Windows 2000/XP and Linux. All files can be read on these platforms, except for the executables, which will only run under Win32. The file `"readme.txt"` in the root of the CD, contains a description of structure and contents. Briefly stated, it contains the source code of the implementation, images, motion sequences, and running demonstrations.

2 Theory

This project has its starting point from the physical model of elastically deformable objects. An explanation of how this model arises from physical theory is beyond the scope of this project. The object of focus for this chapter is to take a closer look at how this model is manipulated into an approximation that can be implemented on a computer. The subsequent sections aim at giving the reader a better understanding of how this is done.

In the first section the physical model will be presented to the reader. We will then present theory on differential geometry, on which the approximation is based. An explanation on how this theory is used to approximate the elasticity term of the physical model is presented. We will end this chapter with a section on finite differences, which is used to obtain a general way to handle the derivatives in the model.

2.1 The Elastically Deformable Model

In [21] a model to simulate elastic deformation is presented, which is based firmly in physics. The model is derived from Newtonian mechanics and takes into account external forces acting on the deformable object. The model that is the basis of this paper is

$$\frac{\partial}{\partial t}\left(\mu \frac{\partial \mathbf{r}}{\partial t}\right) + \gamma \frac{\partial \mathbf{r}}{\partial t} + \frac{\delta \varepsilon(\mathbf{r})}{\delta \mathbf{r}} = \mathbf{f}(\mathbf{r}, t)\,. \tag{2.1.1}$$

As can be seen from (2.1.1) the model consists of three terms on the left hand side and one term on the right hand side. The latter represents the total external force working on the object at time t. The total force comes from adding up the individual external forces working on the object. The first two terms on the left hand side are controlled by two constants μ and γ, which are the mass and damping parameters of the model, respectively. The type of object that we will be looking at here will mainly be deformable surfaces, but the model can also be extended to work for curves and solids. The object is thought of, as a collection of particles making up the structure. The symbol $\mathbf{r}$ represents the position of the particle $\mathbf{a}$ in an object. If the object is a surface, it will contain particles in two dimensions, which can be considered as orthogonal directions. These directions are designated $\mathbf{a} = (a_1, a_2)$. If the object was a solid, the directions would be $\mathbf{a} = (a_1, a_2, a_3)$. To make the object move according to the model, it should evolve over a period of time, thus $\mathbf{r}(\mathbf{a}, t)$ represents the position of the particle $\mathbf{a}$ in an object at time t. The third term in the model is

$$\frac{\delta \varepsilon(\mathbf{r})}{\delta \mathbf{r}}\,, \tag{2.1.2}$$

where all the elasticity and bending energy of the object is handled. (2.1.2) is what is called a variational derivative, and basically describes the potential energy that is created in an object that is deformed away from its natural resting shape. The rest of the theory section deals primarily with approximating this term into something that can be readily computed.

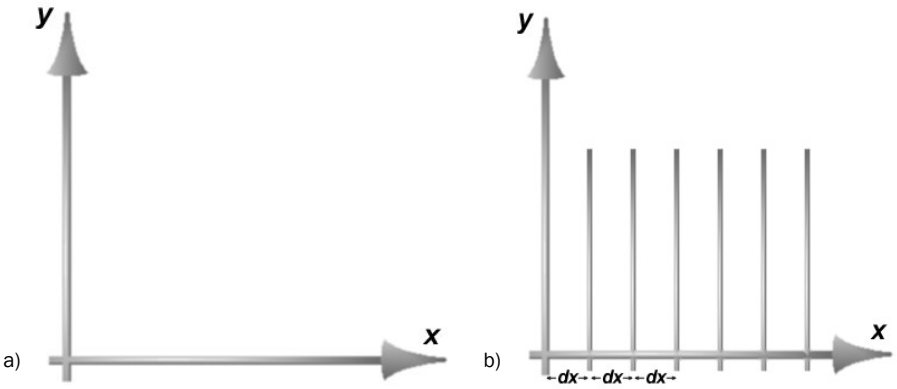


Figure 2.1 Coordinate systems and forms **a)** Two-dimensional coordinate system **b)** Form in the x -direction

2.2 Differential Geometry

Differential geometry is a complex area of the combined study of mathematical calculus and analysis. Differential geometry is the study of manifolds and their metrical descriptions. The shape of a manifold is uniquely described by its different fundamental forms. In [21] the fundamental forms are used without an in-depth explanation. It is our intension to elucidate the different aspects of the fundamental forms, without the need to study differential geometry. Before we continue with the subject, let us introduce a brief survey on manifolds.

2.2.1 Manifolds

The shape of an object can be characterized according to its mathematical description. A curve is a one-dimensional manifold, or one-manifold, and a surface is a two-manifold, etc. Focusing on a two-manifold, it is a surface embedded in Euclidean three-space, such that the infinitesimal neighborhood around a point of the surface is locally topological equivalent, thus a manifold is differentiable. We distinguish between open (unbounded or non-compact) manifolds, e.g. a plane, a paraboloid, and closed (bounded or compact) manifolds, e.g. sphere, torus. The shape of a deformable surface in this paper, can be any type of two-manifold. Further details on manifolds and their applications can be found in [27, 2].

2.2.2 Deformation energy

Terzopoulos et al. [21] uses differential geometry to handle the deformation energy. In this section an introduction to the differential geometry that is needed to understand deformation energy, is presented.

Deformable objects are thought of as having a natural shape, in which the energy of deformation, or the elastic energy, is zero, see Figure 2.5. When the object is deformed, deformation energy is introduced because it deviates from its natural shape. Differential geometry is used as a mean to measure the difference between the deformed object shape and the natural shape.

2.2.3 Forms

Let us focus on how measurement is used at the most basic level in differential geometry. This is done by the use of something called *forms*. Forms are basically units of measurement on mathematical rulers. The way to think of this is as a pile of planes stacked on top of each other, in some

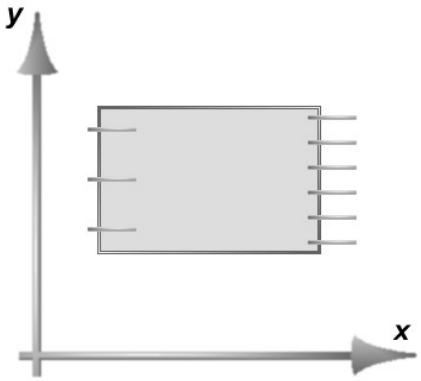


Figure 2.2 Increasing number of planes in the y-direction

direction, with a uniform distance between them. In Figure 2.1a, a simple two-dimensional coordinate system is displayed, with two labeled directions x and y. In Figure 2.1b, some vertical lines have been overlaid with the coordinate system. These lines represent the form known as dx. If the spatial dimension was added it could be seen that these lines are in fact planes. The value of dx tells us of the spacing between the planes, and the symbol dx shows that this form is used for *measuring* in the x-direction. The form dy does the same for the y-direction. If measurement of a vector is desired with the form dx, simply count how many lines (or planes in 3d) of the form that the vector is piercing. This will return the length of the vector in the x-direction. More complex forms than these basic forms need not have the same direction everywhere, as we shall see later on. In fact, the purpose of introducing differential geometry here is to be able to do measurement on a curved surface. Here is an example of a more elaborate form, which is given the label $\tilde{\varphi}$ [14] p. 96, and defined as

$$\tilde{\varphi} = x\,dy\,, \tag{2.2.1}$$

which is a form that has the direction of the basic form dy, but with something extra. One unit in the y-direction (dy) will be divided by x planes. This means that moving in the positive x-direction will make one unit in the y-direction be divided by more planes. This can be seen by looking at a rectangle in a coordinate system with the form inserted, as depicted in Figure 2.2. We take a look at the right and the left side of the rectangle, and see that the number of planes in the y-direction does actually rise with an increasing x.

2.2.4 First fundamental form

As previously announced we are, among other things, interested in a form that can be used for measurement on a curved surface. In our implementation of the deformable model we use a parameterized surface to represent the shape of a surface patch. A surface patch in space can be defined by a function $\mathbf{Q} : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ of two variables (or directions) u and v, running in the interval $[0 : 1]$. The underlying coordinate functions of $\mathbf{Q}$ are

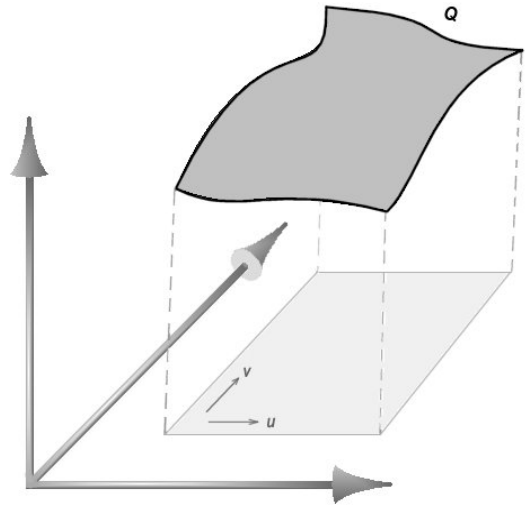


Figure 2.3 A two-manifold $\mathbf{Q}$

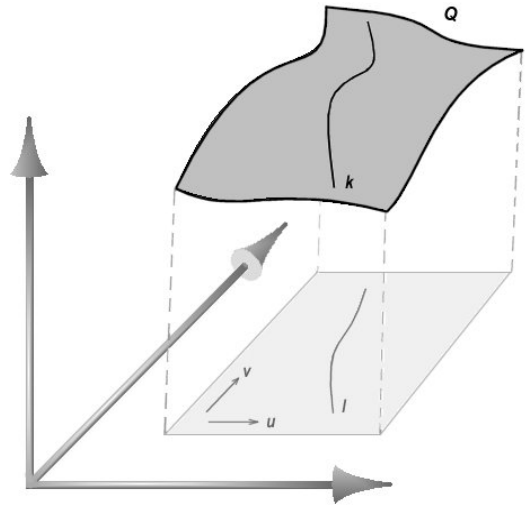


Figure 2.4 A parameterized curve $\mathbf{k}$

$$\begin{aligned} x &= f(u,v) \\ y &= g(u,v) \\ z &= h(u,v). \end{aligned} \tag{2.2.2}$$

The function $\mathbf{Q}$ describes a two-manifold in space with domain (u,v), as depicted in Figure 2.3.

Let us look at a curve $\mathbf{l}$ running in the (u, v) domain in some interval $[a : b]$. It will be a one-variable parameterized curve with coordinate functions

$$\begin{aligned} u &= \varphi(t) \\ v &= \psi(t), \end{aligned} \tag{2.2.3}$$

where t is the parameter. This curve will have a corresponding curve $\mathbf{k}$ that runs on the surface $\mathbf{Q}$, that is

$$\mathbf{k} = \mathbf{Q}(\varphi(t), \psi(t)), \tag{2.2.4}$$

with

$$\begin{aligned} x &= f(\varphi(t), \psi(t)) \\ y &= g(\varphi(t), \psi(t)) \\ z &= h(\varphi(t), \psi(t)). \end{aligned} \tag{2.2.5}$$

In Figure 2.4 a visual representation of the two curves, $\mathbf{l}$ and $\mathbf{k}$, can be seen. Each of the coordinate functions can then be differentiated with respect to the parameter t, to find the change of the curve across the surface $\mathbf{Q}$. Using the chain rule of differentiation yields

$$\begin{aligned} \frac{dx}{dt} &= \frac{\partial x}{\partial u}\frac{du}{dt} + \frac{\partial x}{\partial v}\frac{dv}{dt} \\ \frac{dy}{dt} &= \frac{\partial y}{\partial u}\frac{du}{dt} + \frac{\partial y}{\partial v}\frac{dv}{dt} \\ \frac{dz}{dt} &= \frac{\partial z}{\partial u}\frac{du}{dt} + \frac{\partial z}{\partial v}\frac{dv}{dt}, \end{aligned} \tag{2.2.6}$$

where du and dv are forms in the u and v directions, respectively. This can be compressed into a derivative of $\mathbf{Q}$ on the curve

$$\frac{d\mathbf{Q}}{dt} = \frac{\partial \mathbf{Q}}{\partial u}\frac{du}{dt} + \frac{\partial \mathbf{Q}}{\partial v}\frac{dv}{dt}. \tag{2.2.7}$$

Let us assume that $s(t)$ represents the arc length along the curve $\mathbf{k}$. Generally, arc length along a curve $\boldsymbol{\lambda}$ is defined as

$$s \equiv \int_{\lambda} |d\boldsymbol{\ell}|, \tag{2.2.8}$$

where $d\boldsymbol{\ell}$ is a differential displacement vector along the curve $\boldsymbol{\lambda}$ [26]. Here we use

$$d\boldsymbol{\ell} \equiv \frac{d\mathbf{Q}}{dt}. \tag{2.2.9}$$

If $s(a) = 0$, then the arc-length from a to t along $\mathbf{k}$ is

$$s(t) = \int_a^t \left|\frac{d\mathbf{Q}}{dt}\right|. \tag{2.2.10}$$

Finally by differentiating the arc-length with respect to t and squaring the result, we arrive at what is known as the first fundamental form of the surface

$$\begin{aligned}\left(\frac{ds}{dt}\right)^2 &= \left|\frac{d\mathbf{Q}}{dt}\right|^2 = \frac{d\mathbf{Q}}{dt}\cdot\frac{d\mathbf{Q}}{dt} \\ &= E\left(\frac{du}{dt}\right)^2 + 2F\frac{du}{dt}\frac{dv}{dt} + G\left(\frac{dv}{dt}\right)^2,\end{aligned} \tag{2.2.11}$$

where

$$\begin{aligned}E &= \left(\frac{\partial\mathbf{Q}}{\partial u}\right)^2 \\ F &= \frac{\partial\mathbf{Q}}{\partial u}\cdot\frac{\partial\mathbf{Q}}{\partial v} \\ G &= \left(\frac{\partial\mathbf{Q}}{\partial v}\right)^2.\end{aligned} \tag{2.2.12}$$

E, F, and G are the coefficients of the first fundamental form. These coefficients are what we are interested in, for the implementation of the deformable model. This is because the coefficients them selves, are representations of the surface with regard to arc length. The first fundamental form can be used as a local property of a point on the surface, such that the point it self, knows about arc lengths in an area around it. Since the implemented deformable model will evolve in time, a point can detect changes in arc length in an area around it, by monitoring the coefficients of the first fundamental form. The coefficients E, F, and G can be assembled into what is called the metric tensor $\mathbf{G} : \mathbb{R}^{2\times2}$ of the surface. This is defined as

$$\mathbf{G} = \begin{bmatrix}E & F\\ F & G\end{bmatrix} = \begin{bmatrix}\frac{\partial\mathbf{Q}}{\partial u}\cdot\frac{\partial\mathbf{Q}}{\partial u} & \frac{\partial\mathbf{Q}}{\partial u}\cdot\frac{\partial\mathbf{Q}}{\partial v}\\ \frac{\partial\mathbf{Q}}{\partial v}\cdot\frac{\partial\mathbf{Q}}{\partial u} & \frac{\partial\mathbf{Q}}{\partial v}\cdot\frac{\partial\mathbf{Q}}{\partial v}\end{bmatrix}. \tag{2.2.13}$$

2.2.5 Second fundamental form

Similarly, as we are able to measure arc lengths in an area around a point on a surface, we will also like to know something about the curvature in this area. This is where the second fundamental form comes in. By looking at a curve running on the surface, we are now interested in the second derivative of the surface along this curve. We know from the previous section that

$$\frac{d\mathbf{Q}}{dt} = \frac{\partial\mathbf{Q}}{\partial u}\frac{du}{dt} + \frac{\partial\mathbf{Q}}{\partial v}\frac{dv}{dt}, \tag{2.2.14}$$

and by differentiating a second time we find that

$$\begin{aligned}\frac{d^2\mathbf{Q}}{dt^2} = \frac{\partial\mathbf{Q}}{\partial u}\frac{d^2u}{dt^2} + \frac{\partial\mathbf{Q}}{\partial v}\frac{d^2v}{dt^2} + \frac{\partial^2\mathbf{Q}}{\partial u^2}\left(\frac{du}{dt}\right)^2 + \\ 2\frac{\partial^2\mathbf{Q}}{\partial u\partial v}\frac{du}{dt}\frac{dv}{dt} + \frac{\partial^2\mathbf{Q}}{\partial v^2}\left(\frac{dv}{dt}\right)^2.\end{aligned} \tag{2.2.15}$$

In short hand notation this can be written as

$$\mathbf{Q}_{tt} = \mathbf{Q}_u u_{tt} + \mathbf{Q}_v v_{tt} + \mathbf{Q}_{uu}{u_t}^2 + 2\mathbf{Q}_{uv}u_t v_t + \mathbf{Q}_{vv}{v_t}^2. \tag{2.2.16}$$

The next step is to look at the normal to the curve

$$\mathbf{n} = \mathbf{Q}_u \times \mathbf{Q}_v, \tag{2.2.17}$$

and at the projection a of $\mathbf{Q}_{tt}$ on $\mathbf{n}$

$$a = \frac{\mathbf{n} \cdot \mathbf{Q}_{tt}}{|\mathbf{n}|}. \tag{2.2.18}$$

By inserting the expression of $\mathbf{Q}_{tt}$ from (2.2.16) into (2.2.18) we arrive at an expression for the second fundamental form of the surface. Notice that the first two terms $\mathbf{Q}_u u_{tt}$ and $\mathbf{Q}_v v_{tt}$ of (2.2.16) vanish in the dot product with $\mathbf{n}$, because they are perpendicular to $\mathbf{n}$. The uniform scaling of the coefficients of the second fundamental form with $|\mathbf{n}|^{-1}$ is also disregarded, since we are just looking for some measure of the curvature, and this measure is just as good without the scaling

$$a = L\left(\frac{du}{dt}\right)^2 + 2M\frac{du}{dt}\frac{dv}{dt} + N\left(\frac{dv}{dt}\right)^2, \tag{2.2.19}$$

where

$$\begin{aligned} L &= \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial u^2} \\ M &= \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial u \partial v} \\ N &= \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial v^2}. \end{aligned} \tag{2.2.20}$$

The coefficients L, M, and N can be assembled into a tensor $\mathbf{B} : \mathbb{R}^{2\times 2}$ that is called the curvature tensor of the surface. This is defined as

$$\mathbf{B} = \begin{bmatrix} L & M \\ M & N \end{bmatrix} = \begin{bmatrix} \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial u^2} & \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial u \partial v} \\ \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial v \partial u} & \mathbf{n} \cdot \frac{\partial^2 \mathbf{Q}}{\partial v^2} \end{bmatrix}. \tag{2.2.21}$$

Like with the first fundamental form, the coefficients (2.2.20) are enough to monitor changes in curvature around any point on the surface. Together, the two fundamental forms represent a good way to monitor deformation in a deformable surface. In the implementation of the model it is the difference of fundamental forms over time that will be used to represent how deformation energy changes as the surface deforms.

2.3 Approximation of the Elasticity

In the continuous deformable object model (2.1.1), the external forces working on the object are balanced against a number of internal forces. One of these internal forces deals with the elastic energy that is produced, when the shape of an elastically deformable object is different from its rest state. For example when a deformable object is compressed, as depicted in Figure 2.5, the particles at the deformation will have an increased elastic energy. This will result in an elastic force that will make the particles want to seek towards the natural rest state of the object. This agrees with the behavior of real-life ideally elastic objects, which will always return to their natural shape upon deformation. The internal force of elasticity is expressed as

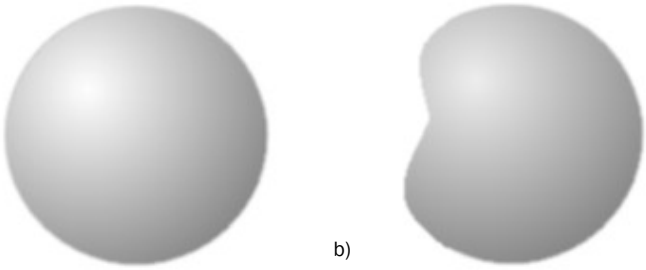


Figure 2.5 A deformable sphere **a)** No deformation, natural shape **b)** Deformation, increased energy

$$\frac{\delta \varepsilon(\mathbf{r})}{\delta \mathbf{r}}, \tag{2.2.22}$$

which is a variational derivative of the potential energy of deformation $\varepsilon(\mathbf{r})$ [21]. It is this potential energy that we wish to approximate.

2.3.1 The elastic force

A simple way to model the energy of a deformation is to look at the difference between the fundamental forms of the deformed body, and the fundamental forms of the body initially at rest. We use the following expression as the deformation energy

$$\varepsilon(\mathbf{r}) = \int_{\Omega} \sum_{i,j=1}^{2} \left(\eta_{ij} \left(G_{ij} - G_{ij}^{0} \right)^2 + \xi_{ij} (B_{ij} - B_{ij}^{0})^2 \right) da_1 da_2 , \tag{2.2.23}$$

where Ω is the domain of the elastically deformable body, and η, ξ are weighting matrices that will control the influences of the first and second fundamental form, respectively. The tensors $\mathbf{G}^0$ and $\mathbf{B}^0$, are the metric and curvature tensors, (2.2.13) and (2.2.21), of the body in its natural shape. To have the object evolve towards its natural shape, an expression that minimizes the energy is desirable. This is because an elastically deformable object, in its natural shape, will have zero deformation energy. For this purpose the mathematical tool of variational calculus is needed, and it will briefly be introduced in the following section.

2.3.2 Variational calculus

A branch of mathematics called calculus of variations [11] can be applied to find extremes in physical problems, and that is exactly what we need here. Usually, finding extremes for simple functions, involves differentiating the function, and finding the value in the domain that gives the function a zero slope, when evaluated in the derivative. What we want to do in calculus of variations, is to look at an integral of the form

$$I(y) = \int_{a}^{b} F(x, y, y') \, dx , \tag{2.2.24}$$

where $I(y)$ is called a functional. The task is now to find a function $y(x)$ that gives the functional its highest or lowest value. We are looking in a domain of functions, to find a function that is an extreme of the functional. To define an extreme of a functional we say that when the *variation* of the functional is zero, we have found an extreme. The variation of the functional equal to zero is written as

$$\delta I = 0\,. \tag{2.2.25}$$

It is assumed that a function $\overline{y}$ exists that makes the functional extremal. By applying a small variation to this function, and handing it to the functional, it is possible to figure out when the variation of the functional equals zero. The variation of $\overline{y}$ can be written as

$$\delta\overline{y} = \overline{y} + \varepsilon\eta\,, \tag{2.2.26}$$

where ε is some constant in a small interval around zero, and η is the variating function of which it is only demanded that

$$\eta(a) = \eta(b) = 0\,. \tag{2.2.27}$$

The variation (2.2.26) can now be inserted into the functional

$$I(\overline{y} + \varepsilon\eta) = \int_a^b F(x, \overline{y} + \varepsilon\eta, \overline{y}' + \varepsilon\eta')\,dx\,. \tag{2.2.28}$$

The next step is to make a function of one variable $J(\varepsilon)$, with the following two properties:

1. $J(\varepsilon)$ is a differentiable function in a small interval around $\varepsilon = 0$
2. $J(0)$ is a local extreme of J

We now define

$$J(\varepsilon) = I(\delta\overline{y})\,, \tag{2.2.29}$$

such that J is the functional evaluated at the variation of $\overline{y}$. From property 2 of J, we know that

$$\frac{d}{d\varepsilon}J = 0\,, \tag{2.2.30}$$

which can now be used in (2.2.29), such that we differentiate the variation of the functional with respect to ε

$$\begin{aligned}\frac{d}{d\varepsilon}I(\delta\overline{y}) &= \frac{d}{d\varepsilon}\int_a^b F(x, \delta\overline{y}, \delta\overline{y}')dx \\ &= \int_a^b \frac{d}{d\varepsilon}F(x, \delta\overline{y}, \delta\overline{y}')dx = 0,\end{aligned} \tag{2.2.31}$$

assuming F is differentiable. By applying the chain-rule inside the integral we get

$$\begin{aligned}&\int_b^a \left(\frac{\partial F}{\partial x}\frac{\partial x}{\partial\varepsilon} + \frac{\partial F}{\partial\delta\overline{y}}\frac{\partial\delta\overline{y}}{\partial\varepsilon} + \frac{\partial F}{\partial\delta\overline{y}'}\frac{\partial\delta\overline{y}'}{\partial\varepsilon}\right)dx \\ &= \int_b^a \left(\frac{\partial F}{\partial\delta\overline{y}}\frac{\partial\delta\overline{y}}{\partial\varepsilon} + \frac{\partial F}{\partial\delta\overline{y}'}\frac{\partial\delta\overline{y}'}{\partial\varepsilon}\right)dx \\ &= \int_b^a \left(\frac{\partial F}{\partial\delta\overline{y}}\eta + \frac{\partial F}{\partial\delta\overline{y}'}\eta'\right)dx = 0.\end{aligned} \tag{2.2.32}$$

This equation is then evaluated at $\varepsilon = 0$ to obtain

$$\int_b^a \left(\frac{\partial F}{\partial\overline{y}}\eta + \frac{\partial F}{\partial\overline{y}'}\eta'\right)dx = 0\,. \tag{2.2.33}$$

where we notice ε is contained in (2.2.26). Integration by parts is now applied to the last term of the left hand side of (2.2.33)

$$\int_b^a \left(\frac{\partial F}{\partial \overline{y}} \eta \right) dx + \left[\frac{\partial F}{\partial \overline{y}'} \eta \right]_a^b - \int_a^b \left(\frac{d}{dx} \frac{\partial F}{\partial \overline{y}'} \right) \eta dx = 0 . \tag{2.2.34}$$

Since we know that $\eta(a) = \eta(b) = 0$, the term $\left[\frac{\partial F}{\partial \overline{y}'} \eta \right]_a^b$ is zero for both a and b, we can write (2.2.34) as

$$\int_b^a \left(\frac{\partial F}{\partial \overline{y}} - \frac{d}{dx} \frac{\partial F}{\partial \overline{y}'} \right) \eta dx = 0 . \tag{2.2.35}$$

For the final step, a lemma called the fundamental lemma of variational calculus is needed. It states that, if f is a function such that $f \in C^0[a,b]$ and

$$\int_a^b f(x)\, \eta(x)\, dx = 0 , \tag{2.2.36}$$

then for every function $\eta \in C^1[a,b]$, where $\eta(a) = \eta(b) = 0$, it must be that $f(x) = 0,\ \forall x \in [a,b]$. This has been proved [11]. From the lemma, we can finally conclude that

$$\frac{\partial F}{\partial \overline{y}} - \frac{d}{dx} \frac{\partial F}{\partial \overline{y}'} = 0 . \tag{2.2.37}$$

Equation (2.2.37) is called the Euler-Lagrange equation for functions of one variable [10] and with this in hand, it is possible to find a function that makes a given functional extreme. Using a similar approach, it can also be shown that the Euler-Lagrange equation for functions of two independent variables, x_1 and x_2, is

$$\frac{\partial F}{\partial \overline{y}} - \frac{d}{dx_1} \frac{\partial F}{\partial \overline{y}_{x_1}} - \frac{d}{dx_2} \frac{\partial F}{\partial \overline{y}_{x_2}} = 0 . \tag{2.2.38}$$

2.3.3 Deriving the elastic force

To derive the elastic force, the Euler-Lagrange equation for functions of two independent variables is used to find an expression that minimizes the energy, resulting from deformation of an elastically deformable body. The derivation of the elastic force follows [7].

Taking another look at (2.2.23), the expression will be split in two parts, S and T, which will deal with the first and the second fundamental form, respectively

$$S = \sum_{i,j=1}^{2} \eta_{ij} \left(G_{ij} - G_{ij}^0 \right)^2$$

and (2.2.39)

$$T = \sum_{i,j=1}^{2} \xi_{ij} \left(B_{ij} - B_{ij}^0 \right)^2 .$$

These terms will be developed individually with the calculus of variations. Writing out S, we have

$$\begin{aligned} S = \quad & \eta_{11}\left(G_{11} - G_{11}^0\right)^2 + \eta_{12}\left(G_{12} - G_{12}^0\right)^2 \\ & + \eta_{21}\left(G_{21} - G_{21}^0\right)^2 + \eta_{22}\left(G_{22} - G_{22}^0\right)^2 . \end{aligned} \tag{2.2.40}$$

From (2.2.13) we know that $\mathbf{G}$ is symmetrical, $G_{ij} = \mathbf{r}_{a_i} \cdot \mathbf{r}_{a_j}$, where $\mathbf{r}$ describes the surface $\mathbf{Q}$, and $\eta_{12} = \eta_{21}$. Applying this on S yields

$$\begin{aligned} S = \quad & \eta_{11}\left(\mathbf{r}_{a_1} \cdot \mathbf{r}_{a_1} - G_{11}^0\right)^2 \\ & + 2\eta_{12}\left(\mathbf{r}_{a_1} \cdot \mathbf{r}_{a_2} - G_{12}^0\right)^2 \\ & + \eta_{22}\left(\mathbf{r}_{a_2} \cdot \mathbf{r}_{a_2} - G_{22}^0\right)^2 . \end{aligned} \tag{2.2.41}$$

An expression that minimizes S over time is wanted, and from 2.3.2 Variational calculus we know that this involves applying variational calculus on the functional

$$I(\mathbf{r}) = \int_\Omega S\left(a_1, a_2, \mathbf{r}, \mathbf{r}_{a_1}, \mathbf{r}_{a_2}\right) da_1 da_2 . \tag{2.2.42}$$

The first variational derivative of S with respect to the variation of $\mathbf{r}$, is then defined as

$$\frac{\delta S}{\delta \mathbf{r}} = S_{\mathbf{r}} - \partial_{a_1} S_{\mathbf{r}_{a_1}} - \partial_{a_2} S_{\mathbf{r}_{a_2}} = 0 . \tag{2.2.43}$$

Looking at (2.2.41), it is clear that S does not depend directly on $\mathbf{r}$, this means that $S_{\mathbf{r}} = 0$ and vanish. The last two terms involves differentiating S with respect to $\mathbf{r}_{a_1}$ and $\mathbf{r}_{a_2}$. In the first case, we look at (2.2.41) and find that only the first two terms involve $\mathbf{r}_{a_1}$. It might seem odd that it is possible to differentiate with respect to a function, but in this situation it is possible to just consider it as a symbol. This means that

$$\frac{\partial}{\partial a_1}\left(\eta_{11}\left(\mathbf{r}_{a_1} \cdot \mathbf{r}_{a_1} - G_{11}^0\right)^2\right), \tag{2.2.44}$$

can simply be thought of as the symbolic differentiation

$$\frac{\partial}{\partial x}\left(k\left(x^2 - y\right)^2\right) = 4k\left(x^2 - y\right)x . \tag{2.2.45}$$

Performing the rest of the differentiation in S yields

$$\begin{aligned} \frac{\delta S}{\delta \mathbf{r}} = \quad & -\partial_{a_1}\left(4\eta_{11}\left(\mathbf{r}_{a_1} \cdot \mathbf{r}_{a_1} - G_{11}^0\right)\mathbf{r}_{a_1} + 4\eta_{12}\left(\mathbf{r}_{a_1} \cdot \mathbf{r}_{a_2} - G_{12}^0\right)\mathbf{r}_{a_2}\right) \\ & -\partial_{a_2}\left(4\eta_{12}\left(\mathbf{r}_{a_1} \cdot \mathbf{r}_{a_2} - G_{12}^0\right)\mathbf{r}_{a_1} + 4\eta_{22}\left(\mathbf{r}_{a_2} \cdot \mathbf{r}_{a_2} - G_{22}^0\right)\mathbf{r}_{a_2}\right). \end{aligned} \tag{2.2.46}$$

We name the terms $4\eta_{11}\left(\mathbf{r}_{a_1}\cdot\mathbf{r}_{a_1}-G^0_{11}\right)$ as α_{11}, $4\eta_{12}\left(\mathbf{r}_{a_1}\cdot\mathbf{r}_{a_2}-G^0_{12}\right)$ as α_{12}, and $4\eta_{22}\left(\mathbf{r}_{a_2}\cdot\mathbf{r}_{a_2}-G^0_{22}\right)$ as α_{22}, which means that (2.2.46) can be written as

$$\frac{\delta S}{\delta \mathbf{r}}=-\partial_{a_1}\left(\alpha_{11}\mathbf{r}_{a_1}+\alpha_{12}\mathbf{r}_{a_2}\right)-\partial_{a_2}\left(\alpha_{21}\mathbf{r}_{a_1}+\alpha_{22}\mathbf{r}_{a_2}\right), \tag{2.2.47}$$

where we note that $\alpha_{12}=\alpha_{21}$.

Writing out equation (2.2.47) using the product rule, results in

$$\begin{aligned}\frac{\delta S}{\delta \mathbf{r}}= &-\left(\partial_{a_1}\alpha_{11}\right)\mathbf{r}_{a_1}-\alpha_{11}\mathbf{r}_{a_1a_1}-\left(\partial_{a_1}\alpha_{12}\right)\mathbf{r}_{a_2}-\alpha_{12}\mathbf{r}_{a_1a_2}\\ &-\left(\partial_{a_2}\alpha_{21}\right)\mathbf{r}_{a_1}-\alpha_{21}\mathbf{r}_{a_2a_1}-\left(\partial_{a_2}\alpha_{22}\right)\mathbf{r}_{a_2}-\alpha_{22}\mathbf{r}_{a_2a_2},\end{aligned} \tag{2.2.48}$$

which can be compressed into short writing, using the Einstein summation convention

$$\frac{\delta S}{\delta \mathbf{r}}=-\sum_{i,j=1}^{2}\partial_{a_i}\alpha_{ij}\mathbf{r}_{a_j}. \tag{2.2.49}$$

This gives us an expression of the variation on the first fundamental form. The expression of the second fundamental form is arrived at in a similar manner. This time we use that $B_{ij}=\mathbf{n}\cdot\mathbf{r}_{a_ia_j}$, which we know from 2.2.5 Second fundamental form. Writing out T from (2.2.39) gives

$$\begin{aligned}T= &\ \xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2+\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2\\ &+\xi_{21}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_1}-B^0_{21}\right)^2+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2.\end{aligned} \tag{2.2.50}$$

Using that $\xi_{12}=\xi_{21}$ we arrive at

$$\begin{aligned}T= &\ \xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2\\ &+2\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2\\ &+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2.\end{aligned} \tag{2.2.51}$$

Once again we turn to the calculus of variations, to find an expression that will minimize T. We therefore look at the functional

$$I(\mathbf{r})=\int_\Omega T\left(a_1,a_2,\mathbf{r},\mathbf{r}_{a_1},\mathbf{r}_{a_2},\mathbf{r}_{a_1a_1},\mathbf{r}_{a_1a_2},\mathbf{r}_{a_2a_2}\right)da_1da_2, \tag{2.2.52}$$

and according to the theory, the first variational derivative is then

$$\begin{aligned}\frac{\delta T}{\delta \mathbf{r}}= &\ T_{\mathbf{r}}-\partial_{a_1}T_{\mathbf{r}_{a_1}}-\partial_{a_2}T_{\mathbf{r}_{a_2}}\\ &+\partial_{a_1a_1}T_{\mathbf{r}_{a_1a_1}}+2\partial_{a_1a_2}T_{\mathbf{r}_{a_1a_2}}+\partial_{a_2a_2}T_{\mathbf{r}_{a_2a_2}}.\end{aligned} \tag{2.2.53}$$

Observing in (2.2.51) that T is not dependent of $\mathbf{r}$, thus $T_{\mathbf{r}} = 0$ in (2.2.53). Writing out $\frac{\delta T}{\delta \mathbf{r}}$ yields

$$\begin{aligned}
\frac{\delta T}{\delta \mathbf{r}} = \quad & -\partial_{a_1}\partial_{\mathbf{r}_{a_1}}\left(\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2+2\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2\right)\\
& -\partial_{a_2}\partial_{\mathbf{r}_{a_2}}\left(\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2+2\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2\right)\\
& +\partial_{a_1a_1}\partial_{\mathbf{r}_{a_1a_1}}\left(\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2+2\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2\right) \qquad (2.2.54)\\
& +\partial_{a_1a_2}\partial_{\underline{\mathbf{r}}_{a_1a_2}}\left(\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2+2\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2\right)\\
& +\partial_{a_2a_2}\partial_{\underline{\mathbf{r}}_{a_2a_2}}\left(\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)^2+2\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)^2+\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)^2\right)
\end{aligned}$$

Performing the first part of the differentiation, involving derivatives of $\mathbf{r}$ yields

$$\begin{aligned}
\frac{\delta T}{\delta \mathbf{r}} = \quad & -\partial_{a_1}\left(2\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)\mathbf{n}^T_{\mathbf{r}_{a_1}}\mathbf{r}_{a_1a_1}+4\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)\mathbf{n}^T_{\mathbf{r}_{a_1}}\mathbf{r}_{a_1a_2}\right.\\
& \left.+2\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)\mathbf{n}^T_{\mathbf{r}_{a_1}}\mathbf{r}_{a_2a_2}\right)\\
& -\partial_{a_2}\left(2\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)\mathbf{n}^T_{\mathbf{r}_{a_2}}\mathbf{r}_{a_1a_1}+4\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)\mathbf{n}^T_{\mathbf{r}_{a_2}}\mathbf{r}_{a_1a_2}\right.\\
& \left.+2\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)\mathbf{n}^T_{\mathbf{r}_{a_2}}\mathbf{r}_{a_2a_2}\right) \qquad (2.2.55)\\
& +\partial_{a_1a_1}\left(2\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)\mathbf{n}\right)\\
& +2\partial_{a_1a_2}\left(4\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)\mathbf{n}\right)\\
& +\partial_{a_2a_2}\left(2\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)\mathbf{n}\right).
\end{aligned}$$

Just like for S, we will again introduce symbols to represent the difference operation involving the natural shape of the curvature $\mathbf{B}^0$. This means that $2\xi_{11}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_1}-B^0_{11}\right)$ is named β_{11}, $4\xi_{12}\left(\mathbf{n}\cdot\mathbf{r}_{a_1a_2}-B^0_{12}\right)$ is named β_{12}, and $2\xi_{22}\left(\mathbf{n}\cdot\mathbf{r}_{a_2a_2}-B^0_{22}\right)$ is named β_{22}. This shortens the equation to

$$\begin{aligned}
\frac{\delta T}{\delta \mathbf{r}} = \quad & -\partial_{a_1}\left(\beta_{11}\mathbf{n}^T_{\mathbf{r}_{a_1}}\mathbf{r}_{a_1a_1}+\beta_{12}\mathbf{n}^T_{\mathbf{r}_{a_1}}\mathbf{r}_{a_1a_2}+\beta_{22}\mathbf{n}^T_{\mathbf{r}_{a_1}}\mathbf{r}_{a_2a_2}\right)\\
& -\partial_{a_2}\left(\beta_{11}\mathbf{n}^T_{\mathbf{r}_{a_2}}\mathbf{r}_{a_1a_1}+\beta_{12}\mathbf{n}^T_{\mathbf{r}_{a_2}}\mathbf{r}_{a_1a_2}+\beta_{22}\mathbf{n}^T_{\mathbf{r}_{a_2}}\mathbf{r}_{a_2a_2}\right) \qquad (2.2.56)\\
& +\partial_{a_1a_1}\left(\beta_{11}\mathbf{n}\right)+2\partial_{a_1a_2}\left(\beta_{12}\mathbf{n}\right)+\partial_{a_2a_2}\left(\beta_{22}\mathbf{n}\right).
\end{aligned}$$

Cutting away the least dominant terms of (2.2.56) will result in an approximation. The most interesting things here are the second derivatives of the β terms, with regard to the curvature, so all first order derivatives is thrown out. Also $\mathbf{n}$ is approximated by $\mathbf{r}_{a_i a_j}$, which gives

$$\frac{\delta T}{\delta \mathbf{r}} = \partial_{a_1 a_1}\left(\beta_{11}\mathbf{r}_{a_1 a_1}\right) + 2\partial_{a_1 a_2}\left(\beta_{12}\mathbf{r}_{a_1 a_2}\right) + \partial_{a_2 a_2}\left(\beta_{22}\mathbf{r}_{a_2 a_2}\right). \tag{2.2.57}$$

Finally putting this into summation form yields

$$\frac{\delta T}{\delta \mathbf{r}} = \sum_{i,j=1}^{2} \partial_{a_i a_j}\left(\beta_{ij}\mathbf{r}_{a_i a_j}\right). \tag{2.2.58}$$

By merging S and T, we end up with

$$\frac{\delta e}{\delta \mathbf{r}} = \sum_{i,j=1}^{2} -\partial_{a_i}\left(\alpha_{ij}\mathbf{r}_{a_j}\right) + \partial_{a_i a_j}\left(\beta_{ij}\mathbf{r}_{a_i a_j}\right). \tag{2.2.59}$$

This equation gives a good way to handle the elastic forces on the particles of a deformable surface. It includes the first and second fundamental form, giving the object constraints on both stretching (length) and bending (curvature).

2.4 Finite Differences

In this section we will present a method that can be used to approximate derivatives, in the implementation of the elastically deformable models. The expression for the elastic force, 2.3 *Approximation of the Elasticity*, contains derivatives of the surface, which is one reason why we look at derivative approximations by finite differences. Calculation of surface normals also involves derivative approximation.

2.4.1 Functions of one variable

Generally the finite difference approximation that we are going to use, derives from the Taylor series expansion of a function about a point. This means that focusing at a point x in a function $F(x)$, and at the point on both sides of x, $F(x+h)$ and $F(x-h)$, with distance h from x. Taylor [5] tells us that these expansions are

$$F(x+h) = F(x) + hF'(x) + \frac{h^2}{2!}F''(x) + \ldots = \sum_{n=0}^{\infty}\frac{h^n}{n!}F^{(n)}(x) \tag{2.3.1}$$

and

$$F(x-h) = F(x) - hF'(x) + \frac{h^2}{2!}F''(x) + \ldots = \sum_{n=0}^{\infty}(-1)^n\frac{h^n}{n!}F^{(n)}(x). \tag{2.3.2}$$

These expansions are infinite series and when all terms are included, we know precisely how the function F behaves around x. Since computers do not yet have infinite computing powers, we need to leave out some terms of the calculation. Approximations appear when we leave out terms of the Taylor series. The terms that are left out become the error of the approximation. If h is sufficiently small, it turns out that just looking at the first two terms of the right hand side of (2.3.1) and (2.3.2), is a good approximation of the function. In this case we can find an approximation for the first derivative, either by forward difference or backward difference, depending on which equation we use.

Forward difference approximation

$$F'(x) = \frac{F(x+h) - F(x)}{h} + O(h). \tag{2.3.3}$$

Backward difference approximation

$$F'(x) = \frac{F(x) - F(x-h)}{h} + O(h). \tag{2.3.4}$$

The last term indicates that the error term is of first order. It is also possible to create what is known as a central difference approximation that looks at both sides of the point x simultaneously. This is done by subtracting (2.3.4) from (2.3.3).

Central difference approximation

$$F'(x) = \frac{F(x+h) - F(x-h)}{2h} + O\left(h^2\right). \tag{2.3.5}$$

Notice that this approximation has an error of second order. Approximations of higher order derivatives can be obtained in a similar manner, by including more terms from the Taylor series and also expanding the search distance h by integer increments. This means looking at points $F(x+ih)$ in the forward direction and $F(x-ih)$ in the backward direction. As an example let us look at a forward difference approximation of the second derivative, which can be found by looking at $2F(x+h)$ and $F(x+2h)$ where

$$F(x+2h) = F(x) + 2hF'(x) + \frac{(2h)^2}{2!}F''(x) + \ldots = \sum_{n=0}^{\infty} \frac{(2h)^n}{n!} F^{(n)}(x). \tag{2.3.6}$$

Using only the first three terms of $2F(x+h)$ and the first two of (2.3.6) and subtracting them from each other, we arrive at an approximation for the second derivative

$$\begin{gathered} F(x+2h) - 2F(x+h) = F(x) + 2hF'(x) - 2\left(F(x) + hF'(x) + \frac{h^2}{2!}F''(x)\right) + O(h) \\ \Updownarrow \\ F''(x) = \frac{F(x+2h) - 2F(x+h) + F(x)}{h^2} + O(h). \end{gathered} \tag{2.3.7}$$

It is similarly possible to increase the order of the error term, by using the same method of more terms from the Taylor series and wider search areas. One example is a forward difference approximation of the first derivative, with a second order error term

$$F'(x) = \frac{-F(x+2h) + 4F(x+h) - 3F(x)}{2h} + O\left(h^2\right). \tag{2.3.8}$$

2.4.1.1 The general approach

It is of great interest to find a general method that can specify a difference approximation from a given set of input parameters. This means that we can find any difference approximation we need by specifying parameters. One such method is specified in [5] and it takes, as input parameters,

the desired order of the derivative, d, and the order of the error term, p. The desired output is the coefficients C_i to the general difference approximation

$$\sum_{i=i_{\min}}^{i_{\max}} C_i F(x+ih), \tag{2.3.9}$$

for some choice of $i_{\min}$ and $i_{\max}$. The term $F(x+ih)$ can be written as

$$F(x+ih) = \sum_{n=0}^{\infty} i^n \frac{h^n}{n!} F^{(n)}(x). \tag{2.3.10}$$

Inserting (2.3.10) into (2.3.9) yields

$$\sum_{i=i_{\min}}^{i_{\max}} C_i \sum_{n=0}^{\infty} i^n \frac{h^n}{n!} F^{(n)}(x) = \sum_{n=0}^{\infty} \left(\sum_{i=i_{\min}}^{i_{\max}} i^n C_i \right) \frac{h^n}{n!} F^{(n)}(x). \tag{2.3.11}$$

Cutting off the number of terms in the series at $d+p$ results in

$$\sum_{n=0}^{d+p-1} \left(\sum_{i=i_{\min}}^{i_{\max}} i^n C_i \right) \frac{h^n}{n!} F^{(n)}(x) + O\left(h^{d+p}\right), \tag{2.3.12}$$

with an error term of order $d+p$. Multiplying by $\frac{d!}{h^d}$ gives the desired approximation

$$F^{(d)}(x) = \frac{d!}{h^d} \sum_{n=0}^{d+p-1} \left(\sum_{i=i_{\min}}^{i_{\max}} i^n C_i \right) \frac{h^n}{n!} F^{(n)}(x) + O\left(h^{p}\right), \tag{2.3.13}$$

with an error term of order p. For (2.3.13) to hold, the following must be satisfied

$$\sum_{i=i_{\min}}^{i_{\max}} i^n C_i = \begin{cases} 0, & 0 \le n \le d+p-1 \wedge n \neq d \\ 1, & n = d \end{cases}. \tag{2.3.14}$$

This is a system of $d+p$ linear equations with $i_{\max} - i_{\min} + 1$ unknowns. The system has a unique solution, if the number of unknowns is constrained to $d+p$. The type of difference approximation is controlled by the $[i_{\min}..i_{\max}]$ interval. Setting $i_{\min} = 0$ and $i_{\max} = d+p-1$ produces a forward difference approximation, while a backward difference approximation is produced by setting $i_{\min} = -(d+p-1)$ and $i_{\max} = 0$. The central difference approximation arises when $-i_{\min} = i_{\max} = \frac{d+p-1}{2}$, and it requires that $d+p$ is an odd number. As an example, let us look at the forward approximation of the second derivative that was presented in (2.3.7). Here we set $d=2$ and $p=1$, this means that $i_{\min} = 0$ and $i_{\max} = 2$. The linear system of equations in (2.3.14) becomes

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.3.15}$$

This system has the solution $(C_0, C_1, C_2) = (1, -2, 1)/2$, which can be verified to be the right coefficients for the approximation (2.3.7).

2.4.2 Functions of two variables

Since we are working with deformable surfaces, we need to work with derivatives of functions of two variables. This means that we are interested in generalizing the previous section to functions of two variables. This is a fairly simple task, which utilizes that the above results can be applied to each of the two variables of the function in turn. If the function is $F(x,y)$ and h is the search distance in the x-direction, and k is the search distance in the y-direction. The forward difference approximations of the first derivatives are

$$\begin{aligned} f_x(x,y) &\approx \frac{F(x+h,y) - F(x,y)}{h} \\ f_y(x,y) &\approx \frac{F(x,y+k) - F(x,y)}{k}, \end{aligned} \tag{2.3.16}$$

the backward difference approximations are

$$\begin{aligned} f_x(x,y) &\approx \frac{F(x,y) - F(x-h,y)}{h} \\ f_y(x,y) &\approx \frac{F(x,y) - F(x,y-k)}{k}, \end{aligned} \tag{2.3.17}$$

and the central difference approximations are

$$\begin{aligned} f_x(x,y) &\approx \frac{F(x+h,y) - F(x-h,y)}{2h} \\ f_y(x,y) &\approx \frac{F(x,y+k) - F(x,y-k)}{2k}. \end{aligned} \tag{2.3.18}$$

Function names in lowercase are the actual approximations with the error terms removed. Finding second derivatives requires a bit more, since this also includes finding mixed derivatives. Second derivatives are found by differentiating twice, which is why we can actually use the above results, by combining them into the desired second derivatives. For example it is possible to find a central difference approximation to the second derivative, by combining the forwards and backwards difference formula of the first derivative, (2.3.16) and (2.3.17). This is done by applying one of the approximations, and then applying the other to the result. Below we have determined the second derivative in the x-direction, using first the backwards approximation and then applying the forward approximation to the result

$$\begin{aligned} f_{xx}(x,y) &\approx \frac{f_x^+(x,y) - f_x^+(x-h,y)}{h} \\ &\approx \frac{\frac{F(x+h,y)-F(x,y)}{h} - \frac{F(x,y)-F(x-h,y)}{h}}{h} \\ &= \frac{F(x+h,y) - 2F(x,y) + F(x-h,y)}{h^2}, \end{aligned} \tag{2.3.19}$$

where the superscript $+$ denotes a forward difference operator, and the superscript $-$ denotes a backward difference operator.
The mixed derivatives can be found in the same way. This time we combine the two forward approximations of the first derivative, in each of the two directions, x and y. We apply the approximation in the x-direction, followed by applying the approximation in the y-direction

$$\begin{aligned} f_{xy}(x,y) &\approx \frac{f_y^+(x+h,y) - f_y^+(x,y)}{h} \\ &\approx \frac{\frac{F(x+h,y+k)-F(x+h,y)}{h} - \frac{F(x,y+k)-F(x,y)}{h}}{k} \\ &= \frac{F(x+h,y+k) - F(x+h,y) - F(x,y+k) + F(x,y)}{hk}. \end{aligned} \tag{2.3.20}$$

In this way it is possible to combine the results into any required approximation. The procedure is first to find the one dimensional approximations, and then to combine these into higher order approximations for the functions of two variables or more, by applying each approximation in turn.

3 Execution and Practice

We have presented the main theory needed for understanding elastically deformable models. We will now continue the development of how to actually execute the theory, and implement all the different parts from the article [21].

As we have been through all the areas of the article during the study of elastic deformation, we will through the forthcoming sections discuss the most difficult parts. The general idea is that since we have been struggling with those difficulties, it is likely that the reader might find the same areas troublesome. In this chapter we will present all our own contributions to erroneous parts from the article and a better understanding of the tricky subjects.

3.1 Discretization

In the discretization part we move from the continuous domain, where the elastically deformable model was originally specified, and into the discrete domain. This will finally enable us to implement an approximation of the model (2.1.1) on a computer.

The physical model specified, is based on objects consisting of particles. In the real world, objects are composed of a very large number of particles (atoms, molecules), but for computational reasons, we will be looking at a much smaller number of particles. This is also due to the fact that the implementation will be aiming at a real-time visualization of the evolving model, as opposed to an implementation rendering the progress to a file. The implementation will be scalable with regards to the number of particles, but increasing this number radically, will only result in lower frame rates.

Focusing at a two dimensional deformable surface, this will, in the discrete case, be considered as a 2D lattice of particles. The numbering of the individual particles are handled by two indices, m and n, representing each of the two dimensions on the surface. Looking at Figure 3.1 we will see an example of a surface with 4x4 particles. If we look at the positions of the particles in a surface, these are three dimensional vectors, and can be collectively thought of as a vector field. For the surface this position field is named $\mathbf{r}[m,n]$. This gives us a way to describe the position of a surface and its particles. The next thing that should be done is to apply the elastically deformable model to the particles, and let them evolve over time.

An important step of the discretization of the model is the way to transform the elastic force into something that is easily calculated. This will be done using finite differencing, which was introduced in 2.4 Finite Differences. The finite differences will work on the positional vector field of the surface, to create approximations to the derivatives in each of the two directions, m and n. This is exactly what we need to discretize the elastic force on the particles, since this force consist solely of directional derivatives of the positions. To ease up the notation, each of the two directions is given a number, thus m will be 1 and n will be 2. For example, the first derivative in the first direction using forward differencing will be

$$D_1^+(\mathbf{r})[m,n] = \frac{\mathbf{r}[m+1,n] - \mathbf{r}[m,n]}{h_1}, \tag{3.1.1}$$

where $h_1 = \frac{1}{M-1}$ represents the uniform spacing between particles in the m-direction, and M represents the amount of particles along that direction. Similarly $h_2 = \frac{1}{N-1}$ is the spacing in the n-direction, where N is the amount of particles in that direction.

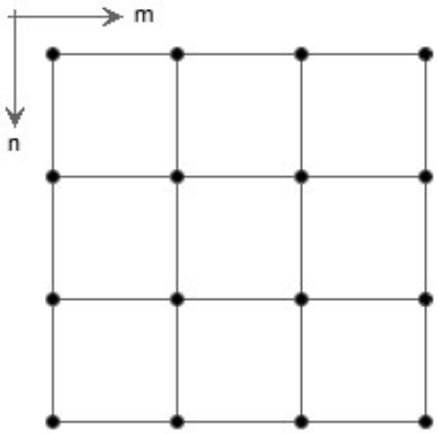


Figure 3.1 A 4×4 lattice of particles

Considering the expression that was derived for the elastic force, 2.3 *Approximation of the Elasticity*

$$\mathbf{e} = \sum_{i,j=1}^{2} -\partial_{a_i} \alpha_{ij} \mathbf{r}_{a_j} + \partial^2_{a_i a_j} \beta_{ij} \mathbf{r}_{a_i a_j}, \tag{3.1.2}$$

with

$$\alpha_{ij} = \eta_{ij}(\mathbf{r}_{a_i} \cdot \mathbf{r}_{a_j} - \mathbf{G}^0_{ij})$$

and (3.1.3)

$$\beta_{ij} = \xi_{ij}(\mathbf{n} \cdot \mathbf{r}_{a_i a_j} - \mathbf{B}^0_{ij}).$$

The process of discretizing these equations will simply be to replace the directional derivatives with appropriate elements from finite differencing. The superscripts $(+)$ and $(-)$, indicate the forward and backward cross difference operators are used when $i \neq j$, which will be defined later. The elastic force $\mathbf{e}$ becomes

$$\mathbf{e}[m,n] = \sum_{i,j=1}^{2} -D_i^-(\mathbf{p})[m,n] + D_{ij}^{(-)}(\mathbf{q})[m,n], \tag{3.1.4}$$

where the backwards difference operators work on the vector fields $\mathbf{p}$ and $\mathbf{q}$ defined by

$$\mathbf{p}[m,n] = \alpha_{ij}[m,n] D_j^+(\mathbf{r})[m,n]$$

and (3.1.5)

$$\mathbf{q}[m,n] = \beta_{ij}[m,n] D_{ij}^{(+)}(\mathbf{r})[m,n],$$

and the discretized versions of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are

$$\alpha_{ij}[m,n] = \eta_{ij}[m,n](D_i^+(\mathbf{r})[m,n] \cdot D_j^+(\mathbf{r})[m,n] - G_{ij}^0[m,n])$$

and (3.1.6)

$$\beta_{ij}[m,n] = \xi_{ij}[m,n](\mathbf{n}[m,n] \cdot D_{ij}^{(+)}(\mathbf{r})[m,n] - B_{ij}^0[m,n]).$$

This expression for the elastic force are now ready to be used, either in a direct computation of the elasticity, for each particle in the explicit method, or in the further developments of the matrix calculations, of the total system in the semi-implicit method. The elasticity is the third term in the continuous model (2.1.1). The first two terms can be written as

$$\mathbf{M}\frac{\partial^2 \underline{\mathbf{r}}}{\partial t^2} + \mathbf{C}\frac{\partial \underline{\mathbf{r}}}{\partial t}, \tag{3.1.7}$$

where

$$\mathbf{M} \equiv \begin{bmatrix} \mu[1,1] & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \mu[2,1] & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \mu[M,1] & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \mu[1,2] & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \ldots & \mu[M,N] \end{bmatrix} \tag{3.1.8}$$

is the diagonal $MN \times MN$ mass matrix, assembled from $\mu[m,n]$, and

$$\mathbf{C} \equiv \begin{bmatrix} \gamma[1,1] & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \gamma[2,1] & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \gamma[M,1] & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \gamma[1,2] & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \ldots & \mu[M,N] \end{bmatrix} \tag{3.1.9}$$

is the diagonal $MN \times MN$ damping matrix, assembled from $\gamma[m,n]$. The important thing here is that the time derivatives, which will be divided into discrete time steps, are integrated through time. At some time t, $t + \Delta t$ is the next time step and $t - \Delta t$ is the previous. For the derivatives, second-order accurate discrete time approximations are used, such that

$$\begin{aligned} \frac{\partial^2 \underline{\mathbf{r}}}{\partial t^2} &\approx \frac{\underline{\mathbf{r}}_{t+\Delta t} - 2\underline{\mathbf{r}}_t + \underline{\mathbf{r}}_{t-\Delta t}}{\Delta t^2} \\ \frac{\partial \underline{\mathbf{r}}}{\partial t} &\approx \frac{\underline{\mathbf{r}}_{t+\Delta t} - \underline{\mathbf{r}}_{t-\Delta t}}{2\Delta t}. \end{aligned} \tag{3.1.10}$$

These approximations to the time derivatives will also be included in subsequent sections, when solutions for the explicit and semi-implicit methods are to be found.

3.1.1 Difference operators

An explanation of the different types of difference operators can be found in 2.4 *Finite Differences*. In this section, we will derive the difference operators proposed by Terzopoulos et al. [21]. We will be using a generic grid function $\mathbf{u}[m,n]$ that can be replaced by any other grid function, e.g. the position field $\mathbf{r}[m,n]$.

The forward first-order difference operators are given by

$$\begin{aligned} D_1^+(\mathbf{u})[m,n] &= \frac{\mathbf{u}[m+1,n]-\mathbf{u}[m,n]}{h_1} \\ D_2^+(\mathbf{u})[m,n] &= \frac{\mathbf{u}[m,n+1]-\mathbf{u}[m,n]}{h_2}, \end{aligned} \tag{3.1.11}$$

the backward first-order difference operators

$$\begin{aligned} D_1^-(\mathbf{u})[m,n] &= \frac{\mathbf{u}[m,n]-\mathbf{u}[m-1,n]}{h_1} \\ D_2^-(\mathbf{u})[m,n] &= \frac{\mathbf{u}[m,n]-\mathbf{u}[m,n-1]}{h_2}, \end{aligned} \tag{3.1.12}$$

and the central first-order difference operators

$$\begin{aligned} D_1(\mathbf{u})[m,n] &= \frac{\mathbf{u}[m+1,n]-\mathbf{u}[m-1,n]}{2h_1} \\ D_2(\mathbf{u})[m,n] &= \frac{\mathbf{u}[m,n+1]-\mathbf{u}[m,n-1]}{2h_2}. \end{aligned} \tag{3.1.13}$$

Using (3.1.11) the forward second-order cross difference operators are given by

$$\begin{aligned} D_{12}^+(\mathbf{u})[m,n] &= D_1^+\left(D_2^+(\mathbf{u})\right)[m,n] = \\ &\frac{D_2^+(\mathbf{u})[m+1,n]-D_2^+(\mathbf{u})[m,n]}{h_1} = \\ &\frac{\frac{\mathbf{u}[m+1,n+1]-\mathbf{u}[m+1,n]}{h_2}-\frac{\mathbf{u}[m,n+1]-\mathbf{u}[m,n]}{h_2}}{h_1} = \\ &\frac{\mathbf{u}[m,n]-\mathbf{u}[m+1,n]-\mathbf{u}[m,n+1]+\mathbf{u}[m+1,n+1]}{h_1h_2} \end{aligned} \tag{3.1.14}$$

$$D_{21}^+(\mathbf{u})[m,n] = D_{12}^+(\mathbf{u})[m,n],$$

and using (3.1.12) the backward second-order cross difference operators are given by

$$
\begin{aligned}
D^-_{12}(\mathbf{u})[m,n] = D^-_1\left(D^-_2(\mathbf{u})\right)[m,n] = \\
\frac{D^-_2(\mathbf{u})[m,n] - D^-_2(\mathbf{u})[m-1,n]}{h_1} = \\
\frac{\dfrac{\mathbf{u}[m,n]-\mathbf{u}[m,n-1]}{h_2} - \dfrac{\mathbf{u}[m-1,n]-\mathbf{u}[m-1,n-1]}{h_2}}{h_1} = \\
\frac{\mathbf{u}[m,n]-\mathbf{u}[m-1,n]-\mathbf{u}[m,n-1]+\mathbf{u}[m-1,n-1]}{h_1 h_2}
\end{aligned}
\tag{3.1.15}
$$

$$
D^-_{21}(\mathbf{u})[m,n] = D^-_{12}(\mathbf{u})[m,n].
$$

The central second-order cross difference operators are equally obtained using both (3.1.11) and (3.1.12)

$$
\begin{aligned}
D_{11}(\mathbf{u})[m,n] = D^-_1\left(D^+_1(\mathbf{u})\right)[m,n] = \\
\frac{D^+_1(\mathbf{u})[m,n] - D^+_1(\mathbf{u})[m-1,n]}{h_1} = \\
\frac{\dfrac{\mathbf{u}[m+1,n]-\mathbf{u}[m,n]}{h_1} - \dfrac{\mathbf{u}[m,n]-\mathbf{u}[m-1,n]}{h_1}}{h_1} = \\
\frac{\mathbf{u}[m+1,n]-2\mathbf{u}[m,n]+\mathbf{u}[m-1,n]}{h_1^2}
\end{aligned}
\tag{3.1.16}
$$

$$
\begin{aligned}
D_{22}(\mathbf{u})[m,n] = D^-_2\left(D^+_2(\mathbf{u})\right)[m,n] = \\
\frac{D^+_2(\mathbf{u})[m,n] - D^+_2(\mathbf{u})[m,n-1]}{h_2} = \\
\frac{\dfrac{\mathbf{u}[m,n+1]-\mathbf{u}[m,n]}{h_2} - \dfrac{\mathbf{u}[m,n]-\mathbf{u}[m,n-1]}{h_2}}{h_2} = \\
\frac{\mathbf{u}[m,n+1]-2\mathbf{u}[m,n]+\mathbf{u}[m,n-1]}{h_2^2}.
\end{aligned}
$$

We have now computed all the difference operators needed to write out the whole discretized version of the elasticity for surfaces (3.1.4). The trick is first to calculate $\boldsymbol{\alpha}[m,n]$ and $\boldsymbol{\beta}[m,n]$ for all MN particles, thus creating the tension and rigidity tensor fields. Writing out the elasticity $\mathbf{e}[m,n]$ is a matter of using the same techniques, as we have applied to derive the second-order difference operators.

3.2 Elasticity

In this section we will take a closer look at the discrete equation for the elasticity presented in 3.1 *Discretization*. The equation is

$$\mathbf{e}[m,n] = \sum_{i,j=1}^{2} -D_i^-(\mathbf{p})[m,n] + D_{ij}^{(-)}(\mathbf{q})[m,n] \tag{3.2.1}$$

where

$$\mathbf{p}[m,n] = \alpha_{ij}[m,n] D_j^+(\mathbf{r})[m,n]$$

and

$$\mathbf{q}[m,n] = \beta_{ij}[m,n] D_{ij}^{(+)}(\mathbf{r})[m,n], \tag{3.2.2}$$

with the constitutive functions α_{ij} and β_{ij} defined as

$$\alpha_{ij}[m,n] = \eta_{ij}[m,n](D_i^+(\mathbf{r})[m,n] \cdot D_j^+(\mathbf{r})[m,n] - G_{ij}^0[m,n]) \tag{3.2.3}$$

and

$$\beta_{ij}[m,n] = \xi_{ij}[m,n](\mathbf{n}[m,n] \cdot D_{ij}^{(+)}(\mathbf{r})[m,n] - B_{ij}^0[m,n])\,. \tag{3.2.4}$$

As can be seen in (3.2.2), $\mathbf{p}$ and $\mathbf{q}$ have been separated from (3.2.1) into there own equations. The reason for this is to put emphasis on the fact that they represent fields of the same size as the discretized surface $\mathbf{r}[m,n]$. $\mathbf{p}$ and $\mathbf{q}$, are in fact tensor fields and the indices i and j indicates that four values of $\mathbf{p}$ and $\mathbf{q}$ exists for each particle. The important property is that they are fields that can be inserted into the finite difference operators. In fact, every symbol in (3.2.1) through (3.2.4) is postfixed with $[m,n]$ meaning that this symbol has a value for every particle on the surface. Even symbols such as $D_j^+(\mathbf{r})$ has a value for every particle. This is actually the key to understanding how it works. The particle that is to have its elasticity, $\mathbf{e}$, calculated, has its position at $\mathbf{r}[m,n]$, meaning that it is the position of the particle at the center of attention. Now evaluating the elasticity is simply a matter of doing a number of neighborhood operations through the finite difference operators, e.g. the particle position $\mathbf{r}[m+1,n]$ is the position just next to the particle at hand, and similarly $D_2^+(\mathbf{r})[m+1,n]$ is the value of the forward difference operator in the second direction (the n-direction) on $\mathbf{r}$, for that same particle.

The elastic force in (3.2.1) consists of two terms. The first term involves the first fundamental form of the surface, and the second term involves the second fundamental form, see 2.2 *Differential Geometry* for the details. Dealing with the terms separately, will make things a little easier to overcome. Both terms are handled in turn throughout this section.

3.2.1 Intrinsic elastic force

The elastic force (3.2.1), describes the elasticity inherent in a particle on a deformable surface, through the size of the elasticity vector $\mathbf{e}$. The elasticity can be thought of as an internal property of the particle, and it increases as the surface deforms at the intrinsic material position. The elastic energy is internal, as opposed to external energy that comes into the system in the form of external forces, such as gravitational pull acting on the particles. The elasticity, $\mathbf{e}$, is the elastic force that should be working on the particle, in order to minimize the deformation.

Taking a closer look at (3.2.3) and (3.2.4) reveals the underlying model that governs the elastic motion. It is seen here that these equations are in fact spring models that closely resembles Hooke's law, which states that

$$\mathbf{F} = -k\mathbf{x}\,, \tag{3.2.5}$$

where $\mathbf{x}$ is a displacement vector from the resting state of the spring and k is called the spring constant. This constant k describes the behavior of the spring. The $\boldsymbol{\eta}$ and $\boldsymbol{\xi}$ tensors both work as spring constants in (3.2.3) and (3.2.4) but without the sign from (3.2.5). In (3.2.3) and (3.2.4) the $\mathbf{x}$ value of (3.2.5) also exists in the form of the difference operation on the states of the fundamental forms. In (3.2.3), the expression inside the parenthesis is simply put

$$\mathbf{G} - \mathbf{G}^0\,, \tag{3.2.6}$$

and represents a difference between the current state of the coefficients of the first fundamental form, and the natural resting state of the first fundamental form. Likewise in (3.2.4) we have the value inside the parenthesis equal to

$$\mathbf{B} - \mathbf{B}^0\,, \tag{3.2.7}$$

but the only change here, is that this is a difference between the states of the second fundamental form. What this shows, is that the behavior of the elasticity is actually a spring model, with the fundamental forms as the *units of measurement*.

3.2.2 Tension

The first term of (3.2.1) can be called the tension constraint on the elasticity. This term involves the first fundamental form of the surface through (3.2.3). In 2.2 *Differential Geometry* we have seen that this form looks at arc lengths on a surface. The first term of (3.2.1) makes sure that the absolute value of the energy increases when the surface is stretched or compressed, which means that the arc length of the surface has changed. Positive values of (3.2.3) indicate that the surface has stretched, while negative values indicate that it has been compressed.
Looking closer at the $\mathbf{G}$ tensor

$$\mathbf{G} = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{r}}{\partial m} \cdot \frac{\partial \mathbf{r}}{\partial m} & \frac{\partial \mathbf{r}}{\partial m} \cdot \frac{\partial \mathbf{r}}{\partial n} \\ \frac{\partial \mathbf{r}}{\partial n} \cdot \frac{\partial \mathbf{r}}{\partial m} & \frac{\partial \mathbf{r}}{\partial n} \cdot \frac{\partial \mathbf{r}}{\partial n} \end{bmatrix}, \tag{3.2.8}$$

for the surface $\mathbf{r}$, and the coordinate directions m and n, we see the significance of the different elements. G_{11} and G_{22} is the squared vector norm of the derivative vectors in each of the coordinate directions. These entries look at the length in each direction, and in comparison with the resting state $\mathbf{G}^0$, determine stretching or compressing of the material. The entries $G_{12} = G_{21}$ determine an indication of the angle between the two coordinate directions, and together with $\mathbf{G}^0$ presents an angular constraint.
The constant $\boldsymbol{\eta}$ contains the weights of the metric tensor, and looks like

$$\boldsymbol{\eta} = \begin{bmatrix} \eta_{11} & \eta_{12} \\ \eta_{21} & \eta_{22} \end{bmatrix}, \tag{3.2.9}$$

where $\eta_{ij} \geq 0$. The entries control the corresponding elements in $\mathbf{G}$. η_{11} and η_{22} describe the surface resistance to length deformations in the two directions, while $\eta_{12} = \eta_{21}$ describes resistance to shearing. Simulating surfaces that are very stretchable, such as soft rubber, is done by

setting $\boldsymbol{\eta}$ small, whereas less stretchable surfaces, such as paper, will have a larger $\boldsymbol{\eta}$. If $\eta_{ij} > 1$ the system becomes over constrained, and tends to become unstable.

3.2.2.1 Writing out the tension term

We will briefly give an example of how to write out the tension term. Expanding the tension term from (3.2.1), instead of using the summation symbol, and inserting $\mathbf{p}$ from (3.2.2), will give an overview of the situation

$$\begin{aligned}
&-D_1^-\left(\alpha_{11}[m,n]D_1^+(\mathbf{r})[m,n]\right) \quad for \quad i=1,\ j=1\\
&-D_1^-\left(\alpha_{12}[m,n]D_2^+(\mathbf{r})[m,n]\right) \quad for \quad i=1,\ j=2\\
&-D_2^-\left(\alpha_{21}[m,n]D_1^+(\mathbf{r})[m,n]\right) \quad for \quad i=2,\ j=1\\
&-D_2^-\left(\alpha_{22}[m,n]D_2^+(\mathbf{r})[m,n]\right) \quad for \quad i=2,\ j=2.
\end{aligned} \tag{3.2.10}$$

To make it clear what happens next, we will focus on the first line of (3.2.10), i.e. for $i=j=1$. We know from 3.1 *Discretization* that the backward difference operator is

$$D_1^-(\mathbf{u})[m,n] = \frac{\mathbf{u}[m,n]-\mathbf{u}[m-1,n]}{h_1}. \tag{3.2.11}$$

Using $\alpha_{11}[m,n]D_1^+(\mathbf{r})[m,n]$ as the grid function $\mathbf{u}$ yields

$$\begin{aligned}
&-\left(\frac{\alpha_{11}[m,n]D_1^+(\mathbf{r})[m,n]-\alpha_{11}[m-1,n]D_1^+(\mathbf{r})[m-1,n]}{h_1}\right)=\\
&\frac{-\alpha_{11}[m,n]D_1^+(\mathbf{r})[m,n]+\alpha_{11}[m-1,n]D_1^+(\mathbf{r})[m-1,n]}{h_1}
\end{aligned} \tag{3.2.12}$$

and further extending the forward difference operators gives

$$\frac{-\alpha_{11}[m,n]\big(\mathbf{r}[m+1,n]-\mathbf{r}[m,n]\big)+\alpha_{11}[m-1,n]\big(\mathbf{r}[m,n]-\mathbf{r}[m-1,n]\big)}{{h_1}^2}. \tag{3.2.13}$$

Notice that $D_1^+(\mathbf{r})[m-1,n]$ is simply a displacement by one in the negative m-direction of $D_1^+(\mathbf{r})[m,n]$.

3.2.3 Curvature

The curvature constraint of the elasticity is the second term of (3.2.1). This term involves the second fundamental form of the surface through (3.2.4). From 2.2 *Differential Geometry* we know that this form concerns itself with the curvature of a surface. Equation (3.2.4) is the measurement of the curvature constraint. In effect, it tells how much the surface deviates from its resting state with regards to bending and twisting. When β_{ij} grows in the positive direction, it means that the surface is more curved than its natural shape $\mathbf{B}^0$, and that it wants to flatten. In [21] the opposite case is also stated. Here it is mentioned that when β_{ij} grows in the negative direction, the surface wants to become more curved, which means that the natural shape of the surface is a curved one. However this case does not seem to be as simple as that. In our efforts to implement the model,

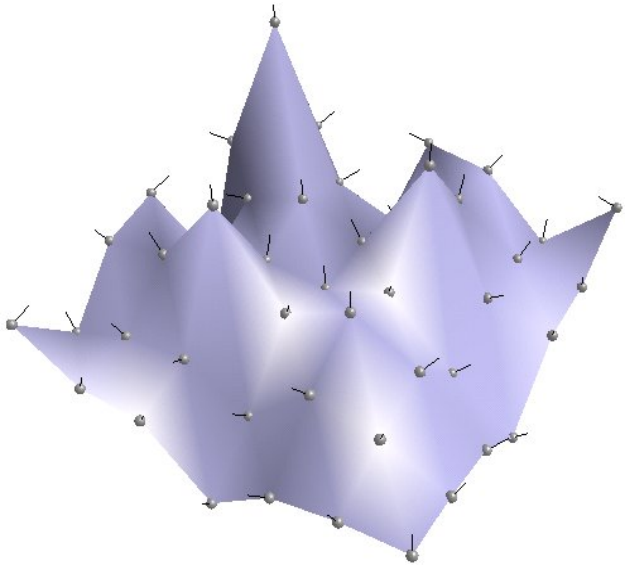


Figure 3.2 Unwanted features in the surface with negative curvature

we have found no indication that it is possible to have a natural shape $\mathbf{B}^0$ that is curved. The model seems to work perfectly when the natural shape is zero, which means that the surface is completely flat. When the natural shape is curved, the model becomes extremely unstable to the point where it explodes. Right before this happens the effects of the negative curvature can be observed. Unwanted features are introduced in the surface in wave-like patterns, as depicted in Figure 3.2. The features also appear to be completely symmetrical in nature. Why does this happen? The explanation does not seem to be straight forward, but one way to look at it, is in terms of energy introduction. When the natural shape of the deformable surface is flat, the system is always trying to remove energy until there is nothing left. No energy is left in the system when it reaches equilibrium, which means that no instabilities, with regards to the curvature, can occur. On the other hand, if the natural shape is a curved one, then energy is introduced into the system in order for it to reach this state. This turns out not to be a stable situation, when trying to put the system to rest at a point, where energy is still in the system. As of writing, the conclusion to this problem is that the implementation is not able to handle this curiosity in a satisfactory manner. What we have done instead is to modify (3.2.4) in order to only use the positive values, even though β_{ij} becomes negative. The modified function that is able to handle the curvature to flatten all surfaces is

$$\beta_{ij}[m,n] = \left|\xi_{ij}[m,n](\mathbf{n}[m,n] \cdot D_{ij}^{(+)}(\mathbf{r})[m,n] - B_{ij}^{0}[m,n])\right|, \tag{3.2.14}$$

that is, only to use the absolute values.

The coefficients of the second fundamental form is gathered in the tensor $\mathbf{B}$ and they are

$$\mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{n} \cdot \dfrac{\partial^2 \mathbf{r}}{\partial m^2} & \mathbf{n} \cdot \dfrac{\partial^2 \mathbf{r}}{\partial m \partial n} \\ \mathbf{n} \cdot \dfrac{\partial^2 \mathbf{r}}{\partial n \partial m} & \mathbf{n} \cdot \dfrac{\partial^2 \mathbf{r}}{\partial n^2} \end{bmatrix}, \tag{3.2.15}$$

where the coefficients B_{11} and B_{22}, measure bending in each of the two coordinate directions, while $B_{12} = B_{21}$ look at twisting of the surface. Again, by looking at the comparison between $\mathbf{B}$ and $\mathbf{B}^0$ in (3.2.4), we see that this is a measure of the deformation with regards to bending and twisting of the surface.
In (3.2.4) or (3.2.14), $\boldsymbol{\xi}$ is the constant rigidity tensor of the surface, and it looks like

$$\boldsymbol{\xi} = \begin{bmatrix} \xi_{11} & \xi_{12} \\ \xi_{21} & \xi_{22} \end{bmatrix}, \tag{3.2.16}$$

where parameters ξ_{11} and ξ_{22}, describe the resistance to bending in each of the coordinate directions, while $\xi_{12} = \xi_{21}$ put a weight on the resistance to twisting. To simulate surfaces with high resistance to bending, such as thin metal plates, $\boldsymbol{\xi}$ should generally be large, though still $\boldsymbol{\xi} < \mathbf{1}$. On the other hand, to simulate surfaces with flexible bending properties, such as cloth, $\boldsymbol{\xi}$ should be close to zero.

3.2.3.1 Writing out the curvature term

We will give a short example of how to write out the curvature term. Starting out, we insert $\mathbf{q}$ from (3.2.2) into (3.2.1) and look at the second term in isolation. Let us take a closer look at the situation for $i = j = 1$. In this situation we have

$$D_{11}\left(\beta_{11}[m,n]\, D_{11}(\mathbf{r})[m,n]\right). \tag{3.2.17}$$

Referring to 3.1.1 *Difference operators* we find that

$$D_{11}(\mathbf{u})[m,n] = \frac{\mathbf{u}[m+1,n] - 2\mathbf{u}[m,n] + \mathbf{u}[m-1,n]}{h_1^2}, \tag{3.2.18}$$

thus using $\beta_{11}[m,n]\, D_{11}(\mathbf{r})[m,n]$ as $\mathbf{u}$ yields

$$\frac{\beta_{11}[m+1,n]\, D_{11}(\mathbf{r})[m+1,n] - 2\beta_{11}[m,n]\, D_{11}(\mathbf{r})[m,n] + \beta_{11}[m-1,n]\, D_{11}(\mathbf{r})[m-1,n]}{h_1^2}. \tag{3.2.19}$$

Expanding the rest of the difference operators we have

$$\begin{aligned} &\frac{\beta_{11}[m+1,n]\big(\mathbf{r}[m+2,n] - 2\mathbf{r}[m+1,n] + \mathbf{r}[m,n]\big)}{h_1^4} \\ &- \frac{2\beta_{11}[m,n]\big(\mathbf{r}[m+1,n] - 2\mathbf{r}[m,n] + \mathbf{r}[m-1,n]\big)}{h_1^4} \\ &+ \frac{\beta_{11}[m-1,n]\big(\mathbf{r}[m,n] - 2\mathbf{r}[m-1,n] + \mathbf{r}[m-2,n]\big)}{h_1^4}. \end{aligned} \tag{3.2.20}$$

The same procedure is applied for the three other values of i, j.

3.2.4 Alternate method

The way we have been writing out (3.2.1) up until now, is actually not the only way to think of this process. Considering (3.2.1), we could easily think that the product rule of differentiation was

needed. This is because $\mathbf{p}$ and $\mathbf{q}$ from (3.2.2) are actually products of two functions, one of them being α_{ij} or β_{ij}, and the other being some differentiation of the surface $\mathbf{r}$. The product rule states that

$$\frac{d}{dx}(fg) = f\frac{dg}{dx} + g\frac{df}{dx}. \tag{3.2.21}$$

One of the problems with this procedure, is that it produces a great deal more terms in writing out (3.2.1). The biggest problem though, seems to be the increased sizes of the stencils that describe the neighborhood operations of the tension and curvature terms. For the tension term, this means that second order derivatives are involved, and for the curvature term, fourth order derivatives are involved. It not only increases the amount of calculations that need to be done, but also introduces new problems at the boundaries of the surface. We have tried to use the product rule in writing out the elasticity equation, but must simply conclude that more problems arise, and it is basically impossible to end up with a stable system. We have included this section to inform other interested programmers that would want to try using the product rule that it is simply not worth the effort. However, our experiments are still available. Please consult 4.2.2.3 *Stiffness matrix method* for details about using other methods to assemble the stiffness matrix.

3.3 The Stiffness Matrix

Considering the description of the semi-implicit method to evolve the deformable model we find that the elasticity $\mathbf{e}$, derived at 2.3.3 *Deriving the elastic force*, no longer can be calculated explicitly by the derived equation. Instead the positions are extracted from the elasticity term by the expression

$$\underline{\mathbf{e}} = \mathbf{K}(\underline{\mathbf{r}})\underline{\mathbf{r}}. \tag{3.3.1}$$

With the semi-implicit method we will solve for all particles at the same time. For this reason $\underline{\mathbf{r}}$ in (3.3.1) is a large vector containing the positions of every particle in the model. In this section we will only consider deformable surfaces consisting of M by N particles. This means that $\underline{\mathbf{r}}$ will be a vector of $M \times N$ elements, where each element is a 3D position vector. The components of the positions x, y, and z can be handled individually such that the elements of $\underline{\mathbf{r}}$ becomes numbers. Later on this will also make the examples less complicated.

In [21] no explanation is given on how to assemble the matrix $\mathbf{K}$, but since this is not just a trivial task we will now provide one possible explanation. The matrix $\mathbf{K}$ is called the *stiffness matrix* of the model and the assembled matrix is used in the semi-implicit method to solve (3.3.1) as a system of linear equations.

To get some idea of how the stiffness matrix looks, we first have to look at the calculation of the elasticity of a single particle. From 3.1 *Discretization* and 3.2 *Elasticity* we found that this calculation should be done using finite differencing. For the convenience of the reader, the expression of the discrete elasticity is repeated here

$$\mathbf{e}[m,n] = \sum_{i,j=1}^{2} -D_i^- (\mathbf{p})[m,n] + D_{ij}^{(-)}(\mathbf{q})[m,n], \tag{3.3.2}$$

with

$$\mathbf{p}[m,n] = \alpha_{ij}[m,n] D_j^+ (\mathbf{r})[m,n]$$

and

$$\mathbf{q}[m,n] = \beta_{ij}[m,n] D_{ij}^{(+)} (\mathbf{r})[m,n]. \tag{3.3.3}$$

The result that we would like to achieve, is the separation of the positional grid $\mathbf{r}$ from (3.3.2). To see how this is done, we take a look at some examples introduced in 3.2 *Elasticity* and develop them a bit further. This is done to identify the neighborhood stencils of both the first and second term in (3.3.2). We call these stencils the alpha and beta stencils, respectively.

3.3.1 The alpha stencil

When deriving the stencil of the neighborhood operations for the first term in (3.3.2), the goal is to identify what terms are multiplied on each of the neighbors of a particle position $\mathbf{r}[m,n]$, and it self. In 3.2.2.1 *Writing out the tension term* the following expression was arrived at, for the first term of (3.3.2)

$$\frac{-\alpha_{11}[m,n]\big(\mathbf{r}[m+1,n]-\mathbf{r}[m,n]\big)+\alpha_{11}[m-1,n]\big(\mathbf{r}[m,n]-\mathbf{r}[m-1,n]\big)}{{h_1}^2}. \tag{3.3.4}$$

In this regard, it is worth noting that (3.3.4) can also be written as

$$\begin{aligned}
&-\frac{\alpha_{11}[m,n]}{{h_1}^2}\mathbf{r}[m+1,n]\\
&+\frac{\alpha_{11}[m,n]+\alpha_{11}[m-1,n]}{{h_1}^2}\mathbf{r}[m,n]\\
&-\frac{\alpha_{11}[m-1,n]}{{h_1}^2}\mathbf{r}[m-1,n].
\end{aligned} \tag{3.3.5}$$

With (3.3.5) it becomes clear which terms are multiplied by which particle positions in the neighborhood of $\mathbf{r}[m,n]$. Writing out the whole first term of (3.3.2) in the same way yields

$$\begin{aligned}
&-\frac{\alpha_{11}[m,n]}{{h_1}^2}\mathbf{r}[m+1,n]\\
&+\frac{\alpha_{11}[m,n]+\alpha_{11}[m-1,n]}{{h_1}^2}\mathbf{r}[m,n]\\
&-\frac{\alpha_{11}[m-1,n]}{{h_1}^2}\mathbf{r}[m-1,n]
\end{aligned}$$

$$\begin{aligned}
&-\frac{\alpha_{12}[m,n]}{h_1h_2}\mathbf{r}[m,n+1]\\
&+\frac{\alpha_{12}[m,n]}{h_1h_2}\mathbf{r}[m,n]\\
&+\frac{\alpha_{12}[m-1,n]}{h_1h_2}\mathbf{r}[m-1,n+1]\\
&-\frac{\alpha_{12}[m-1,n]}{h_1h_2}\mathbf{r}[m-1,n]
\end{aligned}$$

$$
\begin{aligned}
&-\frac{\alpha_{21}[m,n]}{h_1 h_2}\mathbf{r}[m+1,n] \\
&+\frac{\alpha_{21}[m,n]}{h_1 h_2}\mathbf{r}[m,n] \\
&+\frac{\alpha_{21}[m,n-1]}{h_1 h_2}\mathbf{r}[m+1,n-1] \\
&-\frac{\alpha_{21}[m,n-1]}{h_1 h_2}\mathbf{r}[m,n-1] \\
&-\frac{\alpha_{22}[m,n]}{{h_2}^2}\mathbf{r}[m,n+1] \\
&+\frac{\alpha_{22}[m,n]+\alpha_{22}[m,n-1]}{{h_2}^2}\mathbf{r}[m,n] \\
&-\frac{\alpha_{22}[m,n-1]}{{h_2}^2}\mathbf{r}[m,n-1].
\end{aligned}
\tag{3.3.6}
$$

From (3.3.6) we are finally able to determine what we call the alpha stencil $\mathbf{a}$ of this neighborhood operation. The stencil looks like

$$
\begin{bmatrix}
\boxed{\mathbf{a}[m-1,n-1]} & \boxed{\mathbf{a}[m,n-1]} & \boxed{\mathbf{a}[m+1,n-1]} \\
\boxed{\mathbf{a}[m-1,n]} & \boxed{\mathbf{a}[m,n]} & \boxed{\mathbf{a}[m+1,n]} \\
\boxed{\mathbf{a}[m-1,n+1]} & \boxed{\mathbf{a}[m,n+1]} & \boxed{\mathbf{a}[m+1,n+1]}
\end{bmatrix},
\tag{3.3.7}
$$

The alpha stencil (3.3.7) is a 3×3 stencil, because none of the operations in (3.3.6) reach more than 1 place away from the particle at the center of attention $\mathbf{r}[m,n]$. The task of filling out the elements of (3.3.7) comes from looking at (3.3.6). The $\mathbf{a}$ stencil becomes

$$
\begin{bmatrix}
\boxed{0} & \boxed{-\frac{\alpha_{21}[m,n-1]}{h_1 h_2}-\frac{\alpha_{22}[m,n-1]}{{h_2}^2}} & \boxed{\frac{\alpha_{21}[m,n-1]}{h_1 h_2}} \\
\boxed{-\frac{\alpha_{11}[m-1,n]}{{h_1}^2}-\frac{\alpha_{12}[m-1,n]}{h_1 h_2}} & \boxed{\begin{aligned}&\frac{\alpha_{11}[m,n]+\alpha_{11}[m-1,n]}{{h_1}^2}+\\ &\frac{\alpha_{12}[m,n]+\alpha_{21}[m,n]}{h_1 h_2}+\\ &\frac{\alpha_{22}[m,n]+\alpha_{22}[m,n-1]}{{h_2}^2}\end{aligned}} & \boxed{-\frac{\alpha_{11}[m,n]}{{h_1}^2}-\frac{\alpha_{21}[m,n]}{h_1 h_2}} \\
\boxed{\frac{\alpha_{12}[m-1,n]}{h_1 h_2}} & \boxed{-\frac{\alpha_{12}[m,n]}{h_1 h_2}-\frac{\alpha_{22}[m,n]}{{h_2}^2}} & \boxed{0}
\end{bmatrix}
\tag{3.3.8}
$$

The alpha stencil is ready to be applied as a filter on any point of the surface, to calculate the first part of the elasticity. Think of this as a mask that can be put down on a particle on the surface. The mask will then cover a 3×3 neighborhood of a particle.

3.3.2 The beta stencil

The derivation of the neighborhood stencil for the second part of (3.3.2), is a very tedious task, since this is done in exactly the same way as the alpha stencil. There are some differences in the result though. The differences arise from the fact that second derivatives are involved in this part. The second derivatives simply results in an expansion of the corresponding stencil, which we will call the beta stencil $\mathbf{b}$ of size 5×5

$$\begin{array}{|c|c|c|c|c|}
\hline
\mathbf{b}[m-2,n-2] & \mathbf{b}[m-1,n-2] & \mathbf{b}[m,n-2] & \mathbf{b}[m+1,n-2] & \mathbf{b}[m+1,n-2] \\
\hline
\mathbf{b}[m-2,n-1] & \mathbf{b}[m-1,n-1] & \mathbf{b}[m,n-1] & \mathbf{b}[m+1,n-1] & \mathbf{b}[m+2,n-1] \\
\hline
\mathbf{b}[m-2,n] & \mathbf{b}[m-1,n] & \mathbf{b}[m,n] & \mathbf{b}[m+1,n] & \mathbf{b}[m+2,n] \\
\hline
\mathbf{b}[m-2,n+1] & \mathbf{b}[m-1,n+1] & \mathbf{b}[m,n+1] & \mathbf{b}[m+1,n+1] & \mathbf{b}[m+2,n+1] \\
\hline
\mathbf{b}[m-2,n+2] & \mathbf{b}[m-1,n+2] & \mathbf{b}[m,n+2] & \mathbf{b}[m+1,n+2] & \mathbf{b}[m+2,n+2] \\
\hline
\end{array} \tag{3.3.9}$$

In 3.2.3.1 *Writing out the curvature term* we arrived at the following expression for the second term in (3.3.2)

$$\begin{aligned}
&\frac{\beta_{11}[m+1,n]\big(\mathbf{r}[m+2,n]-2\mathbf{r}[m+1,n]+\mathbf{r}[m,n]\big)}{h_1^4} \\
&-\frac{2\beta_{11}[m,n]\big(\mathbf{r}[m+1,n]-2\mathbf{r}[m,n]+\mathbf{r}[m-1,n]\big)}{h_1^4} \\
&+\frac{\beta_{11}[m-1,n]\big(\mathbf{r}[m,n]-2\mathbf{r}[m-1,n]+\mathbf{r}[m-2,n]\big)}{h_1^4}.
\end{aligned} \tag{3.3.10}$$

At this point it should be evident that the values of $\mathbf{r}$ can be separated from the expression just like in (3.3.5) for the first term. The final stencil of $\mathbf{b}$ is seen in (3.3.9), where

$$\begin{aligned}
\mathbf{b}[m,n-2] &= \frac{\beta_{22}[m,n-1]}{h_2^4}, \\
\mathbf{b}[m-1,n-1] &= \frac{\beta_{12}[m-1,n-1]+\beta_{21}[m-1,n-1]}{h_1^2h_2^2}, \\
\mathbf{b}[m,n-1] &= \frac{-\beta_{12}[m,n-1]-\beta_{12}[m-1,n-1]-\beta_{21}[m,n-1]-\beta_{21}[m-1,n-1]}{h_1^2h_2^2} \\
&\quad+\frac{-2\beta_{22}[m,n]-2\beta_{22}[m,n-1]}{h_2^4}, \\
\mathbf{b}[m+1,n-1] &= \frac{\beta_{12}[m,n-1]+\beta_{21}[m,n-1]}{h_1^2h_2^2}, \\
\mathbf{b}[m-2,n] &= \frac{\beta_{11}[m-1,n]}{h_1^4}, \\
\mathbf{b}[m-1,n] &= \frac{-2\beta_{11}[m,n]-2\beta_{11}[m-1,n]}{h_1^4} \\
&\quad+\frac{-\beta_{12}[m-1,n]-\beta_{12}[m-1,n-1]-\beta_{21}[m-1,n]-\beta_{21}[m-1,n-1]}{h_1^2h_2^2},
\end{aligned}$$

$$
\begin{aligned}
\mathbf{b}[m,n] = \quad & \frac{\beta_{11}[m+1,n] + 4\beta_{11}[m,n] + \beta_{11}[m-1,n]}{h_1^4} \\
& + \frac{\beta_{12}[m,n] + \beta_{12}[m,n-1] + \beta_{21}[m-1,n] + \beta_{21}[m-1,n-1]}{h_1^2 h_2^2} \\
& + \frac{\beta_{21}[m,n] + \beta_{12}[m,n-1] + \beta_{21}[m-1,n] + \beta_{21}[m-1,n-1]}{h_1^2 h_2^2} \\
& + \frac{\beta_{22}[m,n+1] + 4\beta_{22}[m,n] + \beta_{22}[m,n-1]}{h_2^4}, \\
\mathbf{b}[m+1,n] = \quad & \frac{-2\beta_{11}[m+1,n] - 2\beta_{11}[m,n]}{h_1^4} \\
& + \frac{-\beta_{12}[m,n] - \beta_{12}[m,n-1] - \beta_{21}[m,n] - \beta_{21}[m,n-1]}{h_1^2 h_2^2}, \\
\mathbf{b}[m+2,n] = \quad & \frac{\beta_{11}[m+1,n]}{h_1^4}, \\
\mathbf{b}[m-1,n+1] = \quad & \frac{\beta_{12}[m-1,n] + \beta_{21}[m-1,n]}{h_1^2 h_2^2}, \\
\mathbf{b}[m,n+1] = \quad & \frac{-\beta_{12}[m,n] - \beta_{12}[m-1,n] - \beta_{21}[m,n] - \beta_{21}[m-1,n]}{h_1^2 h_2^2} \\
& + \frac{-2\beta_{22}[m,n+1] - 2\beta_{22}[m,n]}{h_2^4}, \\
\mathbf{b}[m+1,n+1] = \quad & \frac{\beta_{12}[m,n] + \beta_{21}[m,n]}{h_1^2 h_2^2}, \\
\mathbf{b}[m,n+2] = \quad & \frac{\beta_{22}[m,n+1]}{h_2^4},
\end{aligned}
\tag{3.3.11}
$$

and all the unused slots are zero.
Again like with stencil $\mathbf{a}$, this can now be applied to a particle on the surface. The final stencil we will call $\mathbf{s}$, and it will be the concatenation of the $\mathbf{a}$ and $\mathbf{b}$ stencils, thus it will look like

$\mathbf{b}[m-2,n-2]$	$\mathbf{b}[m-1,n-2]$	$\mathbf{b}[m,n-2]$	$\mathbf{b}[m+1,n-2]$	$\mathbf{b}[m+1,n-2]$
$\mathbf{b}[m-2,n-1]$	$\mathbf{a}[m-1,n-1]$ $+\mathbf{b}[m-1,n-1]$	$\mathbf{a}[m,n-1]$ $+\mathbf{b}[m,n-1]$	$\mathbf{a}[m+1,n-1]$ $+\mathbf{b}[m+1,n-1]$	$\mathbf{b}[m+2,n-1]$
$\mathbf{b}[m-2,n]$	$\mathbf{a}[m-1,n]$ $+\mathbf{b}[m-1,n]$	$\mathbf{a}[m,n]$ $+\mathbf{b}[m,n]$	$\mathbf{a}[m+1,n]$ $+\mathbf{b}[m+1,n]$	$\mathbf{b}[m+2,n]$
$\mathbf{b}[m-2,n+1]$	$\mathbf{a}[m-1,n+1]$ $+\mathbf{b}[m-1,n+1]$	$\mathbf{a}[m,n+1]$ $+\mathbf{b}[m,n+1]$	$\mathbf{a}[m+1,n+1]$ $+\mathbf{b}[m+1,n+1]$	$\mathbf{b}[m+2,n+1]$
$\mathbf{b}[m-2,n+2]$	$\mathbf{b}[m-1,n+2]$	$\mathbf{b}[m,n+2]$	$\mathbf{b}[m+1,n+2]$	$\mathbf{b}[m+2,n+2]$

(3.3.12)

This stencil includes all of (3.3.2), and it contains what we need, to finally understand how the stiffness matrix $\mathbf{K}$ is assembled.

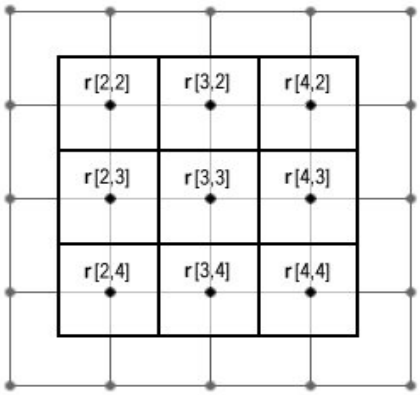


Figure 3.3 A discrete surface grid of 25 particles, where the 9 center particles has been grouped into a stencil

3.3.3 The K matrix

We note that the vector $\underline{\mathbf{r}}$ in (3.3.1), is a vector collection containing positions for all particles. This means that when $\underline{\mathbf{r}}$ is multiplied with $\mathbf{K}$ we will get the elasticity of all particles in a vector $\underline{\mathbf{e}}$ of size MN. It might sound a bit confusing at first, but it gets easier when remembering that matrix/vector multiplication is done row by row. Multiplication of $\underline{\mathbf{r}}$ with the first row in $\mathbf{K}$ should result in the elasticity for the first particle placed in $\underline{\mathbf{e}}$. This means that we have to unwrap the stencil $\mathbf{s}$ on a particle in $\underline{\mathbf{e}}$, onto the corresponding row in the matrix. This unwrapping have to be done in the same way that the particles were unwrapped into $\underline{\mathbf{r}}$, either by row or by column, so that the right terms will fit together when computing $\mathbf{K}(\underline{\mathbf{r}})\underline{\mathbf{r}}$.
The row-wise unwrapping of $\mathbf{s}$, could be written as

$$\mathbf{s}[m-2,n-2] \quad \mathbf{s}[m-1,n-2] \quad \cdots \quad \mathbf{s}[m,n] \quad \cdots \quad \mathbf{s}[m+1,n+2] \quad \mathbf{s}[m+2,n+2], \quad (3.3.13)$$

but the only problem with this procedure is that in order to match $\underline{\mathbf{r}}$ in the multiplication, zeros need to be inserted to make an MN sized row. Zeros are placed between “rows” of the unwrapped stencil. For example the first row of the stencil is

$$\mathbf{s}[m-2,n-2] \quad \mathbf{s}[m-1,n-2] \quad \mathbf{s}[m,n-2] \quad \mathbf{s}[m+1,n-2] \quad \mathbf{s}[m+2,n-2], \quad (3.3.14)$$

and the corresponding particles in $\underline{\mathbf{r}}$ of these stencil slots, will be just below each other because $\underline{\mathbf{r}}$ is also unwrapped by row. But the corresponding particle of the first slot in the next row of the stencil $\mathbf{s}[m-2,n-1]$, will not be below the particle corresponding to $\mathbf{s}[m+2,n-2]$. This is because the rows of particles on the surface, are much longer than the size of a row in the stencil which is 5. For example, the particle $\mathbf{r}[m,n]$ is just above $\mathbf{r}[m,n+1]$ in the 2D grid, but in the unwrapped vector $\underline{\mathbf{r}}$, they will be separated by M elements, which is the number of particles of a row in the grid. The stencil positions, $\mathbf{s}[m+2,n-2]$ and $\mathbf{s}[m-2,n-1]$, will be separated by $M-5$ zeros in $\mathbf{K}$. A small example to illustrate this spacing might be useful. Let us assume that the surface has been discretized into a 5×5 grid of particles, and that we are looking at a 3×3

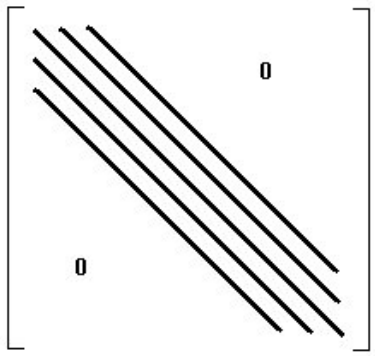


Figure 3.4 Banded sparse matrix

stencil $\mathbf{t}$, placed on the surface with the center element $\mathbf{r}[3,3]$, as the center of attention, as depicted in Figure 3.3.
Storing all 25 particle positions into the vector collection $\underline{\mathbf{r}}$ yields

$$\underline{\mathbf{r}} = \begin{pmatrix} \mathbf{r}[1,1] & \mathbf{r}[2,1] & \mathbf{r}[3,1] & \mathbf{r}[4,1] & \mathbf{r}[5,1] & \mathbf{r}[1,2] & \mathbf{r}[2,2] & \cdots & \mathbf{r}[5,5] \end{pmatrix}^T, \quad (3.3.15)$$

and mapping (3.3.15) into the one dimensional index is simply

$$\underline{\mathbf{r}} = \begin{pmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 & \mathbf{r}_4 & \mathbf{r}_5 & \mathbf{r}_6 & \mathbf{r}_7 & \cdots & \mathbf{r}_{25} \end{pmatrix}^T. \quad (3.3.16)$$

By looking at Figure 3.3 and (3.3.16) it can be verified that the first row of the stencil $\mathbf{t}$ covers the three subsequent particle positions $\mathbf{r}_7$, $\mathbf{r}_8$, and $\mathbf{r}_9$, while the next row covers $\mathbf{r}_{12}$, $\mathbf{r}_{13}$, and $\mathbf{r}_{14}$. It is this spacing that creates the zero padding in the $\mathbf{K}$ matrix. So if a similar $\mathbf{K}$ matrix were to be constructed for this example, we would know that it would be of size 25×25 and that the 13th row corresponding to $\mathbf{r}_{13}$, or $\mathbf{r}[3,3]$, would be the row

$$\begin{matrix} 0_1 & \cdots & 0_6 & \mathbf{t}[m-1,n-1]_7 & \mathbf{t}[m,n-1]_8 & \mathbf{t}[m+1,n-1]_9 & 0_{10} & 0_{11} \\ \mathbf{t}[m-1,n]_{12} & \mathbf{t}[m,n]_{13} & \mathbf{t}[m+1,n]_{14} & 0_{15} & 0_{16} \\ \mathbf{t}[m-1,n+1]_{17} & \mathbf{t}[m,n+1]_{18} & \mathbf{t}[m+1,n+1]_{19} & 0_{20} & \cdots & 0_{25}. \end{matrix} \quad (3.3.17)$$

The subscripts of the elements in (3.3.17), indicate which column in the row that the element is placed in. The indices should agree with the placing of the stencil in Figure 3.3, and the corresponding indices in (3.3.16).

We have tried to illustrate how the final $\mathbf{K}$ matrix looks. Unfortunately it is too large to fit on these pages, instead we have included it as an image on the supplemental CD [3], under the name "`stiffness_matrix_K`". Just like in the previous example, the subscripts indicate which column of the matrix we are looking at. An important thing to understand about this matrix is that it is banded. This is because the rows of the stencil $\mathbf{s}$ are propagated diagonally through the matrix for every particle. The end result will be five diagonal bands as illustrated in Figure 3.4, each representing a row in the stencil $\mathbf{s}$, and separated by zeros, as discussed in the example above.

The stiffness matrix $\mathbf{K}$ has some desirable computational properties that make it relatively fast to invert. It is banded because the unwrapped row of the stencil is simply shifted by one for

every row. It is also sparse, symmetrical and positive definite. These properties will be considered in 3.8 *Numerical Integration*, when choosing a method to solve the system.

3.3.4 Numerical instabilities

Because we are using a computer to perform the simulation, we are forced into numerical instabilities. Numerical instabilities in the floating point calculations are unavoidable, but are important issues when it comes to the handling of a stable system. If entries in the stiffness matrix are very small, they will become very large when the matrix is inverted. Over time these numerical features affect the final result of computing new particle positions. The most typical problem we have encountered is a body that has retuned to its equilibrium, suddenly starts deforming oddly, and the deformation obviously never stops.

A simple hack to insure that a deformable body remains stable, is to cut away all contributions to the stiffness matrix that is below some predefined threshold. This procedure works excessively well in practice, and is one of the things that secure a stable system in the implementation.

3.4 Boundaries

In this section we will discuss the issues we have encountered regarding the boundaries of the Discretized grid. During 3.3 *The Stiffness Matrix* we saw how the finite difference operators on the vector/tensor fields were working. The resulting indexes reached both 1 and 2 nodes away from the particle at hand, in the directions parallel to the grid axes. Our intuition tells us that problems will occur on the grid boundaries. The problems come mainly from the fact that the discrete deformable model heavily depends on being smooth everywhere, which is only true for models with no boundaries. If the model for example is a rectangular carpet approximated by a deformable Bézier patch, the model will have discontinuities at the edges, thus it will not be differentiable there.

We have used two strategies to overcome the boundary problems, both will be described next. We have also implemented support for models with continuous, or periodic, boundaries, which we will discuss in between.

3.4.1 Projection method

Usually, objects have boundaries (or edges). This gives problems when evaluating derivatives at these places. These derivatives are among other things used to calculate the normals for the surface, at the position of the particles. Some methods are used in the implementation to counteract the lack of information about a surface on the other side of its edges. One method is simple orthogonal projection, which, among other things, is used when calculating the derivatives for the normal calculations. A surface normal at a particle is calculated as a cross product between the derivatives in each of the coordinate directions. These derivatives are, for the normal calculation, calculated by central differences. For further information about surface normals please go to 3.7 *Surface Normals*. A central difference calculation includes particle positions from the next and the previous particle, in the coordinate direction. When this difference is performed on an edge, as in Figure 3.5, information is missing from across the edge. The dots in the figure represent the grid particles, and the box shows the particles involved in the central difference calculation. The circled particle is the particle having its central difference computed. This is where the projection method comes into play. When ever the positional grid is indexed, a check is performed to see if the index lies outside the grid. If this is the case, the value of this index will be the value gotten by projecting orthogonally onto the grid.

When information is needed for a difference calculation on the other side of a boundary, the positional value of the nearest particle is used. This concept is illustrated in Figure 3.6. The labels **x** in this figure describe some index values that lie outside the particle grid. The arrows tell which particle values that are used instead, after the orthogonal projection.

The projection algorithm can be implemented quite simple. In Listing 3.1 the algorithm for the adjustment is given. It uses the `min` and `max` functions to make sure that the indices are clamped

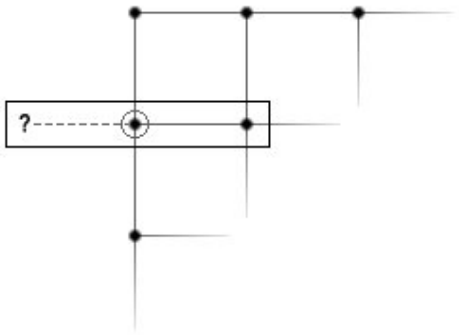


Figure 3.5 A central difference taken across an edge

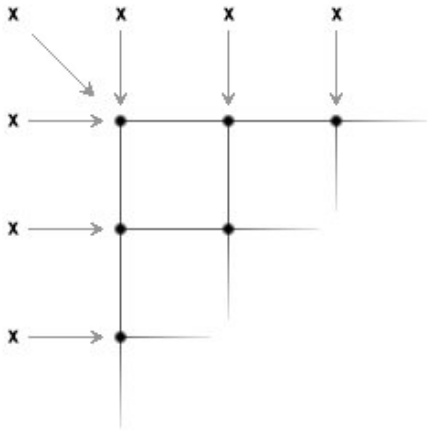


Figure 3.6 The orthogonal projection method

to the nearest grid particle, if they reference outside of the grid. The constants M and N are the number of particles in the two directions. It should be noted that m and n are both zero based indices.

```
algorithm adjust_index(long &m, long &n)
{
  m = max(0, min(m, M-1));
  n = max(0, min(n, N-1));
}
```

Listing 3.1 The index adjustment algorithm

The effect of the projection method could also have been achieved by expanding the particle grid with ghost particles that have the same values as the particles on the edges of the grid. The projection method is simply more effective during maintenance, but also memory wise.

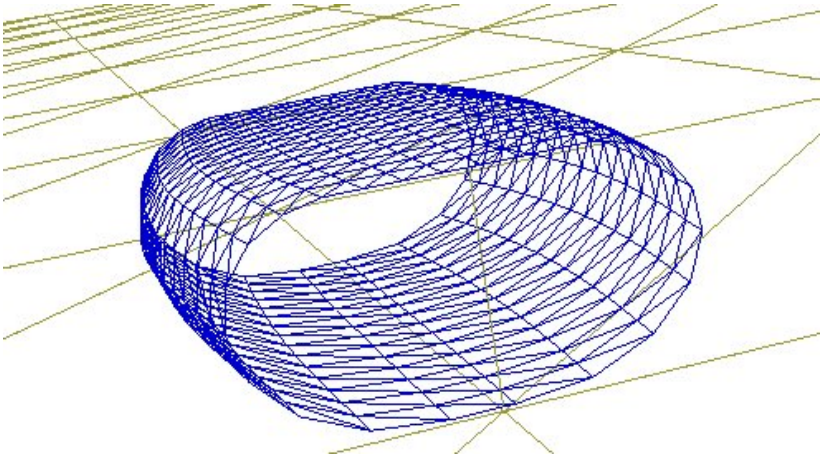


Figure 3.7 Surfaces with continuous boundaries in one coordinate direction

3.4.2 Continuous boundaries

In our implementation of the deformable model it is possible to create surface patches with continuous boundaries. This of course implies that the object is visually circular of some form or another. A surface patch is parameterized by two parameters, one for each direction on the surface. If one of the directions were described as having no boundary, the implementation handles this as a ”wrap-around” for the particles on the two edges in that direction.

The concept is visually depicted on Figure 3.7, and mathematically it means that the object is differentiable everywhere in that direction. When approximating the derivatives on the edges of the grid, values of the particles on the opposite edge are used. The smallest value of the indices m and n is 0, thus if at any point a negative value is given as an index for a continuous coordinate direction, the index is wrapped around to the opposite edge. Such an edge starts with index $M-1$ or $N-1$, which is the highest index possible for a particle. The wrap-around also occurs in the opposite direction. If either the m or the n index is specified as a continuous coordinate direction, the adjusted indices can be computed as the algorithm given in Listing 3.2. The algorithm is an expansion from Listing 3.1, where `wrap_m` and `wrap_n` are Boolean constants indicating if continuous boundaries are applied to either direction.

```
algorithm adjust_index_wrap(long &m, long &n)
{
  if (wrap_m)
    m = ((m % -M) + M) % M;
  else
    m = max(0, min(m, M-1));
  if (wrap_n)
    n = ((n % -N) + N) % N;
  else
    n = max(0, min(n, N-1));
}
```

Listing 3.2 The index adjustment algorithm, with wrapping expansion

3.4.3 Stiffness boundary conditions

Another method that is used to handle problems with the boundaries is brought up when the large stiffness matrix $\mathbf{K}$ is to be assembled. Simulation of a natural boundary condition can be

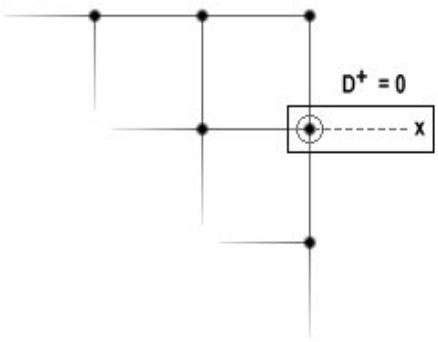


Figure 3.8 The forward difference operator working beyond the boundary

achieved by setting to zero every forward differencing operator that involves values on both sides of a boundary. In Figure 3.8, we have given an example of a forward difference operation that reaches beyond the grid boundary. If such a situation occurs, then the result of the difference operation must be zero.
Of course this goes for both first and second derivative approximations. Every time forward differencing is done in $\mathbf{K}$, a check needs to be done to see if the value of the operator should be set to zero. The index check is fairly simple to make, as it is very much like the adjustment algorithms in Listing 3.1 and Listing 3.2. It should take advantage of any coordinate directions specified as continuous. Since continuous coordinate directions are differentiable in every index, setting difference calculations to zero is not required. The matrix $\mathbf{K}$ was essentially derived from the elasticity equation (3.3.2). It is the particle indices involved in the forward differences (3.3.3) that should be checked for lying outside of the grid. When implementing the stencils specified in 3.3 *The Stiffness Matrix*, this should always be kept in mind. The index check algorithm on m and n, could be done as shown in Listing 3.3, where the Boolean return value indicates, if the provided index pair is valid or not, and thus if the results from the difference operation should be applied to the final $\mathbf{K}$ matrix. It should be noted that if continuous boundaries are used on any direction, the index check on that direction always succeeds.

```
bool index_check(long m, long n)
{
  bool m_ok = true;
  bool n_ok = true;
  if (!wrap_m)
    m_ok = 0 <= m && m < M;
  if (!wrap_n)
    n_ok = 0 <= n && n < N;
  return m_ok && n_ok;
}
```

Listing 3.3 The index verity algorithm

3.5 External Forces

The right hand side of (2.1.1)

$$\mathbf{f}(\mathbf{r}, t), \tag{3.5.1}$$

represents the net externally applied forces, which balance the internal elastic forces of the deformable model. The net external force (3.5.1), is the sum of all individual external forces working

a) Solid planetary bodies

Body	Mercury	Venus	Earth	Moon	Mars	Pluto
$\mathbf{g}\ (m\mathbf{s}^{-2})$	3.70	8.87	9.798	1.62	3.71	0.58

b) Gaseous planetary bodies

Body	Sun	Jupiter	Saturn	Uranus	Neptune
$\mathbf{g}\ (m\mathbf{s}^{-2})$	274.00	24.79	10.44	8.87	11.15

Table 3.1 Different gravitational accelerations from planetary bodies in our solar system

on $\mathbf{r}(\mathbf{a})$. If each individual external force is indexed by i, the total applied external force can also be written as

$$\mathbf{f}_{total} = \sum_i \mathbf{f}_i(\mathbf{r}, t). \tag{3.5.2}$$

Applying external forces to elastic deformable bodies yields a comprehensive realistic behavior of the dynamics of the final animation. This section will describe some of the possible external forces that can enter into the sum in (3.5.2), with one exception; the external collision force. This force is described in greater detail in 3.6.1 *Penalty collision force*, but it is, after all, just an external force, like the rest of the forces from this section, thus its contribution should equally be added to $\mathbf{f}_{total}$. Through this section we can distinguish between external forces that work globally and external forces that only work on surfaces, as external forces that work globally, do not contain the surface normal in their equations.

3.5.1 Gravity

The force of gravity is vital for all systems that wish to simulate objects on any planetary globe. The gravitational force is acting on everyone and everything that has a mass, and it is given by

$$\mathbf{f}_{gravity}(\mathbf{a}) = \mu(\mathbf{a})\mathbf{g}, \tag{3.5.3}$$

where μ is the mass density of the particle $\mathbf{a}$ and $\mathbf{g}$ is the gravitational acceleration. In Table 3.1, we have shown the average gravitational acceleration of the most known planetary bodies in our solar system [28]. The direction of $\mathbf{g}$ always points towards the center of mass, whether it is given in the body frame or world coordinates. As we normally consider gravity to pull bodies downwards towards the planetary globe it is working on, the gravitational field around our present location on Earth[1] is

$$\mathbf{g} = [0, 0, -9.82]^T, \tag{3.5.4}$$

and this constant of gravity can be seen several places in the scene configurations located on the supplemental CD [3]. As (3.5.4) reveals, we have chosen the gravity to work along the negative z-axis. This could of course be any of the three axes (x, y, z). It is a matter of how the world scene has been designed. If the force of gravity is provided to the simulated system, it should work on every particle in the discrete version of the deformable body. The gravity force can also be used to provide any other constant force to the simulated system.

[1] The authors of this paper are living in Europe, Denmark

3.5.2 Viscosity

The viscosity of a fluid can be defined as a measure of how resistant the fluid is to a flow field. The force on the surface of an elastically deformable body, due to viscous fluid is

$$\mathbf{f}_{viscous}(\mathbf{a},t) = c\big(\mathbf{n}(\mathbf{a},t)\cdot\widehat{\mathbf{v}}(\mathbf{a},t)\big)\mathbf{n}(\mathbf{a},t), \tag{3.5.5}$$

where c is the strength of the fluid force, $\mathbf{n}$ is the unit surface normal at $\mathbf{a}$ on the surface of the body, and $\widehat{\mathbf{v}}$ is the relative velocity of the surface at particle $\mathbf{a}$. It is given by

$$\widehat{\mathbf{v}}(\mathbf{a},t) = \mathbf{u} - \frac{\partial\mathbf{r}(\mathbf{a},t)}{\partial t}, \tag{3.5.6}$$

where $\mathbf{u}$ is some constant stream velocity of the fluid. It is worth mentioning that the last term of (3.5.6) is the current velocity of the particle

$$\mathbf{v}(\mathbf{a},t) = \frac{\partial\mathbf{r}(\mathbf{a},t)}{\partial t}. \tag{3.5.7}$$

Imagine a ball rolling on a clean smooth horizontal surface of some table. If the ball, at some point on the surface, is rolling through a puddle of water, the ball will slow down, due to the viscosity of the water. If the viscous force is provided to a system, without using the constant stream velocity $\mathbf{u}$, the same effect can be simulated as when the ball was rolling through the puddle. The stream velocity in (3.5.6) describes some constant fluid velocity in the flow field, e.g. the current we can be grazed by when swimming in the ocean at summer time. Viscosity can be used to damp the simulated system, as well as create other types of interesting forces. A wind force can be obtained using (3.5.5) with a constant stream velocity in (3.5.6) different from zero, $\mathbf{u} \neq \mathbf{0}$, see 6.1.2 *Deformable open surfaces*. Remember the viscous force should only be applied to surfaces.

3.5.3 Spring

The restoring spring force is given by Hooke's law

$$\mathbf{F} = m\mathbf{a} = m\frac{d^2\mathbf{x}}{dt^2} = -k\mathbf{x}, \tag{3.5.8}$$

where $\mathbf{x}$ is the displacement vector from the spring's equilibrium space point and k is the spring constant. When $\mathbf{x} = \mathbf{0}$ the spring is by default in equilibrium, thus when $\mathbf{x} \neq \mathbf{0}$ the magnitude of $\mathbf{x}$ is equivalent to the distance from $\mathbf{x}$ to the equilibrium position, when the equilibrium position is considered as a local origin of the spring. The displacement $\mathbf{x}$ is given by

$$\mathbf{x} = \mathbf{r}_t - \mathbf{r}_0, \tag{3.5.9}$$

where $\mathbf{r}_t$ is the current position and $\mathbf{r}_0$ is the position of equilibrium. The direction of $\mathbf{x}$ in (3.5.9) always points away from the equilibrium, and this is why Hooke's law originally negates k in (3.5.8).
The differential equation describing the motion of the spring is

$$\frac{d^2\mathbf{x}}{dt^2} + \frac{k}{m}\mathbf{x} = \mathbf{0}, \tag{3.5.10}$$

which is a simple harmonic oscillator [10].

Hooke's law works for all types of displacements and likewise in any dimension. The spring force that connects a specific particle $\mathbf{a}_0$ to a fixed point $\mathbf{r}_0$ in world coordinates, is given by

$$\mathbf{f}_{spring}(t) = -k\big(\mathbf{r}(\mathbf{a}_0,t) - \mathbf{r}_0\big) = k\big(\mathbf{r}_0 - \mathbf{r}(\mathbf{a}_0,t)\big). \tag{3.5.11}$$

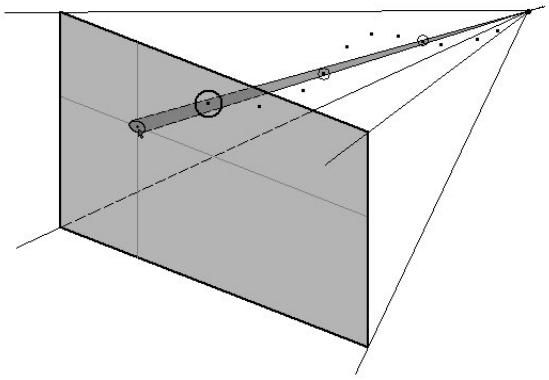


Figure 3.9 Pointing device marks the centre of the search circle, and all particles within this search range are found, but only the nearest particle is selected

Any particle can be constrained to a fixed position by the elastically spring in (3.5.11). Particles can also be rendered frozen by the effect of position constraints. Several methods exist to lock a particle. Its mass can be infinite to render movement impossible, but this requires forces like gravity to be disregarded. Another consequence of using infinite heavy particles is instabilities within the system. A more practicable and easy method is to include a property on a particle that informs the system if the particle is fixed or not, and thus only update the position if the particle is movable. We have implemented the latter method, and it works very well.

3.5.4 User controlled forces

We define user controlled forces to be either pulling or pushing operations, performed on arbitrary particles by the end user. User controlled forces are important when it comes to convincing viewers of the stability and realism of the simulated system. It also provides a certain feeling of "playing god" when the user is able to throw around with a fully elastically deformable object.

In the sense of the external forces described through this section, user controlled forces cannot be defined as such forces. A user controlled force is merely a method of direct vertex manipulation, which can be integrated into the system perfectly, due to the fact that particle velocities are given implicitly as we will discuss in 3.6.2 *Projection*.

3.5.4.1 Particle selection

Letting the end user grab particles using a pointing device, such as a mouse or another electronic pointer, enables the system to support interactive pull and push operations on particles. For the end user to grab particles with his pointing device, we must be able to transform the 2D screen position provided by the pointing device into the current 3D view volume. In the viewport the three dimensions all range form $[-1..1]$, and we choose $z = -1$ to be nearest to the end user. Assume we are able to transform a window position into a uniform viewport position, we can now compare our position in view volume coordinates against all the particle positions that also must be represented in view volume coordinates. If we come across more than one particle lying on the same line through the z-direction, we choose the particle with the smallest z-value, which by default is the one closest to the end user, and most likely is the one the user is aiming at.

The above method works in theory, but not very often in practice as a particle must lie exactly on the same xy position as the pointing device. To let the method work in practice as well, we expand the search area from a single point to a circle. Given a search position in viewport coordi-

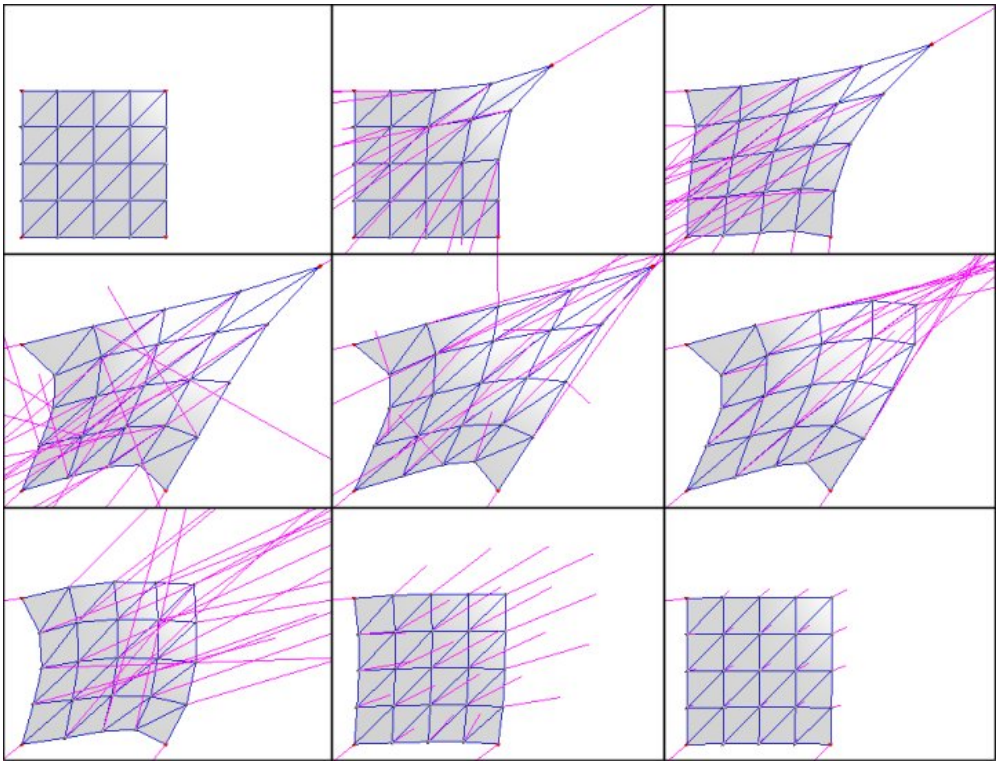


Figure 3.10 9 frames from a motion sequence, with visual debug turned on, showing the upper right particle being pulled away from the object and then released

nates we will use this position as the center of the circle and extend the area with some search radius. We will select the particle at hand if the following holds

$$(x - x_p)^2 + (y - y_p)^2 \leq \varepsilon^2 \quad \wedge \quad z < z_0 \,, \tag{3.5.12}$$

where (x, y, z) is the position of the particle being compared in view volume coordinates, (x_p, y_p) is the position of the pointing device also in view volume coordinates, ε is the circle search radius, and z_0 is the z-coordinate of the nearest particle found so far. The method is depicted in Figure 3.9.

We have basically used the window-to-viewport transformation method by Foley et al. from [9], but with some small modification in order to let the hardware struggle with the computations. As previously stated we will not go into details about the underlying and chosen rendering system, but only mention that we have implemented the transformations using OpenGL that provided us with the needed system matrices and vectors, along with hardware supported functions to map from 2D to 3D and vice versa.

3.5.4.2 The pull operation

Once the particle selection has taken place, the pointing device can be moved to another location on the screen, and if the particle is moved to this new location we will gain the pull effect. The

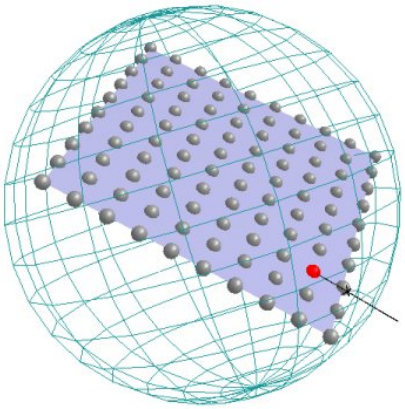


Figure 3.11 An EDO surrounded by a bounding sphere with a selected particle (red) and its projection onto the sphere

same procedure that transformed the pointing device from the 2D screen to the 3D world upon particle selection must be used again, to calculate the new 3D coordinate in the viewport. The direction and distance a grabbed particle is being pulled can be given as

$$\mathbf{f}_{pull} = \mathbf{p} - \mathbf{p}_0 , \tag{3.5.13}$$

where $\mathbf{p}$ is the new pointing device location in the viewport and $\mathbf{p}_0$ is the current position of the selected particle.

We have implemented the pull operation, and it enables us to perform particle pulling quite convincingly, as the small slide sequence on figure 3.10 demonstrates. The entire animation of the pull sequence is called "`pull.avi`" and it can be found on the supplemental CD [3].

3.5.4.3 The push operation

The push operation also continues from the actual particle selection. Once a particle has been selected, it is a straight forward matter of choosing a convincing method to find the "punch" vector. The punch-vector both describes the pushing direction and the strength of the punch. The strength of the punch can be a fixed value determined by a user defined constant, or perhaps the more obvious method of calculating the strength as a function of the state of a mouse button, e.g. the longer a mouse button is held down, the more powerful the punch will be. The direction of the vector should somehow give the idea that the end user is pushing the particle that has been chosen. A dull solution will be just to push the particle in the z-direction of the viewport, thus pushing it further into the monitor or away from the user. On the other hand, we would rather like the pushing to be dependent of the pointing device compared to the particle. During a pull operation we could ignore that it was actually a 3-dimensional world, we were operating in, and thus consider the current viewport as a 2D world. We will now give a short description of a dependent pushing direction with respect to the particle being pushed and the shape of the deformable object.

Assume a bounding sphere can be found that contains the entire deformable object, as illustrated in Figure 3.11. The selected particle has been marked. Our suggestion for the direction of the push vector, is to use the reverse projection from the selected particle onto the sphere. It might be problematic to find such a projection, as we assume the projection is the shortest possible distance from the sphere to the particle. Such property also tells us that the projection from the particle is perpendicular to the sphere, and thus gives us a really nice method to determine the push direction. Describing the bounding sphere as an implicit function, we know from 3.6.1.1

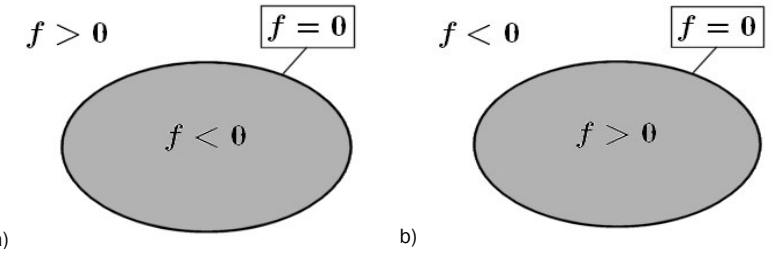


Figure 3.12 The sign of an implicit function f evaluated outside, inside, and on the surface boundary

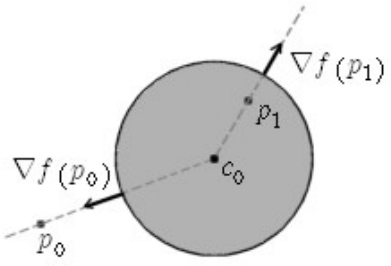


Figure 3.13 Unit gradients evaluated at different points, inside and outside the surface

Implicit surfaces that the implicit function can be designed such that the gradient evaluated at some point yields the reverse of the desired direction. The push vector can now be assembled to

$$\mathbf{f}_{push} = -\nu \frac{\nabla b(\mathbf{p})}{|\nabla b(\mathbf{p})|}, \tag{3.5.14}$$

where $b : \mathbb{R}^3 \, \mathrm{x} \, \mathbb{R}$ is the implicit function describing the bounding sphere, $\mathbf{p}$ is the coordinates of the selected particle, and $\nu > 0$ is the strength of the punch.

3.6 Collision

An important issue when simulating elastically deformable bodies, is the reaction to collision against other objects in the virtual world. An elastically deformable body at rest can only deform if some external force is applied to it, e.g. a pushing collision force or a pulling force, obtained by manual user interaction, see 3.5.4 *User controlled forces*. Throughout the next subsections, we will focus on collision between a deformable body and impenetrable rigid objects, using penalty force methods as in [21], but also briefly discuss collision using a projection method. We will give a brief survey on geometric methods to handle collisions for deformable models, which will include the subject of self-intersection.

3.6.1 Penalty collision force

In [21] collisions between the deformable models and impenetrable rigid objects, are simulated by providing a potential energy function around each object

$$c \exp\left(-\frac{f(\mathbf{r})}{\varepsilon}\right), \tag{3.6.1}$$

where $f(x,y,z)$ is the implicit given function, describing the existence of the object's surface, and c and ε are constants defining the potential of the shape. Such a potential energy function simulates a repulsive power. It must be continuous and determine a large value inside the object, and zero on the surface and outside of the object. Before we continue let us introduce a quick survey on implicit surfaces.

3.6.1.1 Implicit surfaces

An implicit surface consists of points $\mathbf{p} \in \mathbb{R}^3$ that satisfy some mathematical requirement. This requirement is presented by an implicit function $f : \mathbb{R}^3 \text{ x } \mathbb{R}$ that does not explicitly describe the surface, but merely implies its existence. If $f(\mathbf{p}) = 0$ then the point $\mathbf{p}$ is by definition on the surface.
Some implicit surfaces have a unique property that the value of the implicit function, evaluated at the point $\mathbf{p}(x,y,z)$, is proportional to the distance from $\mathbf{p}$ perpendicular to the surface. Unfortunately, this property does not hold for all implicit surfaces. For the same reason a non-zero value of $f(\mathbf{p})$, is typically used as an indication of whether or not the point $\mathbf{p}$ penetrates an implicit described two-manifold. A common issue with an implicit surface, is whether its function is positive or negative upon evaluation at a point $\mathbf{p}$ inside the boundaries of the surface, see Figure 3.12. However, throughout this paper we will only use the latter, i.e. if $f(\mathbf{p}) < 0$, we define the point $\mathbf{p}$ to be inside the surface, as illustrated in Figure 3.12a.
Another property of an implicit function is that any point $\mathbf{p}(x,y,z)$ evaluated at the gradient of the function, will result in a vector perpendicular to the implicit surface[2], see Figure 3.13. By this property the normal of an implicit surface is

$$\mathbf{n}_f \equiv \nabla f = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}, \tag{3.6.2}$$

where the surface is defined by the implicit function f and $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are the 3 basis vectors. Normalizing the gradient will result in a unit surface normal.

Please note that the direction of the gradient ∇f, is directly influenced by how the implicit surface f is chosen. Choosing f such that $f(\mathbf{p}) < 0$ when $\mathbf{p}$ is inside the surface boundary and $f(\mathbf{p}) > 0$ when $\mathbf{p}$ is outside, the direction of the gradient is proportional to the surface normal, as in (3.6.2). Choosing the implicit surface f with the properties as shown in Figure 3.12b, the direction of the surface normal is inversely proportional to the gradient, i.e. $\mathbf{n}_f \equiv -\nabla f$. Further details about implicit functions and implicit surfaces in general can be found in [2, 8].

3.6.1.2 Analysis

In [21] the penalty collision method is modeled as a conservative force, which is the negative gradient of the energy potential (3.6.1). The penalty collision force is given by

---

[2] However, if the function f is a closed manifold and c_0 is the absolute centre of the surface then $\nabla f(c_0) = \mathbf{0}$

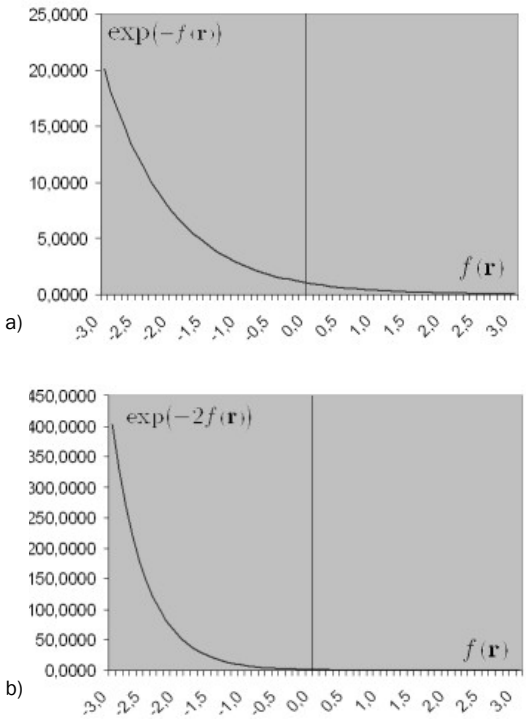


Figure 3.14 The behavior of $c\exp\left(-\frac{f(\mathbf{r})}{\varepsilon}\right)$ **a)** $c=\varepsilon=1$ **b)** $c=1,\ \varepsilon=\frac{1}{2}$

$$\mathbf{f}_{collision} = -\left(\frac{\nabla f(\mathbf{r})}{\varepsilon}\exp\left(-\frac{f(\mathbf{r})}{\varepsilon}\right)\cdot\mathbf{n}\right)\mathbf{n}\,, \tag{3.6.3}$$

where $\mathbf{n}(\mathbf{a})$ is the unit surface normal of the deformable body at $\mathbf{a}$. We will argue that the authors of [21] have reversed the collision force in (3.6.3), which will result in a pulling effect rather than a pushing effect as expected. Let us analyze the equation.

Part 1

The effect of using $\mathbf{f}_{collision}$ should be to force any particle out of objects it penetrates. Equation (3.6.1) is the main contributor for this approach. Figure 3.14 illustrates the charts of 2 examples of the behavior of (3.6.1).

Considering (3.6.1) and Figure 3.14 it is clear that $f(\mathbf{r})$ must return a negative value when $\mathbf{r}$ is inside the surface. Put in another way, the farther inside the surface boundary a particle is, the stronger the penalty force will be. It is clear that we must guarantee that each implicit function f defining an object's shape must return a negative value when the function is evaluated at a position within the object's surface boundary. This is consistent with the property of an energy potential function. If $f(\mathbf{r})$ was chosen to be positive inside the surface, we are forced to use a negative epsilon ε to secure the correct behavior of the potential.

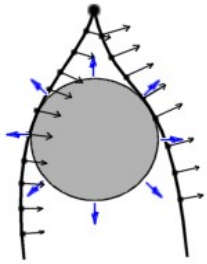


Figure 3.15 Directions of the gradients of an object and surface normals of a deformable model on each side of the object

Part 2

Rewriting (3.6.3) into

$$\mathbf{f}_{collision} = -\big(s\nabla f(\mathbf{r})\cdot\mathbf{n}\big)\mathbf{n}\,, \tag{3.6.4}$$

where $s = \varepsilon^{-1}\exp\big(-\varepsilon^{-1}f(\mathbf{r})\big)$ shows that the sign of $s\nabla f\cdot\mathbf{n}$ is dependent on two situations. The first one is regarding the dot product between 2 vectors

$$\cos\theta = \frac{\mathbf{v}\cdot\mathbf{w}}{|\mathbf{v}||\mathbf{w}|} \;\Leftrightarrow\; \mathbf{v}\cdot\mathbf{w} = |\mathbf{v}||\mathbf{w}|\cos\theta\,. \tag{3.6.5}$$

Equation (3.6.5) is only introduced to support the next understanding. As $|\mathbf{v}||\mathbf{w}|$ always is non-negative, the sign of the dot product is only dependent of the cosine function. It should be common knowledge that $\cos\theta \geq 0$ iff $\theta = \left[-\frac{1}{2}\pi..\frac{1}{2}\pi\right]$. If a vector $\mathbf{v}$ is perpendicular to a plane then the dot product between $\mathbf{v}$ and another vector $\mathbf{w}$ is negative, if $\mathbf{w}$ completely lies on the other half-space of the plane. Consider the synonymous situation in Figure 3.15. The dot product between a surface normal and a gradient of the object is negative when a collision has occurred on the left side, and positive on the right side of the object.

The second dependency is regarding the penalty coefficient s. Consider (3.6.4), as the exponential function is always positive, the only way $s < 0$ is iff $\varepsilon < 0$. We know from part 1 that if the implicit function is negative inside its surface, ε must be positive and thus $s > 0$. As expected, if f is positive inside the surface, ε must be negative and thus $s < 0$.

Part 3

Up to this point, we have shown the behavior of the penalty contraction p in

$$\mathbf{f}_{collision} = -p\mathbf{n}\,, \tag{3.6.6}$$

where $p = s\nabla f\cdot\mathbf{n}$. From part 1 and part 2 we can generate the following Table 3.2.
Let us take a final look at (3.6.3) using (3.6.6) instead. The penalty contraction p is multiplied by the surface normal $\mathbf{n}$, and the resulting vector is negated. Due to Table 3.2 it should be clear that the result of multiplying p by $\mathbf{n}$ always yields a force vector with direction away from the object, moving on the line defined by the surface normal. According to (3.6.6) $p\mathbf{n}$ must still be negated. By doing this, we will completely reverse the penalty force, pulling particles further into the object

	$f > 0$ (outside)	$f < 0$ (inside)
$\nabla f \cdot \mathbf{n} > 0$	$p < 0$	$p > 0$
$\nabla f \cdot \mathbf{n} < 0$	$p > 0$	$p < 0$

Table 3.2 The sign of the penalty contraction p, dependent of the implicit surface f

and the effect is self reinforcing, due to the exponential function in (3.6.1). We know that everything with a mass provides a force of attraction towards other objects with a mass, and so it would seem that this behavior is exactly like a local gravitational field, but as this is not the intension we must assume that this behavior is not only wrong, it is completely reversed. Removing the leading negation sign from (3.6.3), results in a correctly working penalty force as proven in part 1 through part 3

$$\mathbf{f}_{collision} = \left(\nabla f(\mathbf{r})\,\varepsilon^{-1}\exp\left(-\varepsilon^{-1}f(\mathbf{r})\right)\cdot\mathbf{n}\right)\mathbf{n}\,. \tag{3.6.7}$$

We know that (3.6.7) can no longer be recognized as the negative gradient of an energy potential, but instead of modifying both the potential and the insides of $\mathbf{f}_{collision}$ we find this proposed modification reasonable.

The shape of the potential constant, epsilon ε, describes the strength of the penalty. To get any visual effect of the penalty force, ε is typically chosen to be less than one, see Figure 3.14a and Figure 3.14b for a difference of the strength with varied values of ε. For more information about how to work with objects, consult 4.5.2 *Objects*.

3.6.1.3 Reduced aggressiveness

Providing (3.6.7) to our system will carry out a good job of keeping particles out from any impenetrable rigid object. However, we must use a very calm system to prevent particles from "leaving orbit", which could be the result, if penetration depths become large enough. Also if adjacent particles get too far away from each other, huge internal elastic energies will tear the system apart and because the elastic forces work like springs, see 3.2.1 *Intrinsic elastic force*, the system will diverge. To secure a more controlled collision penalty force, we must use a critically damped system. To make the problem more tiresome, a large damping factor on the particles will also make the simulation slow. The animation will be less attractive as everything seems to be running in slow motion. Our problem is the aggressiveness of the exponential function in (3.6.6). On the other hand the exponential function also provides us with a homogeneous method of handling collision.

We now present a quadratic version of the corrected penalty collision force, which has the same effect as using (3.6.7), but has a practicable reduced aggression

$$\mathbf{f}_{collision} = \begin{cases} \mathbf{0} & ,\ f(\mathbf{r}) \geq 0 \\ \left(\left(\varepsilon^{-1}f(\mathbf{r})\right)^2\nabla f(\mathbf{r})\cdot\mathbf{n}\right)\mathbf{n} & ,\ f(\mathbf{r}) < 0. \end{cases} \tag{3.6.8}$$

The only real difference between (3.6.7) and (3.6.8) is the penalty coefficient s. We have reduced the penalty coefficient from being exponential, $s = \varepsilon^{-1}\exp\left(-\varepsilon^{-1}f(\mathbf{r})\right)$, to being quadratic, $s = \left(\varepsilon^{-1}f(\mathbf{r})\right)^2$. The same rules as before naturally apply for s. The reason for squaring the penalty coefficient is equal to the effect of using the exponential function. We want the strength of the penalty force to be dependent of the actual penetration depth of the object. By doing this we now have control over how fast we wish to push the penetrating particles out from an object.

Epsilon, ε, is provided mostly due to backward compatibilities, but also to control the stiffness of the penalty spring. (3.6.8) insures that a user can force particles to penetrate objects without letting an deformable object be torn apart easily. But we can still do better to avoid unnecessary large penalties, especially if the penalty spring contribution is weak. If a particle has a large penetration depth, it might still be inside the object in the next time step. As the system identifies the penetrating particles, it will add yet another additional penalty force contribution. Thus the particles could end up far away from the object instead of just in the close proximity of the surface. A very simple way of securing only an initial penalty force that will be used on penetrating particles is to observe the direction of the particle velocities. If the particle velocity has the same direction as the gradient, it is already leaving the object, and we should ignore adding another penalty contribution. We now extend equation (3.6.8) to

$$\mathbf{f}_{collision} = \begin{cases} \mathbf{0} & , f(\mathbf{r}) \geq 0 \vee \nabla f \cdot \mathbf{v} > 0 \\ \left(\left(\varepsilon^{-1} f(\mathbf{r})\right)^2 \nabla f(\mathbf{r}) \cdot \mathbf{n}\right)\mathbf{n} & , f(\mathbf{r}) < 0 \end{cases} \tag{3.6.9}$$

where $\mathbf{v}$ is the particle velocity. When particles are close to equilibrium they might oscillate heavily, as the velocity is reversed constantly. This tends to get particles a little unstable and could affect visualization. This small issue can be solved using a control penetration threshold. If the penetration is below this threshold only then should we take the velocity directions into account. The penetration threshold constant must be chosen individually, such that it provides the best possible visual result. Using (3.6.9) without the threshold will not affect the outcome of the physical simulation. The procedure is illustrated in Listing 3.4.

```
if (f(obj, r[par]) >= 0)
  return 0
if (f(obj, r[par]) < PENETRATION_THRESHOLD)
  if (grad(obj, r[par] * v[par]) > 0)
    return 0
// continue with (3.6.9)...
```

Listing 3.4 Pseudo code illustrating the penetration threshold procedure, for some particle `par` and some object `obj`

3.6.2 Projection

The velocity of a particle, in the integration method, used throughout [21], is estimated as the difference between positions during one time step,

$$\mathbf{v}_t = \frac{\mathbf{r}_t - \mathbf{r}_{t-\Delta t}}{\Delta t}. \tag{3.6.10}$$

As the velocity is implicitly given, it can be utilized for in yet another way of collision handling; collision by projection. A projection method is fairly simple. It is a matter of updating particle positions for particles that have penetrated an object, and projecting them out of the object and onto the surface. Collision handling by projection is manageable, without any greater concern, only because velocities are calculated implicitly. It is hard for velocity and position to come out of sync. If the velocities were calculated explicitly, this projection method would not be that easy.

Updating particle positions after penetration will not affect the system anymore than letting the particle positions being updated by the integration solver, as long as new estimated velocities are calculated. This scheme is equivalent with the Verlet integration method [24], which is used intensively to simulate molecular dynamics, but works very well in practice for simulating particle dynamics, e.g. in computer games, see [13].

We will still use the different parts presented throughout the collision section for the projection method. We assume that no penalty collision force has been applied to the sum of all external forces for all particles. After all particle positions have been updated, it is now a simple matter of going through all particles and project particles that penetrate the impenetrable objects, out on

the surface boundaries. We will project a particle along the direction of the gradient evaluated at its position. The projection length is proportional to the distance from the particle to the surface along the unit gradient. As we are unable to guarantee that the value of an implicit function $f(\mathbf{p})$ will return the signed distance from $\mathbf{p}$ to the object surface, we will instead use a distance function that is required to return the unsigned distance from any penetrating point $\mathbf{p}$ to the object surface. Using some arbitrary constant $c > 0$, we can control the strength of the projection. This constant can be chosen to work as a material property of the object, e.g. a restitution coefficient to simulate a surface of rubber, or a damping factor to simulate the surface of water. The discussed projection algorithm is illustrated in the following Listing 3.5.

```
for (obj = 0; obj < #objects; obj++)
  for (par = 0; par < #particles; par++)
    if (f(obj, r[par]) < 0)  // only project penetrating particles
      r[par] += c(obj) * d(obj, r[par]) * norm(grad(obj, r[par]));
```

Listing 3.5 Pseudo code illustrating the projection algorithm

It is rarely a flawless world we end up simulating. Even though the projection method is simple it should be used with care. The implicit given velocities might not always be physical accurate, as energies tend to leave the system. The projection will increase the potential energy of the particles as they are unnaturally lifted opposite the gravitational field. If the surface is not perpendicular to the gravitational force, the potential energy $U = mgh$ will transfer into kinetic energy, $K = \frac{1}{2}mv^2$, and thus increase the velocity of the particles

$$\begin{aligned} U &= K \\ mgh &= \frac{1}{2}mv^2 \\ 2gh &= v^2 \\ v &= \sqrt{2gh} \end{aligned} \tag{3.6.11}$$

where g is the gravitational field, v is the particle velocity, m is the mass of the particle, and h is the penetration depth in the direction of the gravitational force. However, as we are more interested in being able to animate deformable models convincingly, rather then securing a physical correct system at all times, this inaccuracy is acceptable.

One advantage using the projection method is to completely avoid visual particle penetrations of the impenetrable rigid objects. None of the penalty methods can guarantee this, especially upon user interaction.

3.6.3 Self-intersection

Compared to rigid bodies a whole new collision aspect exists when working with deformable bodies. Elastic bodies should not self-intersect as they deform. In [21] Terzopoulos et al. propose to surround the surface of the deformable object with a self-repulsive penalty collision force. This is equivalent to the penalty method surrounding the impenetrable rigid objects described in the previous section. Unfortunately the repulsive force also requires an implicit description of the deformable object, which can only be determined locally using the grid structure. Introducing a virtually thin repulsive force field, around each surface triangle, it can then act like a shield rejecting penetrating points. The number of repulsive triangle force fields is proportional to the number of surface particles, say n. Thus, a naive approach for handling self-intersection is extremely expensive, as all particles should check each triangle force field. We know from [4] that any triangulation of P has $2n - 2 - k$ triangles, where P is a set of n points in the plane, and k is the

number of points that lie on the convex hull of P. As the number of triangles is asymptotically bound to the number of points, the amount of detection checks is bound by $O(n^2)$. Placing the triangle force fields into hierarchical bounding boxes can decrease the amount of detection checks, and thus increase the total running time for each simulation step, if self-intersection is applied.

3.6.4 Geometric collision detection

By using hierarchical bounding boxes, we have now stepped into the field of geometric collision detection. A lot of efforts have been put into efficient collision detection between rigid bodies, but the area of collision detection between deformable bodies is still to be researched. Redon et al. have in [18] implemented a fast collision detection algorithm for continuous rigid bodies. As presented in the previous section, a typical issue about collision detection for continuous deformable bodies, is the detection of self-intersection. Further more the problems of updating hierarchical bounding boxes very fast during deformation, and the possibility to retrieve any penetration depths, are both important issues when choosing collision detection algorithms for deformable bodies.

Any collision detection can be split into several phases, typically into a broad phase and a narrow phase. Van den Bergen proposed in 1998 an efficient broad phase collision detection method for complex deformable models using AABB trees [23]. Later in 2001 Larsson and Akenine-Möller introduced a hybrid bounding box method for deformable bodies, which guarantee efficient tree updates during simulation [15]. A fast triangle-triangle intersection test was introduced by Möller in 1997, which can be used with advantage for handling self-intersection and in any narrow phase during normal collision detection [16].

Collision detection between multiple deformable objects, along with collision between complex rigid bodies and deformable objects, should also be possible for complete dynamic simulations using deformable bodies, but requires techniques referenced to during this section.

3.6.5 Using collision handling

We have implemented collision handling using both penalty and the projection methods. The 2 penalty methods introduced in 3.6.1 *Penalty collision force* are available. That is the corrected exponential version of $\mathbf{f}_{collision}$ (3.6.7), and our own quadratic version (3.6.9). The projection method discussed in the previous subsection is also available. All 3 methods of collision handling can be activated individually during compile time. Please consult 4.2.2 *User controlled behavior* for details about user controlled defines.

3.7 Surface Normals

In [21] normals are used for the second fundamental form of a surface, and for several external local forces, but it was never discussed how they are obtained. In the following we will discuss how surface normals are used within our working areas for the different kinds of manifolds. We will go into details about how they can be determined in practice.

To simulate deformable bodies as described in [21] we will need the surface normals on the bodies on several occasions. These include:

- Surface curvature for the second fundamental form, see 2.2.5 *Second fundamental form*
- Local external forces, see 3.5 *External Forces*
- Collision handling, see 3.6 *Collision*
- Visualization, see 3.9 *Visualization*

3.7.1 Surfaces

The bodies we are using as deformable reference surfaces, are primarily bivariate parametric patches, e.g. Hermite patches, Bézier patches, and B-Spline surface patches, etc. [9, 25]. A surface patch is generically given by

$$\mathbf{Q}(s,t) \in \mathbb{R}^3, \quad 0 \leq s,t \leq 1\,, \tag{3.7.1}$$

with the s-directional tangent at some $[s,t]$, is given by the first order derivative in s

$$\mathbf{Q}_s{}'(s,t) = \frac{\partial \mathbf{Q}}{\partial s}(s,t) \tag{3.7.2}$$

and equally the t-directional tangent is given by

$$\mathbf{Q}_t{}'(s,t) = \frac{\partial \mathbf{Q}}{\partial t}(s,t)\,. \tag{3.7.3}$$

The two vector derivatives $\frac{\partial \mathbf{Q}}{\partial s}$ and $\frac{\partial \mathbf{Q}}{\partial t}$, span the tangent plane at some point $\mathbf{Q}(s,t)$ on the patch. The surface normal is perpendicular to the tangent plane, thus it is directly given by

$$\mathbf{n}(s,t) = \frac{\frac{\partial \mathbf{Q}}{\partial s} \times \frac{\partial \mathbf{Q}}{\partial t}}{\left|\frac{\partial \mathbf{Q}}{\partial s} \times \frac{\partial \mathbf{Q}}{\partial t}\right|}\,. \tag{3.7.4}$$

As the cross product is not commutative, that is

$$\frac{\partial \mathbf{Q}}{\partial s} \times \frac{\partial \mathbf{Q}}{\partial t} = -\frac{\partial \mathbf{Q}}{\partial t} \times \frac{\partial \mathbf{Q}}{\partial s}\,, \tag{3.7.5}$$

care should be taken not to mix the components of the vector cross product. It is not significant how the surface normals are calculated, as long as they are all computed by use of the same method. In (3.7.4) we divide by the magnitude to obtain a unit surface normal.

While the above method for calculating surface normals on the patch is analytically correct, it only makes sense to do so while the surface has not been deformed yet, that is, while the patch is still used to reference the natural or initial shape of the deformable model. Once the surface has deformed we cannot use its analytic reference shapes any longer. This is due to the fact that the reference shapes are not deformed, and even if they were, the particles would still be able to move freely around in space within their bounded constraints, resulting in surface deformations that the analytic model of the patch most likely cannot describe. This means something else is needed. The deformable surface normals can instead be estimated using derivative approximation by finite differences, as we have presented in 2.4 *Finite Differences*. We are able to approximate the two partial derivatives by any finite difference method, thus we can still use (3.7.4) to calculate an estimated surface normal.

To deal with boundary issues, see 3.4.1 *Projection method*, and to gain a higher order error term we will use a central difference approximation to determine our partial derivatives

$$\frac{\partial \mathbf{Q}}{\partial s} \approx \frac{\mathbf{Q}(s+h,t) - \mathbf{Q}(s-h,t)}{2h}$$

and (3.7.6)

$$\frac{\partial \mathbf{Q}}{\partial t} \approx \frac{\mathbf{Q}(s,t+h) - \mathbf{Q}(s,t-h)}{2h}.$$

The above approximations only apply to the analytic surface patch. We still need the deformable normals, so we will use the same discretization method that we have described in 3.1 *Discretization*. To approximate the above equations (3.7.6) we will use the two first-order central difference operators (3.1.13). As the generic grid function variable $\mathbf{u}[m,n]$, we are going to use the discrete position equation $\mathbf{r}[m,n]$ for the surface. The approximations of the partial derivates in (3.7.6) will become

$$\begin{aligned} D_1[m,n] &= \frac{\mathbf{r}[m+1,n]-\mathbf{r}[m-1,n]}{2h_1} \\ D_2[m,n] &= \frac{\mathbf{r}[m,n+1]-\mathbf{r}[m,n-1]}{2h_2}, \end{aligned} \tag{3.7.7}$$

and the discrete surface normal can now be computed by

$$\mathbf{n}[m,n] = \frac{D_1 \times D_2}{|D_1 \times D_2|}. \tag{3.7.8}$$

To gain performance we could easily remove both divisions in D_1 and D_2 without any problems. This is of course due to the fact that we would normalize $\mathbf{n}[m,n]$ anyway.

3.7.2 Solids

We can easily generalize the theory of how to provide normals for deformable surfaces to deformable solids. The grid that represents the descretization of deformable solids is now in 3 dimensions. Using the shape of the grid directly, the most trivial shape of such a solid is a cube. As a cube has 6 surfaces, the normals of each of these surfaces can be determined as described above.

Using the discrete position equation for solids, $\mathbf{r}[l,m,n]$, the partial derivates can be approximated by

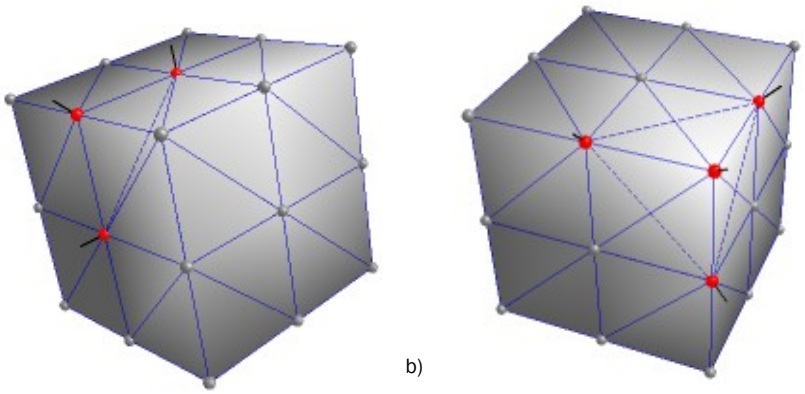
$$\begin{aligned} D_1[l,m,n] &= \frac{\mathbf{r}[l+1,m,n]-\mathbf{r}[l-1,m,n]}{2h_1} \\ D_2[l,m,n] &= \frac{\mathbf{r}[l,m+1,n]-\mathbf{r}[l,m-1,n]}{2h_2} \\ D_3[l,m,n] &= \frac{\mathbf{r}[l,m,n+1]-\mathbf{r}[l,m,n-1]}{2h_3}. \end{aligned} \tag{3.7.9}$$

At least 6 different calculations must be used to determine the discrete surface normals for each side of the solid. In no particular order, the normals of the 6 sides are given by

$$\mathbf{n_1}[l,m,n] = \frac{D_1 \times D_2}{|D_1 \times D_2|}, \tag{3.7.10}$$

$$\mathbf{n_2}[l,m,n] = \frac{D_2 \times D_1}{|D_2 \times D_1|} = -\mathbf{n_1}[l,m,n], \tag{3.7.11}$$

$$\mathbf{n_3}[l,m,n] = \frac{D_1 \times D_3}{|D_1 \times D_3|}, \tag{3.7.12}$$



a) b)

Figure 3.16 Examples of surface normals on a solid **a)** On an edge e **b)** In a corner c

$$\mathbf{n_4}[l,m,n] = \frac{D_3 \times D_1}{|D_3 \times D_1|} = -\mathbf{n_3}[l,m,n], \tag{3.7.13}$$

$$\mathbf{n_5}[l,m,n] = \frac{D_2 \times D_3}{|D_2 \times D_3|}, \tag{3.7.14}$$

and

$$\mathbf{n_6}[l,m,n] = \frac{D_3 \times D_2}{|D_3 \times D_2|} = -\mathbf{n_5}[l,m,n]. \tag{3.7.15}$$

Care should be taken when focusing on the edges and corners. Each edge is shared between 2 faces, and each corner is adjacent to 3 faces.

As an example using (3.7.10) through (3.7.15), an edge e is shared between face 1 and face 3. Each normal for all the particles on edge e other than the corners, can be estimated by normalizing the result that is gained by adding the 2 estimated normals from each side

$$\mathbf{n}_e[l,m,n] = \frac{D_1 \times D_2 + D_1 \times D_3}{|D_1 \times D_2 + D_1 \times D_3|} = \frac{\mathbf{n}_1 + \mathbf{n}_3}{|\mathbf{n}_1 + \mathbf{n}_3|}, \tag{3.7.16}$$

and equivalently if a corner c's surface normal is adjacent to face 1, face 3, and face 5, it can be estimated by

$$\begin{aligned} \mathbf{n}_c[l,m,n] &= \frac{D_1 \times D_2 + D_1 \times D_3 + D_2 \times D_3}{|D_1 \times D_2 + D_1 \times D_3 + D_2 \times D_3|} \\ &= \frac{\mathbf{n}_1 + \mathbf{n}_3 + \mathbf{n}_5}{|\mathbf{n}_1 + \mathbf{n}_3 + \mathbf{n}_5|}. \end{aligned} \tag{3.7.17}$$

The two examples are depicted in Figure 3.16, where Figure 3.16a illustrates equation (3.7.16) and Figure 3.16b illustrates equation (3.7.17).

What about all the internal particles within a solid? Well, as in the real world the molecules and atoms of an intact solid never come in contact with the outside world. The only external force the internal particles of a solid are influenced by, is gravity. All other external forces only influence

particles on a surface. In other words, the sum of all external forces, not including $\mathbf{f}_{gravity}$, should be zero for all particles not on a surface. In practice no special problems concerning the normals needs to be taken care of, as the surface normal of every internal particle simply could be set to $\mathbf{0}$

$$\mathbf{n}[l,m,n] = \mathbf{0}, \quad 0 < l < L-1, \quad 0 < m < M-1, \quad 0 < n < N-1. \tag{3.7.18}$$

This further more strengthens the notion that there is no such thing as a surface normal attached to an internal particle within a solid. External forces that only affect surface particles, usually include the normal in its equation, as described in 3.5 *External Forces*. It is fairly easy to verify that such an external force contributes nothing, if the surface normal is zero.

Of course to speed up performance, a simple check whether or not a particle's surface normal is different from zero could be included, such that the force contribution is only computed if it will make a difference.

3.8 Numerical Integration

To simulate the dynamics for elastically deformable bodies, the system of non-linear partial differential equations (2.1.1) have been reduced into a system of linear ordinary differential equations

$$\mathbf{M}\frac{\partial^2 \underline{\mathbf{r}}}{\partial t^2} + \mathbf{C}\frac{\partial \underline{\mathbf{r}}}{\partial t} + \mathbf{K}(\underline{\mathbf{r}})\,\underline{\mathbf{r}} = \underline{\mathbf{f}}\,. \tag{3.8.1}$$

We will integrate (3.8.1) through time with equal time steps Δt. Different integration schemes can be chosen for different behaviors of the system. It all depends on at which time level we evaluate the different terms of (3.8.1). Time t is the current time, where $t-\Delta t$ and $t+\Delta t$ are the time just before and the time just ahead, respectively. Before we continue we will give a brief survey on some possible integration schemes.

3.8.1 Integration schemes

Consider the single differential equation

$$y' = -cy\,, \tag{3.8.2}$$

where $c > 0$ is a constant. The explicit Euler scheme for integrating (3.8.2) using step size h is

$$\begin{aligned} y_{n+1} &= y_n + hy'_n \\ &= y_n + h(-cy_n) \\ &= y_n - chy_n \\ &= (1-ch)\,y_n. \end{aligned} \tag{3.8.3}$$

The iterative method is called *explicit* due to the fact that the new value y_{n+1} on the left hand side, is computed explicitly in terms of the old value y_n on the right. One problem with this explicit scheme is the stability of the solution. If $h > \frac{2}{c}$ in (3.8.3) then it should be obvious that $|y_n| \to \infty$ as $n \to \infty$.

The cure for the above instability problem, is use an implicit Euler scheme instead. We will let the right hand side be evaluated in the new value y_{n+1}

$$\begin{aligned} y_{n+1} &= y_n + hy'_{n+1} \\ &= y_n + h(-cy_{n+1}) \\ &= y_n - chy_{n+1} \\ &\Updownarrow \\ y_n &= y_{n+1} + chy_{n+1} \\ &= (1+ch)\, y_{n+1} \\ &\Updownarrow \\ y_{n+1} &= \frac{y_n}{1+ch}. \end{aligned} \tag{3.8.4}$$

This *implicit* method is completely stable. Even when $h \to \infty$ then $y_{n+1} \to 0$, which is in fact the correct solution to the differential equation [17]. Equation (3.8.2) can easily be generalized to sets of linear equations

$$\mathbf{y}' = -\mathbf{C} \cdot \mathbf{y}\,, \tag{3.8.5}$$

where $\mathbf{C}$ is a positive definite (PD) matrix. The explicit Euler differencing in (3.8.3) then becomes

$$\mathbf{y}_{n+1} = (\mathbf{I} - \mathbf{C}h) \cdot \mathbf{y}_n\,, \tag{3.8.6}$$

where $\mathbf{I}$ is the identify matrix with the same size as $\mathbf{C}$. The implicit scheme in (3.8.4) becomes

$$\begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + h\mathbf{y}'_{n+1} \\ &= \mathbf{y}_n + h\big(-\mathbf{C} \cdot \mathbf{y}_{n+1}\big) \\ &= \mathbf{y}_n - \mathbf{C}h \cdot \mathbf{y}_{n+1} \\ &\Updownarrow \\ \mathbf{y}_n &= \mathbf{y}_{n+1} + \mathbf{C}h \cdot \mathbf{y}_{n+1} \\ &= (\mathbf{I} + \mathbf{C}h) \cdot \mathbf{y}_{n+1} \\ &\Updownarrow \\ \mathbf{y}_{n+1} &= (\mathbf{I} + \mathbf{C}h)^{-1} \cdot \mathbf{y}_n. \end{aligned} \tag{3.8.7}$$

Like with the implicit scheme for a single differential equation, (3.8.7) is also stable for all step sizes h, but we might be giving up accuracy if h is too large. Care should likewise be taken when choosing h for (3.8.6), as only small h will result in a stable solution. Why not just always use the implicit integration scheme? It is both stable and we can choose a step size that fits our needs for a suitable animation, but the penalty we must pay when using the implicit integration method, is that we are now required to invert a matrix for each iteration.

Equation (3.8.7) holds only for linear systems. Unfortunately not all equations are linear with constant coefficients. Consider the non-linear differential system

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}) \tag{3.8.8}$$

and using implicit differencing yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(\mathbf{y}_{n+1}) . \tag{3.8.9}$$

All is not lost. We can use a technique called linearization to obtain a system of linear equations. The equations (3.8.9) can be linearized, using Newton's method, into

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left(\mathbf{I} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right)^{-1} \cdot \mathbf{f}\left(\mathbf{y}_n\right), \tag{3.8.10}$$

where the matrix $\mathbf{I} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}$ must be inverted at each step. If h is not chosen too big, one iteration of (3.8.10) may be accurate enough to solve (3.8.8). Solving an implicit method by linearization is called a *semi-implicit* method [17]. Semi-implicit methods are not guaranteed to be stable, but they usually are. However, a semi-implicit integration method is still more stable and robust then an explicit method.

In the next two subsections we present the original semi-implicit integration method from [21], and also show how a simple explicit Euler integration method can be obtained.

3.8.2 Semi-implicit integration

Terzopoulos et al. [21] use a semi-implicit integration solver for the system of ODEs (3.8.1). The paper pioneered the physics-based modeling and animation paradigm in computer graphics, primarily due to the fact that it was one of the first to describe an implicit method for this kind of problem.

The second-order accurate partial differential equations with respect to time from (3.8.1), can be approximated by the discrete central differences

$$\frac{\partial^2 \underline{\mathbf{r}}}{\partial t^2} \approx \frac{\underline{\mathbf{r}}_{t+\Delta t} - 2\underline{\mathbf{r}}_t + \underline{\mathbf{r}}_{t-\Delta t}}{\Delta t^2}$$

and (3.8.11)

$$\frac{\partial \underline{\mathbf{r}}}{\partial t} \approx \frac{\underline{\mathbf{r}}_{t+\Delta t} - \underline{\mathbf{r}}_{t-\Delta t}}{2\Delta t}.$$

The implicit integration procedure appears when we evaluate $\underline{\mathbf{e}}$ (3.3.1) at time $t + \Delta t$. The external force vector is evaluated at time t. The matrices $\mathbf{M}$ and $\mathbf{C}$ are constants, hence not dependant of t.
Substituting (3.8.11) into (3.8.1) and rearranging yields

$$\begin{aligned} \mathbf{M}\left(\frac{\underline{\mathbf{r}}_{t+\Delta t} - 2\underline{\mathbf{r}}_t + \underline{\mathbf{r}}_{t-\Delta t}}{\Delta t^2}\right) & \\ +\mathbf{C}\left(\frac{\underline{\mathbf{r}}_{t+\Delta t} - \underline{\mathbf{r}}_{t-\Delta t}}{2\Delta t}\right) + \mathbf{K}\left(\underline{\mathbf{r}}_t\right)\underline{\mathbf{r}}_{t+\Delta t} &= \underline{\mathbf{f}}_t \\ \left(\mathbf{M}\left(\frac{1}{\Delta t^2}\right) + \mathbf{C}\left(\frac{1}{2\Delta t}\right) + \mathbf{K}\left(\underline{\mathbf{r}}_t\right)\right)\underline{\mathbf{r}}_{t+\Delta t} & \\ -\left(\mathbf{M}\left(\frac{2}{\Delta t^2}\right)\right)\underline{\mathbf{r}}_t + \left(\mathbf{M}\left(\frac{1}{\Delta t^2}\right) - \mathbf{C}\left(\frac{1}{2\Delta t}\right)\right)\underline{\mathbf{r}}_{t-\Delta t} &= \underline{\mathbf{f}}_t. \end{aligned} \tag{3.8.12}$$

Adding and subtracting the same product contributes nothing, thus

$$\begin{aligned}
\left(\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right)+\mathbf{K}(\underline{\mathbf{r}}_t)\right)\underline{\mathbf{r}}_{t+\Delta t} & \\
-\left(\mathbf{M}\left(\frac{2}{\Delta t^2}+\mathbf{C}\left(\frac{1}{2\Delta t}\right)-\mathbf{C}\left(\frac{1}{2\Delta t}\right)\right)\right)\underline{\mathbf{r}}_t & \\
+\left(\mathbf{M}\left(\frac{1}{\Delta t^2}\right)-\mathbf{C}\left(\frac{1}{2\Delta t}\right)\right)\underline{\mathbf{r}}_{t-\Delta t} &= \underline{\mathbf{f}}_t \\
\left(\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right)+\mathbf{K}(\underline{\mathbf{r}}_t)\right)\underline{\mathbf{r}}_{t+\Delta t} & \\
-\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right)\underline{\mathbf{r}}_t & \\
-\left(\frac{1}{\Delta t^2}\mathbf{M}-\frac{1}{2\Delta t}\mathbf{C}\right)\underline{\mathbf{r}}_t-\left(\frac{1}{\Delta t^2}\mathbf{M}-\frac{1}{2\Delta t}\mathbf{C}\right)\underline{\mathbf{r}}_{t-\Delta t} &= \underline{\mathbf{f}}_t.
\end{aligned} \tag{3.8.13}$$

Recognizing the last two terms in (3.8.13) as the velocity at time t, we obtain

$$\begin{aligned}
\left(\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right)+\mathbf{K}(\underline{\mathbf{r}}_t)\right)\underline{\mathbf{r}}_{t+\Delta t} & \\
-\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right)\underline{\mathbf{r}}_t-\left(\frac{1}{\Delta t}\mathbf{M}-\frac{1}{2}\mathbf{C}\right)\underline{\mathbf{v}}_t &= \underline{\mathbf{f}}_t,
\end{aligned} \tag{3.8.14}$$

where

$$\underline{\mathbf{v}}_t=\frac{\underline{\mathbf{r}}_t-\underline{\mathbf{r}}_{t-\Delta t}}{\Delta t}. \tag{3.8.15}$$

Rearranging and substituting yields

$$\mathbf{A}_t\underline{\mathbf{r}}_{t+\Delta t}=\underline{\mathbf{g}}_t\,, \tag{3.8.16}$$

where the system matrix

$$\mathbf{A}_t=\mathbf{K}(\underline{\mathbf{r}}_t)+\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right), \tag{3.8.17}$$

and the effective force vector

$$\underline{\mathbf{g}}_t=\underline{\mathbf{f}}_t+\left(\frac{1}{\Delta t^2}\mathbf{M}+\frac{1}{2\Delta t}\mathbf{C}\right)\underline{\mathbf{r}}_t+\left(\frac{1}{\Delta t}\mathbf{M}-\frac{1}{2}\mathbf{C}\right)\underline{\mathbf{v}}_t\,. \tag{3.8.18}$$

Multiplying by the inverse system matrix $\mathbf{A}_t^{-1}$ at both sides of (3.8.16) leads to the desired result

$$\begin{aligned}
\mathbf{A}_t^{-1}\mathbf{A}_t\underline{\mathbf{r}}_{t+\Delta t} &= \mathbf{A}_t^{-1}\underline{\mathbf{g}}_t \\
\underline{\mathbf{r}}_{t+\Delta t} &= \mathbf{A}_t^{-1}\underline{\mathbf{g}}_t
\end{aligned} \tag{3.8.19}$$

Using (3.8.19) it is now only a matter of inverting a matrix and multiplying it with a vector, but it is not as simple as it might seem. Hidden in the system matrix $\mathbf{A}_t$ evaluated at time t, is the stiffness matrix $\mathbf{K}(\mathbf{r}_t)$, which is a rather tough one to assemble, see 3.3 *The Stiffness Matrix*. The semi-implicit procedure is illustrated below in Listing 3.6 for the case for discrete surfaces.

```
const scalar dt = ...  // the time step
// calculate all the new particle positions
Vector<Vector3> new_r[M*N], g[M*N];  // all MN entries are 3d-vectors
// fill in A and g
Matrix<scalar> A[M*N,M*N] = stiffness(alpha,beta);  // A = K
for (n = 0; n < N; n++)
  for (m = 0; m < M; m++) {
    const int entry = n*M+m;
    const Vector3 f = externalForces(m,n);
    const scalar tmp1 = (1/(dt*dt))*μ[m,n] + (1/(2*dt))*γ[m,n];
    const scalar tmp2 = (1/dt)*μ[m,n] - 0.5*γ[m,n];
    A[entry, entry] += tmp1;  // fill in the rest of (3.8.17)
    g[entry] = f + tmp1*r[m,n] + tmp2*v[m,n]; // fill regarding (3.8.18)
  }
// solve the system of linear equations
new_r = A.inverse()*g;
// calculate new velocities and replace old positions with the new ones
for (n = 0; n < N; n++)
  for (m = 0; m < M; m++) {
    v[m,n] = (1/dt)*(new_r[m,n]-r[m,n]);
    r[m,n] = new_r[m,n];
  }
```

Listing 3.6 Pseudo code illustrating the semi-implicit procedure

The pseudo code in Listing 3.6 assumes that all $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ tensors have been pre-computed and stored in `alpha` and `beta`. Listing 3.6 also assumes that matrices and vectors can be multiplied together, even though the dimensions differ. The vectors `new_r` and `g` have the same size as a column of the system matrix `A`, but has scalars as entries opposite 3d-vectors. In practice one would likely do something else, for instances expanding both vectors and matrices 3 times and using scalars for each entry, etc. Listing 3.6 also omits both combined mass $\mathbf{M}$ and damping $\mathbf{C}$ matrices, and adds the result directly, into the diagonal of the system matrix and the effective force vector field, thus saving some memory. The trivial constants in Listing 3.6, i.e. $\frac{1}{\Delta t^2}$, $\frac{1}{2\Delta t}$, and $\frac{1}{\Delta t}$, of course can be pre-calculated prior to the first double for-loop, hence reducing the amount of divisions from $4\mathrm{MN}$ to 4 for each simulation step. If the time step Δt is not allowed to be changed once the simulation is running, the four constants only need to be computed once prior to simulation start.

Several strategies can be chosen to speed up performance, especially considering matrix operations. The largest performance boost can be gained choosing a tailored solution to solve the linear systems (3.8.19). We know from 3.3 *The Stiffness Matrix* that $\mathbf{K}(\mathbf{r}_t)$ has some nice properties. The matrix sum in (3.8.17) still posses the same properties as the stiffness matrix, as we are only adding a positive diagonal matrix. Choosing a solution method such as Choleski decomposition or LU decomposition is very effective. If the chosen linear equations solver has to be implemented manually, the internet is a great resource along with [17].

Originally we have based our implementation of the linear equations solver on a standard matrix inverse operation using LU-decomposition [6], but we have also extended our implementation to use the 3rd party ATLAS (Automatically Tuned Linear Algebra Software) library [1]. ATLAS increased the performance a great deal, enabling us to use a much higher amount of particles in the simulations. The performance boost need not stop here. The final attempt to increase the total amount of particles in the realtime simulation, introduced us to the conjugate gradient method, which can be incredibly fast. Next we will discuss this method in more detail.

Please refer to 4.2 *Compiling and Building* on how to build our implementation using 3rd party software packages, and how to control which solution to be used.

3.8.2.1 Conjugate gradient method

The deformable model has been reduced to a system of linear equations that needs to be solved in each time step. The general linear system of equations has the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,, \tag{3.8.20}$$

where $\mathbf{A}$ is the coefficient matrix of the solution vector $\mathbf{x}$ and $\mathbf{b}$ is a vector with the same size as $\mathbf{x}$. The main linear system of equations (3.8.16) takes the same form as (3.8.20). At first the linear system (3.8.19) was solved using the LU-decomposition method, but we soon discovered that this method did not take advantage of the nice computational properties of the stiffness matrix $\mathbf{K}$, which is the main contributor to the system matrix $\mathbf{A}_t$. The properties are sparseness, bandedness, symmetrical, and positive definiteness. A method that works well under these properties is the conjugate gradient method [12]. This is an iterative method that seeks to minimize the quadratic function

$$\varphi(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}\,. \tag{3.8.21}$$

This function does in fact take on its minimum when $\mathbf{A}\mathbf{x} = \mathbf{b}$ because

$$\varphi'(\mathbf{x}) = 0 \Leftrightarrow \mathbf{A}\mathbf{x} - \mathbf{b} = 0\,. \tag{3.8.22}$$

Another property of the quadratic function is that the negative gradient is the residual vector $\mathbf{r}$

$$-\nabla\varphi(\mathbf{x}) = \mathbf{b} - \mathbf{A}\mathbf{x} = \mathbf{r}\,. \tag{3.8.23}$$

The method starts out with some initial guess $\mathbf{x}_0$, and then it finds some search direction $\mathbf{s}_0$ and moves in this direction by α. This procedure is performed for each iteration, which means that

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\,\mathbf{s}_k\,. \tag{3.8.24}$$

To find the optimal value for α, we want to search along the line in (3.8.24) to find the point where (3.8.21) is minimal. We start by differentiating $\varphi\left(\mathbf{x}_{k+1}\right)$ with respect to α equal to zero

$$\begin{aligned} 0 = \frac{d}{d\alpha}\varphi\left(\mathbf{x}_{k+1}\right) = \nabla\varphi\left(\mathbf{x}_{k+1}\right)^T \frac{d}{d\alpha}\mathbf{x}_{k+1} = \\ \left(\mathbf{A}\mathbf{x}_{k+1} - \mathbf{b}\right)^T \frac{d}{d\alpha}\left(\mathbf{x}_k + \alpha\,\mathbf{s}_k\right) = -\mathbf{r}_{k+1}^T\mathbf{s}_k\,, \end{aligned} \tag{3.8.25}$$

and to find α, the new residual $\mathbf{r}_{k+1}$ is expressed in terms of the old residual

$$\mathbf{r}_{k+1} = \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1} = \mathbf{b} - \mathbf{A}\left(\mathbf{x}_k + \alpha\,\mathbf{s}_k\right) = \left(\mathbf{b} - \mathbf{A}\mathbf{x}_k\right) - \alpha\mathbf{A}\mathbf{s}_k = \mathbf{r}_k - \alpha\mathbf{A}\mathbf{s}_k\,. \tag{3.8.26}$$

The optimal value for α emerges by inserting (3.8.26) into (3.8.25)

$$0 = -\left(\mathbf{r}_k - \alpha\mathbf{A}\mathbf{s}_k\right)^T\mathbf{s}_k \Leftrightarrow \alpha = \frac{\mathbf{r}_k^T\mathbf{s}_k}{\mathbf{s}_k^T\mathbf{A}\mathbf{s}_k}\,. \tag{3.8.27}$$

Now all that is missing is to compute the new search direction for the next step $\mathbf{s}_{k+1}$. This is where the method earns its name, because we want all the search directions $\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, \ldots$ to be mutually $\mathbf{A}$-orthogonal or conjugate, as it is also called. For two vectors $\mathbf{s}_i$ and $\mathbf{s}_j$ to be $\mathbf{A}$-orthogonal the following must hold

$$\mathbf{s}_i^T \mathbf{A} \mathbf{s}_j = 0 \,. \tag{3.8.28}$$

The new search direction is computed as

$$\mathbf{s}_{k+1} = \mathbf{r}_{k+1} + \beta \, \mathbf{s}_k \,, \tag{3.8.29}$$

where the calculation of the β value is chosen, such that the conjugate property is preserved. The algorithm for the conjugate gradient method is showed in Listing 3.7.

```
x_0 = <initial guess>
r_0 = b - A * x_0 ;
s_0 = r_0 ;
for (k = 0; k < MAX_ITERATIONS; k++) {
   α_k = r_k^T * s_k / s_k^T * A * s_k ;   // compute search distance
   x_{k+1} = x_k + α_k * s_k ;             // update solution
   r_{k+1} = r_k - α_k * A * s_k ;         // update residual
   β_{k+1} = r_{k+1}^T * r_{k+1} / r_k^T * r_k ;
   s_{k+1} = r_k + β_{k+1} * s_k ;         // compute new search direction
}
```

Listing 3.7 Pseudo code illustrating the conjugate gradient method

Because of the conjugate property of the search directions, the exact solution to a $n \times n$ system can be computed in at most n iterations, although the method usually converges towards a good solution much faster. This is why a threshold of error is usually tolerated to speed up the algorithm. This threshold is held against the squared vector norm of the newly calculated residual vector, such that the algorithm stops if

$$\left| \mathbf{r}_{k+1} \right|_2^2 < threshold \,. \tag{3.8.30}$$

The algorithm could also stop before n iterations, simply by respecting a maximum number of iterations. Both the maximum amount of iterations and the accepted error tolerance can be specified in the implementation, see 4.2.2 *User controlled behavior*. Using this method instead of the original LU-decomposition, the increase of performance is significant, and enables us to use even more particles in the deformable models, while still running the simulations interactively.

3.8.3 Explicit integration

The semi-implicit integration method presented in the previous section, requires both the assembly of the stiffness matrix $\mathbf{K}(\underline{\mathbf{r}})$, and the inversion of the equally sized system matrix $\mathbf{A}_t$. We are the first to admit that both operations are anything but trivial to work with, and, if not implemented cleverly, huge performance drainers.

In this section we will introduce a discrete explicit integration solver derived from the second-order ordinary differential equations (3.8.1). In the following we will not focus on any specific manifold, as we will only use the elasticity symbolically. We will design the explicit method to work for a single particle, thus we will dissolve the vector collections $\underline{\mathbf{f}}$, $\underline{\mathbf{r}}$, and $\underline{\mathbf{e}}$, into independent 3d-vectors $\mathbf{f}$, $\mathbf{r}$, and $\mathbf{e}$ for each particle. The explicit method requires evaluation of both $\mathbf{e}$ and $\mathbf{f}$ at time t. Given a solver designed for a single particle, an algorithm is introduced to perform all calculations for each particle in turn.

Rewriting (3.8.1) into an equation of independent particle wise scalars and vectors yields

$$\mu \frac{\partial^2 \mathbf{r}}{\partial t^2} + \gamma \frac{\partial \mathbf{r}}{\partial t} + \mathbf{e}_t = \mathbf{f}_t \,, \tag{3.8.31}$$

where $\mathbf{e}_t = \mathbf{K}(\mathbf{r}_t)\mathbf{r}_t$. The isolated second-order accurate partial time differential equations from (3.8.31), can be approximated by the discrete central differences

$$\begin{aligned} \frac{\partial^2 \mathbf{r}}{\partial t^2} &= \frac{\mathbf{r}_{t+\Delta t} - 2\mathbf{r}_t + \mathbf{r}_{t-\Delta t}}{\Delta t^2} \\ \frac{\partial \mathbf{r}}{\partial t} &= \frac{\mathbf{r}_{t+\Delta t} - \mathbf{r}_{t-\Delta t}}{2\Delta t}. \end{aligned} \tag{3.8.32}$$

The explicit method can be obtained in the following way. Substituting (3.8.32) into (3.8.31) and rearranging yields

$$\begin{gathered} \mu\left(\frac{\mathbf{r}_{t+\Delta t} - 2\mathbf{r}_t + \mathbf{r}_{t-\Delta t}}{\Delta t^2}\right) + \gamma\left(\frac{\mathbf{r}_{t+\Delta t} - \mathbf{r}_{t-\Delta t}}{2\Delta t}\right) + \mathbf{e}_t = \mathbf{f}_t \\ \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t+\Delta t} - \left(\frac{2\mu}{\Delta t^2}\right)\mathbf{r}_t \\ + \left(\frac{\mu}{\Delta t^2} - \frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t-\Delta t} + \mathbf{e}_t = \mathbf{f}_t \\ \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t+\Delta t} = \\ \mathbf{f}_t - \mathbf{e}_t + \left(\frac{2\mu}{\Delta t^2}\right)\mathbf{r}_t - \left(\frac{\mu}{\Delta t^2} - \frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t-\Delta t}. \end{gathered} \tag{3.8.33}$$

Introducing the particle velocity

$$\mathbf{v}_t = \frac{\mathbf{r}_t - \mathbf{r}_{t-\Delta t}}{\Delta t} \tag{3.8.34}$$

into (3.8.33) we will get

$$\begin{gathered} \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t+\Delta t} = \\ \mathbf{f}_t - \mathbf{e}_t + \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)\mathbf{r}_t + \left(\frac{\mu}{\Delta t} - \frac{\gamma}{2}\right)\mathbf{v}_t. \end{gathered} \tag{3.8.35}$$

Isolating $\mathbf{r}_{t+\Delta t}$ leads to the desired result

$$\begin{gathered} \mathbf{r}_{t+\Delta t} = \\ \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)^{-1}\left(\mathbf{f}_t - \mathbf{e}_t + \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)\mathbf{r}_t + \left(\frac{\mu}{\Delta t} - \frac{\gamma}{2}\right)\mathbf{v}_t\right). \end{gathered} \tag{3.8.36}$$

What is left behind must be the question of whether or not the following holds

$$\begin{gathered} \left(\frac{2\mu}{\Delta t^2}\right)\mathbf{r}_t - \left(\frac{\mu}{\Delta t^2} - \frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t-\Delta t} = \\ \left(\frac{\mu}{\Delta t^2} + \frac{\gamma}{2\Delta t}\right)\mathbf{r}_t + \left(\frac{\mu}{\Delta t} - \frac{\gamma}{2}\right)\mathbf{v}_t. \end{gathered} \tag{3.8.37}$$

We calculate

$$\begin{aligned}
\left(\frac{\mu}{\Delta t^2}+\frac{\gamma}{2\Delta t}\right)\mathbf{r}_t+\left(\frac{\mu}{\Delta t}-\frac{\gamma}{2}\right)\mathbf{v}_t &= \\
\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_t+\left(\frac{\gamma}{2\Delta t}\right)\mathbf{r}_t+\left(\frac{\mu}{\Delta t}-\frac{\gamma}{2}\right)\left(\frac{\mathbf{r}_t-\mathbf{r}_{t-\Delta t}}{\Delta t}\right) &= \\
\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_t+\cancel{\left(\frac{\gamma}{2\Delta t}\right)\mathbf{r}_t}+\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_t-\cancel{\left(\frac{\gamma}{2\Delta t}\right)\mathbf{r}_t} & \\
-\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_{t-\Delta t}+\left(\frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t-\Delta t} &= \\
\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_t+\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_t-\left(\frac{\mu}{\Delta t^2}\right)\mathbf{r}_{t-\Delta t}+\left(\frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t-\Delta t} &= \\
\left(\frac{2\mu}{\Delta t^2}\right)\mathbf{r}_t-\left(\frac{\mu}{\Delta t^2}-\frac{\gamma}{2\Delta t}\right)\mathbf{r}_{t-\Delta t}. &
\end{aligned} \tag{3.8.38}$$

By (3.8.38) we conclude that (3.8.37) is true, thus we can use (3.8.36) to determine new particle positions using an explicit step-by-step integration method. The vectors in (3.8.36) are all 3-dimensional, and all matrices have been omitted. The inverse operation is now only performed on a scalar, and this is just the reciprocal of the scalar. Note (3.8.36) must be used for each particle to determine the new particle position. Also observe closely that the elasticity vector $\mathbf{e}$ in (3.8.36) is dependent of the 5×5 particle neighborhood around the particle at hand (the center of the 5×5 stencil) at time t, see 3.3 *The Stiffness Matrix*, thus no particle position must be updated before all particle positions at time $t+\Delta t$ have been computed.

The iterative explicit procedure is illustrated in Listing 3.8 for the case of discrete surfaces.

```
const scalar dt = ...  // the time step
// calculate all the new particle positions
Vector3 new_r[M,N];
for (n = 0; n < N; n++)
  for (m = 0; m < M; m++) {
    const Vector3 f = externalForces(m,n);
    const Vector3 e = elasticity(m,n,alpha,beta);
    const scalar tmp1 = (1/(dt*dt))*μ[m,n] + (1/(2*dt))*γ[m,n];
    const scalar tmp2 = (1/dt)*μ[m,n] - 0.5*γ[m,n];
    new_r[m,n] = (1/tmp1)*(f - e + tmp1*r[m,n] + tmp2*v[m,n]);
  }
// calculate new velocities and replace old positions with the new ones
for (n = 0; n < N; n++)
  for (m = 0; m < M; m++) {
    v[m,n] = (1/dt)*(new_r[m,n]-r[m,n]);
    r[m,n] = new_r[m,n];
  }
```

Listing 3.8 Pseudo code illustrating the iterative explicit integration procedure

It is assumed the external functions `externalForces()` and `elasticity()` exist and all α and β tensors have been pre-computed and stored in `alpha` and `beta`. The above pseudo code has been written for clarity and not performance. Of course several things can be improved as well as with the semi-implicit integrator.

3.9 Visualization

To this point, we have been concentrating on how to simulate elastically deformable bodies. Physically-based animation is more than theory and discussions on how to work around several

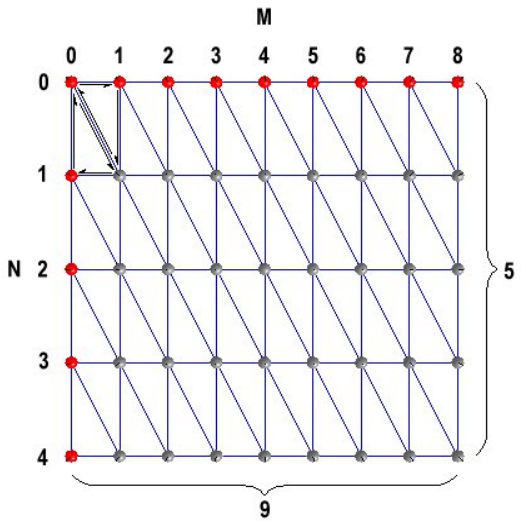


Figure 3.17 Surface grid, with $M = 9$ and $N = 5$, in total 45 particles

issues. In this section we will provide some guidelines on how to render the simulated bodies. Using the grid discretization method from [21], it is a straightforward procedure.
However, we will not discuss the use of any particular rendering system, as it is beyond the scope of this paper, but rather deal with some of the things that any rendering system will expect. For the interested reader we can reveal that we have used OpenGL to visualize our elastically deformable bodies and impenetrable rigid objects, and GLUT to make the source code compile and run on multiple platforms. Please refer to 4.1 *Program Description* for details about the actual implementation.

3.9.1 Direct visualization

The discretization of the continuous model can be used explicitly for visualization. The internal structure of the mesh grid, whether it is 2D or 3D, provides us with a readily accessible method for the creation of triangles to the underlying rendering system.

As visualization of a 3-dimensional mesh grid only is an extended case of the 2-dimensional one, the cube mesh can be considered as 6 surfaces, each to be treated as an individual surface. Even though we are able to model 3-dimensional solids, thus respecting the internal nodes, we only want to visualize the surface. Creating tetrahedra including all internal nodes from a solid is beyond this paper.
Figure 3.17 illustrates an example of how the grid mesh could be organized in this case for a deformable surface of 45 particles, 9 by 5. The discrete representation of the deformable model guarantees the consistency of the model, as two adjacent nodes in the mesh grid, always will be vertex neighbors in the WCS (World Coordinate System) no matter how much the model has deformed and deviates from its natural shape.

```
for (n = 1; n < N; n++)
  for (m = 1; m < M; m++) {
    draw(r[m-1,n-1], r[m,n], r[m-1,n]);  // draw the first triangle (cw)
    draw(r[m-1,n-1], r[m,n-1], r[m,n]);  // draw the second triangle (cw)
  }
```

Listing 3.9 Pseudo code illustrating simple triangulation algorithm using the mesh grid

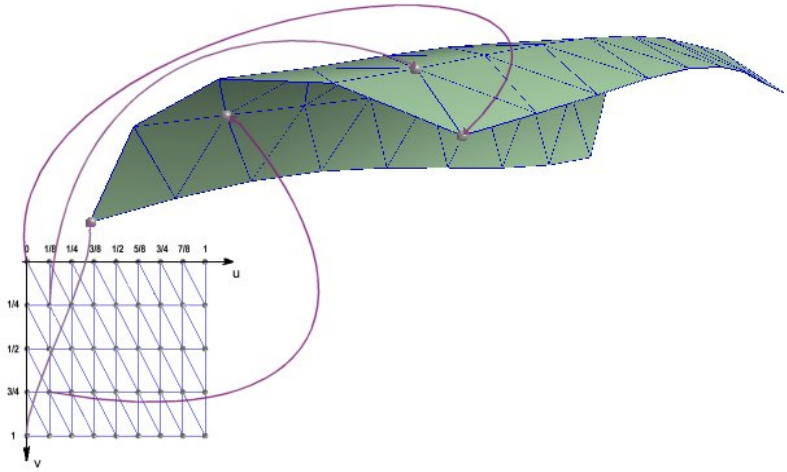


Figure 3.18 2D local grid and corresponding mapped surface in the WCS

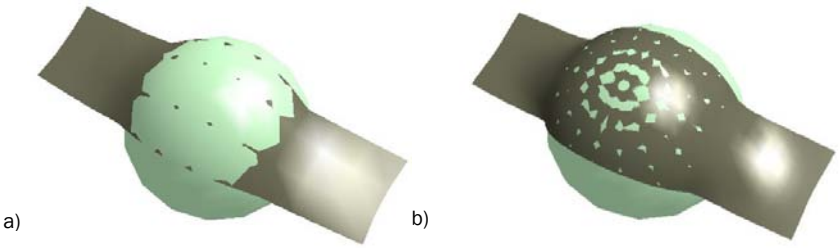


Figure 3.19 Collision area between an elastically deformable model and a rigid object **a)** Low particle density **b)** High particle density

We can use this quality to sketch a very simple algorithm that delivers all the triangles, and that can be used for visualization. Listing 3.9 shows the pseudo code for this straightforward procedure. Note that we use a zero based index representation of the mesh grid. Also notice in Figure 3.17 that we have marked some exterior particles. The marked particles indicate that they are not explicitly used, as Listing 3.9 also states.

The mapping from local grid nodes to the world coordinate system is depicted in Figure 3.18, where some mesh node mappings are illustrated in correspondence to the vertices in the WCS. The visualized surface could just as easily have appeared after applying the triangulation algorithm from Listing 3.9.

When rendering triangles one should always ensure that all triangles are rendered exactly alike. That is, the triangle orientation must be the same for all triangles. We have chosen the orientation to be in clock wise or CW order, as stated in the comments from Listing 3.9. The orientation is directly dependent on the direction of the surface normals.

3.9.2 Particle density

We want to address a small feature regarding visualization of the collision between elastically deformable objects and impenetrable rigid object. The issue at hand is the actual particle density

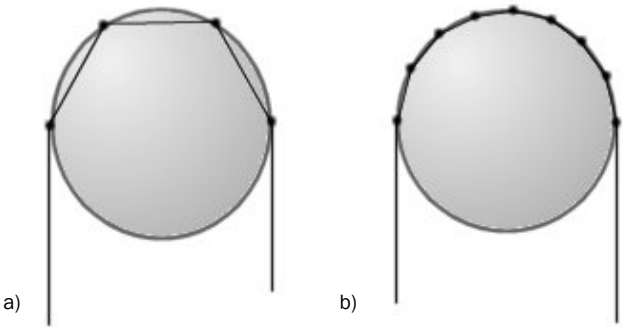


Figure 3.20 Collision area between a curve and a circle **a)** Low particle density **b)** High particle density

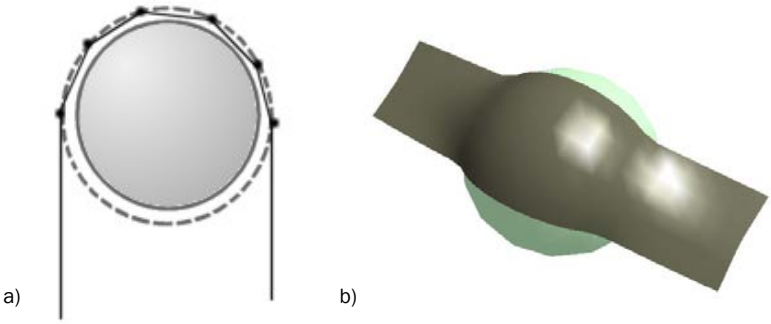


Figure 3.21 Scaling hack **a)** 2D schematics **b)** Deformable surface floating just above a rigid object

of a deformable model when collided with a rigid object. Assuming an elastically deformable object has collided with an impenetrable rigid object, all colliding particles are resting on the surface of the rigid object. Dependent on the particle density we will get different sized holes in the visualization of the collision, as depicted on Figure 3.19. The reason for this irregularity is not due to some tedious error in our interpretation of the rendering process, but merely due to the linearity of triangles.

We will discuss the inappropriateness of this feature and propose a small hack to solve the issue. We will use the term 'particle density' as the amount of particles located on a collision area, thus a high particle density is equivalent to a great deal of particles on a relative small collision area. The issue will be somewhat clearer when reduced to 2D. Figure 3.20 illustrates the two cases from figure 3.19 in 2D, where the triangles similar are reduced to lines. Due to the fact that each line segment penetrates the circle, the line is only visible on a tiny area around each vertex or particle. Increasing the particle density will reduce the amount of holes as the small visible areas grow proportional to the density. However, we can only guarantee a complete visible curve in the 2D case on Figure 3.20, or surface in the 3D case on Figure 3.19, if enough particles exist to fill the entire area, that is, particle positions should only be displaced corresponding to the size of a pixel. Of course this feature only applies to local curved manifolds, e.g. a circle with a radius going towards infinity is locally defined as a straight line, and is not considered a problem of this kind.

Using a very high particle density just to correct the visual irregularity seems a bit intense. We must be able to do better, primarily due to the fact that performance is directly dependent on the amount of particles, i.e. the particle density. We have implemented a small hack, which elegantly

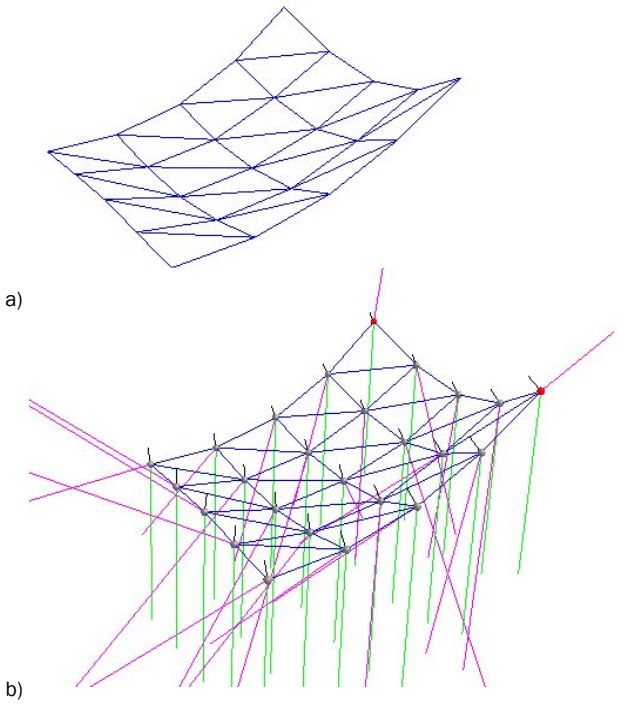


Figure 3.22 A simple wire frame cloth of 5x5 particles **a)** Visual debug off **b)** Visual debug

solves all the troubles. We are able to use whatever particle density we wish, and still being able to avoid any holes in the visualization of the deformable model resting on rigid objects. The whole idea originates from the impenetrable rigid objects. We will use their correct sizes for everything concerning collision handling, but scale down the actual visualization. Figure 3.21a depicts the beauty of the hack, where the dotted circle indicates the correct referenced size, and the actually size of the visual circle deviates by a fraction, just enough to make the visualization convincing.
We call it a hack as this solution does not solve the issue correctly, but instead tries to cheat the viewer to believe that the deformable object is resting on the rigid body, whereas it is actually floating above it. We have enabled the end user to control a visual scaling coefficient when he creates the rigid bodies. This is due to the fact that it is practically impossible to design a scaling constant and function that will satisfy all shapes of rigid bodies and all sizes of particle densities. In Figure 3.21b the rigid body has been scaled to 97.5% of its original size, which is just enough to create a convincing result.

3.9.3 Visual debug

Other information about the visualization of the simulated elastically deformable models might be important. Such information could include a visual presentation of all the external forces along with all the internal elastic forces acting on a particle, the surface normals emanating from the particles, and the particles themselves. The information could be represented in different ways, in different colors, etc.
We have implemented several visual debugging facilities:

- Movable particles are shown as small grey spheres
  As the center of the sphere we simply use the particle position $\mathbf{r}(\mathbf{a})$

- Fixed particles are shown as red spheres
  Each particle has a property indicating whether or not it is fixed. We simply draw the sphere in a red color if this Boolean is true
- Surface normals as short black lines extending from particles
  All normals are scaled to gain the same magnitude
- External forces are illustrated as green lines extending from particles
  A green external force vector representation is the sum of all external forces working on a particle. An external force pulls the particle in the direction from the particle to the end of the line with the strength, proportional to the line length
- Elastic forces as purple lines directed toward the particles
  The elasticity force $\mathbf{e}$, can be used directly from the calculation during an explicit integration step. During a semi-implicit simulation step we must perform an extra matrix/vector multiplication to retrieve the actual elasticity force. This is done by $\underline{\mathbf{e}} = \mathbf{K}(\underline{\mathbf{r}})\underline{\mathbf{r}}$. An internal elastic force pushes the particle in the direction from the end of the line towards the particle with the strength, proportional to the length

Figure 3.22 illustrates an example where a small piece of clothing initially is placed horizontal (on the xy-plane). All particles are influenced by gravity (negative z-axis), but two of the corners are fixed. Upon release, the gravitational force field will pull the particles vertically downwards. Due to the elastic properties of the clothing the internal elastic force will push the particles and keep the internal structure of the cloth together. The direction of both internal and external force vectors are illustrated as they appear physically, and the length of the vectors are proportional to the force magnitudes. Further details about the how to control the different visual debugging features of the program please consult 4.3 *User Guide*.

3.9.4 Texture coordinates

Upon visualization of an elastically deformable object several shading and mapping strategies can be selected. One of these is texture mapping. The standard procedure of using texture mapping consists of drawing specific colored pixels onto the triangles, or any other polygon, of the geometric object. The source of the colored pixels usually comes from one or more texture maps. We will restrict the use of textures to planar 2D texture maps. Please consult [25, 9] for details about rendering and shading techniques using texture mapping.
When visualizing deformable objects using standard 2D textures, we will need the texture coordinates. Both components (u, v) from the texture coordinate system UV ranges from 0 to 1, i.e. $0 \leq u, v \leq 1$. We quickly notice the similarity between texture coordinates (u, v) and the intrinsic or material frame coordinates $\mathbf{a}$ of the body Ω. As a matter of fact, both coordinate systems could easily be the same. In the discretization of the continuous surface we approximate the continuous vector position $\mathbf{r}(a_1, a_2)$ by the grid function $\mathbf{r}[m, n] = \mathbf{r}(mh_1, nh_2)$, where $0 \leq m \in \mathbb{N} < M$ and $0 \leq n \in \mathbb{N} < N$, and $h_1 = \frac{1}{M-1}$ and $h_2 = \frac{1}{N-1}$. When using $[m, n]$ to index the particle position, we have implicitly determined the texture coordinate $(u, v) = (mh_1, nh_2)$. Both constants h_1 and h_2 can be calculated prior to simulation start, thus using only two multiplications to determine a single texture coordinate, rather than storing all texture coordinates in memory.

3.9.5 Material descriptors

This section will describe an idea for future work regarding simulation of materials. A possible way to model all the different material properties conveniently, is to use textures to control the tension $\boldsymbol{\eta}$ and the surface rigidity $\boldsymbol{\xi}$. The idea is to use the pixel values from the textures to describe the different materials. The most convenient way is to use a texture map for each material. Using the

implicitly given parameters, provided by the texture map, it is a matter of using any kind of interpolation to determine the anisotropic material values. For surfaces this will require two material functions, $\boldsymbol{\eta}(u,v)$ and $\boldsymbol{\xi}(u,v)$, and only a single function is required for solids, $\boldsymbol{\eta}(u,v,w)$. Several questions arise regarding how to find and interpret the texture pixels. Should we use 3D textures for solids or several 2D textures? Should we use grey color tone bitmaps or should we distribute the values of RGBA (Red, Green, Blue, and Alpha) for texture pixels onto the particle materials? These questions are not considered problems but merely issues that need to be addressed. Other possible methods to model object materials might exist as well, thus indicating many intriguing ways to solve the puzzle.

One advantage by using textures to model flexible materials is the possibility to use hardware memory on the graphics adapter rather then on the motherboard, thus freeing main memory for other processes to use. While using the graphics adapter for texture storage another advantage also appears. Practically most of the calculations performed in each frame step can be designed to operate on the GPU (Graphical Processing Unit) using textures, thus gaining a huge performance boost. However, this hardware approach is beyond the scope of this paper, but does give the occasion for an interesting future discussion on the subject.

4 Implementation

During the last chapter we have made reference to the actual implementation of the elastically deformable models several times. It is now time to look more closely at the details of our implementation.

We have implemented the method described in [21] using the C++ language and it takes up about 4,900 lines of source code, not including comments and 3rd party source. The implementation includes, besides the actual method to simulate elastically deformable models, different types of generic models and rigid objects, an application that demonstrates the use of the method, and a flexible input interface. This interface allows us to import XML documents that describe configurations of deformable objects and scenes uniformly.

In this chapter we will start up by briefly going through the program description of our implementation, followed by our actual simulation procedure, with references to the discussions in 3 *Execution and Practice*. We will include a section on how to obtain the source, 3rd party dependencies, and how to control the behavior of the implementation. A section will contain a user guide on how to use and control the running application. Finally a section is included, in which we will discuss the layout of the XML scene configuration, and thus encourage the reader to create new and exiting models and use our simulator to create physically based animations of deformable objects.

4.1 Program Description

We will not go into details about the implementation and all the degenerate cases. The interested reader is encouraged to read the source code. In this section we will give an overview of the implementation using the UML notation. We have used UML to briefly give the reader a visual idea of the structure of the used classes. We begin by showing all the referenced classes and their isolated inheritances, followed by the associations between the classes.

4.1.1 Class hierarchy

Six abstract base classes mostly define the whole system. These are `EDMMaterial`, `EDMForce`, `EDMObject`, `DeformableCurve`, `DeformableSurface`, and `DeformableSolid`, and their inheritance structures are depicted in Figure 4.1 to Figure 4.4. The class `DeformableBody` is a super class whose only purpose is to assemble its three deformable model base classes. It is doubtful that any new generalization of `DeformableBody` will be needed. One I/O utility base class is used to support data access to and from the deformable system. The utility class `DeformableSystemIO` and its generalization are visualized in Figure 4.5. The 7 base classes have been implemented with a usable interface to make it fairly easy to expand the system with new forces, objects, models, etc.

4.1.2 Class associations

We have previously seen the different base classes for the implementation along with their generalizations, but we do not yet know how they are working together. In Figure 4.6 we have illustrated the association diagram between the relevant classes. The diagram depicts ownership, references, and dependencies. Two classes might not fit into the diagram, but they both play an important part in the class associations. The class `Application` of course owns and controls everything. The specialized force class `EDMCollision` is not used like the rest of the `EDMForce` implementations. It requires to be called differently, but this behavior is completely controlled by `DeformableBody`, thus there is no need for the user to worry about this.

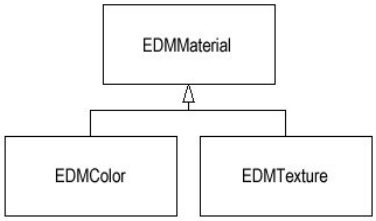


Figure 4.1 The `EDMMaterial` base class and its two generalizations `EDMColor` and `EDMTexture`

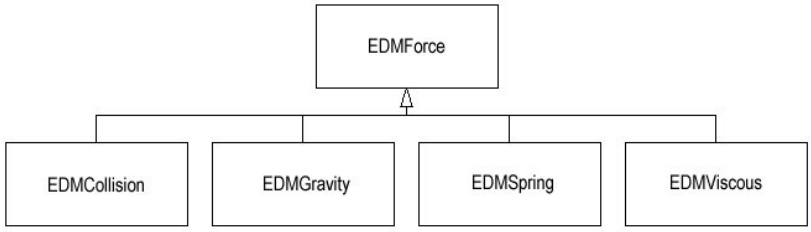


Figure 4.2 The `EDMForce` base class and its four supported generalizations

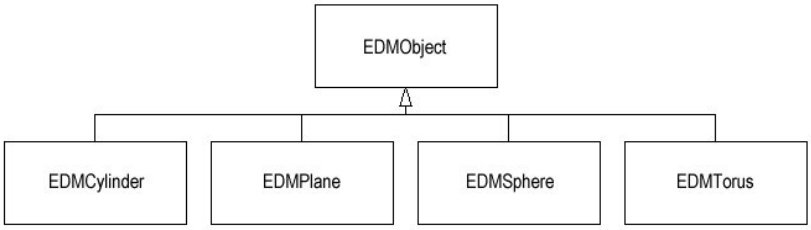


Figure 4.3 The `EDMObject` base class and its four supported generalizations

Although `DeformableSystem` owns all instances of `DeformableBody`, `EDMForce`, `EDMMaterial`, and `EDMObject`, it is any generalization of `DeformableSystemIO` that actually creates all the instances. These relations and responsibilities are acceptable as `DeformableSystem` is the only one that can choose its friends, thus it is responsible for its own choices. Other classes/structures are used throughout the implementation, but these are all auxiliary classes, and not really important for a programmer that wish to expand the current system. One class is important to describe, though. The `EDMParticle` is the base class for all particles, and it has all the generic members referencing to the previous chapters. For more details look into the header file "`edm_types.h`".

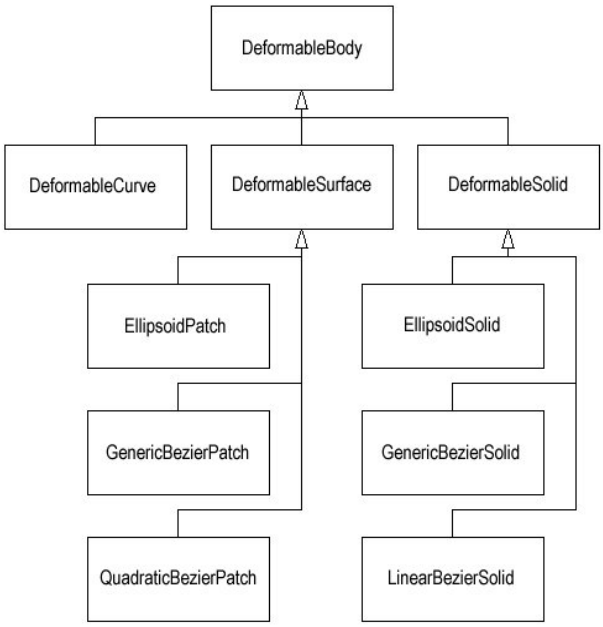


Figure 4.4 The `DeformableBody` super class, its three base generalizations, and the concrete implementations of the used model types

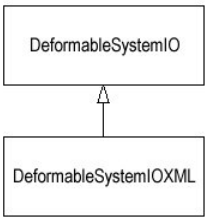


Figure 4.5 The `DeformableSystemIO` base class and its only supported generalization class `DeformableSystemIOXML`

4.1.3 Simulation procedure

We will now present the actual simulation procedure we have used in the implementation. We will describe the procedure for deformable surfaces, thus it is straight forward to expand the procedure for solids. Equation references are included directly in the step descriptions and if a step addresses some bigger area of the document, we will include a reference to the section in parentheses, in which more information can be found.

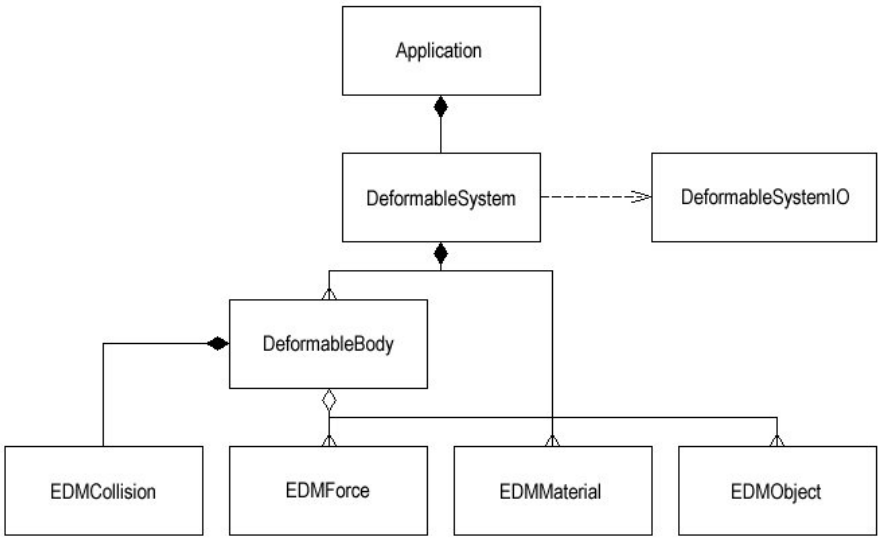


Figure 4.6 The class associations between all relevant classes

4.1.3.1 Initialization step

The following initialization procedure is performed for each deformable body:

1. The discrete 2D grid is created of $M \times N$ particles
2. All the grid particles are initialized – step **c** must be performed before steps **d** and **e**, as metric and curvature tensors are based upon the rest shape of the body
   a. Velocities are reset
   b. Textures coordinates are computed (3.9.4 *Texture coordinates*)
   c. Positions are assigned as the natural shape of the body
   d. Metric tensors are calculated using (3.2.8)
   e. Curvature tensors are calculated using (3.2.15)
   f. Positions are reassigned as the initial shape of the body
   g. Surface normals are computed using (3.7.8)

4.1.3.2 Simulation step

The next items all include a single simulation step, thus the steps **3** to **6** should be repeated over and over again until the simulation stops:

3. The α and β tensor fields are created
   a. The α tensors will be calculated using (3.2.3)
   b. The β tensors will be calculated using (3.2.14)
4. The new particle positions are computed (semi-implicit integration) - if the explicit integration is being used jump to step **5**
   a. The stiffness matrix is assembled using the pre-computed α and β tensor fields (3.3 *The Stiffness Matrix*)
   b. The system matrix is computed using (3.8.17)
   c. The external force vector collection is assembled for all the particles using (3.5.3), (3.5.5), (3.5.11), and (3.6.7) or (3.6.9) if the penalty collision model is used
   d. The effective force vector is computed using (3.8.18)

   e. All the new particle positions are calculated using (3.8.19) and stored in a vector collection
   f. Jump to step **6**
5. The new particle positions are computed (explicit integration) - the following sub-steps are performed for each particle in turn
   a. The elasticity is calculated using the pre-computed α and β tensor fields (3.2.1)
   b. The external force vector is computed using (3.5.3), (3.5.5), (3.5.11), and (3.6.7) or (3.6.9) if the penalty collision model is used
   c. The new position is calculated using (3.8.36) and stored in a vector collection
6. All particles will be updated - For each particle in turn
   a. Old position will be assigned to the current position
   b. If particle is not fixed, current position will be assigned to the new calculated position
   c. If collision by projection is used the current position will be updated using the projection vector (3.6.2 *Projection*)
   d. New velocity is computed using e.g. (3.6.10)

What is missing to fulfill the simulation loop is the visualization of all the objects and the models. The visualization of the system follows directly after each simulation step. Please turn to 3.9 *Visualization* for discussions and details about the visualization part.

4.2 Compiling and Building

The actual implementation of the elastically deformable models is integrated into the open source project at DIKU[3] called OpenTissue [6]. The full source code can be obtained from the project web page. However, we are not the only ones with write access to the OpenTissue project, thus to guarantee that the original unmodified version of the source code can be obtained, we have also included it on the supplemental CD [3].

4.2.1 3rd party dependencies

The only native dependency of the implementation is the OpenTissue project [6]. We have followed the rules and guidelines from OpenTissue, the implementation is therefore written in ANSI C++, using the Standard Template Library or STL extension. We have also used some of the 3rd party dependencies from OpenTissue. Please consult the OpenTissue web page for details about the dependencies and how to compile and build the OpenTissue library and how to use the attached demonstration. On the enclosed CD [3] we have also included all the 3rd party software packages required by the OpenTissue project, in their available versions at this time of writing.

4.2.2 User controlled behavior

Most of the program execution flow can be controlled by the programmer at compile time. We have gathered all such user configurations in the header file `"edm_config.h"`. This section will describe the different behaviors and methods that can be tweaked by a programmer, to activate or deactivate different paths of the program. The defines described in this section cannot be changed at run time. The programmer must decide which methods to use at compile time. All defines are described in the header file.

The settings present in `"edm_config.h"` are the default system configurations we prefer to use. To successfully compile the source code, no settings need to be defined at all. In other words, the header file `"edm_config.h"` can be empty, or completely replaced by another header file. We have added the define `EDM_DEFAULTS` on the preprocessor to tell the compiler the default settings in `"edm_config.h"` should be used. Any programmer can choose to use the same settings or

[3] Datalogisk Institut, Københavns Universitet, DIKU (Department of Computer Science, University of Copenhagen)

include new ones, by providing another preprocessor define and modifying "`edm_types.h`" to let the compiler know that another set of configurations will be used.

4.2.2.1 Collision handling

We have described two different methods of collision handling, the penalty collision force in 3.6.1 *Penalty collision force* and the projection method in 3.6.2 *Projection*. Both collision procedures have been implemented. The penalty collision force is the default collision handler, but collision handling by projection can be activated by including the line

```
#define EDM_COLLISION_PROJECTION
```

When penalty collision force is active, `EDM_COLLISION_PROJECTION` will not be defined. The programmer can choose between both penalty methods. The default penalty force is our own modified version, which reduces the aggressiveness of the original exponential collision force. The original collision force from [21] with the discussed correction can be the favorite by specifying

```
#define EDM_ORIG_PENALTY_FORCE
```

If both `EDM_COLLISION_PROJECTION` and `EDM_ORIG_PENALTY_FORCE` are undefined, our own penalty force collision is used. In 3.6.1.3 *Reduced aggressiveness* we also discussed a penetration threshold constant, to limit the penalty force activation. This threshold can be used by including the `EDM_PENETRATION_THRESHOLD` define. As the penalty force only should be activated of the particles have penetrated a rigid object, the programmer should give the define a negative value[4], e.g.

```
#define EDM_PENETRATION_THRESHOLD  -0.5
```

4.2.2.2 Integration method

The two integration schemes described in 3.8 *Numerical Integration* can both be activated. The default method is the semi-implicit integration. However, it can be replaced by the explicit integration method by including

```
#define EDM_EXPLICIT_METHOD
```

4.2.2.3 Stiffness matrix method

When the default semi-implicit integration method is used, the stiffness matrix will at some point be needed. The stiffness matrix is assembled using the elasticity calculations. We have experimented with several calculation strategies of the elasticity, as discussed in 3.2.4 *Alternate method* and 3.4.3 *Stiffness boundary conditions*, this resulted in 4 different surface stiffness versions and 2 different solid stiffness versions. All the surface stiffness versions can be used by including

```
#define EDM_SURFACE_STIFFNESS_V n ,
```

where n is one of either `1`, `2`, `3`, or `4`. If no surface stiffness version is provided, `EDM_SURFACE_STIFFNESS_V4` will be defined as default, as it is the best possible stable version. In Table 4.1 we have listed the differences between the 4 versions.

The solid stiffness versions can both be used by including

```
#define EDM_SOLID_STIFFNESS_V n ,
```

where n is either `1` or `2`. If no solid stiffness version is provided, `EDM_SOLID_STIFFNESS_V2` will be used as default, as it is the best possible stabile version. In Table 4.2 we have listed the differences between the 2 versions.

[4] If no value is provided a default value of 1 (of true) is normally used by the compiler

Surface stiffness version define	Status	Description
`EDM_SURFACE_STIFFNESS_V1`	Beta stable	Implemented with the difference operators proposed by Terzopoulos et. al [21], with α and β considered as functions, thus using the product rule in the derivation of $\mathbf{e}[m,n]$. When the D^+ operators are indexing beyond the boundaries all difference operators are reversed
`EDM_SURFACE_STIFFNESS_V2`	Unstable	Both 1st order and 2nd order derivatives use central difference operators, rather than the proposed forward and backward operators. Only tension is included in the calculations
`EDM_SURFACE_STIFFNESS_V3`	Unstable	All 1st order derivatives use central difference operators. 2nd order derivatives are calculated as version 1. Only tension is included in the calculations
`EDM_SURFACE_STIFFNESS_V4`	Stable	Implemented without using the product rule, i.e. using α and β as constants. Both tension and rigidity use the free boundary conditions as described by Terzopoulos et. al in [21]

Table 4.1 Description of the 4 different surface stiffness versions

Solid stiffness version define	Status	Description
`EDM_SOLID_STIFFNESS_V1`	Beta stable	Implemented as surface stiffness version 1, but extended to solids, thus the β terms are not included. No boundary conditions are used, resulting in 3 sides of the grid not working correctly
`EDM_SOLID_STIFFNESS_V2`	Stable	Implemented as surface stiffness version 4. Again extended to solids and the free boundary conditions are activated

Table 4.2 Description of the 2 different solid stiffness versions

The threshold value to handle numerical instabilities in the stiffness matrix, discussed in 3.3.4 *Numerical instabilities*, is given by the `EDM_EPSILON` define. The define should be initialized with an appropriately small fraction, which both secures that the system will not be unstable, but also that not too much is cut away. If `EDM_EPSILON` is not added, it will automatically be defined with a value of 2^{-13}, which we have experienced to be a useable threshold value, e.g.

```
#define EDM_EPSILON  (1./8192.)
```

4.2.2.4 Linear equations solver

Solving the linear system equations in the end of the semi-implicit integration procedure, we need to inverse the main system matrix. The inverse of the matrix is by default performed using the LU decomposition algorithm from [6]. Although this algorithm is exact, it is far from performance

friendly, and simply cannot be used if we wish to achieve a detailed simulation in real time. If a simulation consists of many particles, we can use the ATLAS/UBLAS math library [1] to speed things up a great deal. To use the ATLAS/UBLAS library include

```
#define EDM_USE_ATLASUBLAS
```

As described in 3.8.2.1 *Conjugate gradient method* the properties of the system matrix enables us use to use the conjugate gradient equation solver, thus speeding things up significantly. To use conjugate gradient include

```
#define EDM_CONJUGATE_GRADIENT
```

Conjugate gradient uses BOOST and UBLAS, thus `EDM_USE_ATLASUBLAS` must be defined. If `EDM_CONJUGATE_GRADIENT` is defined, `EDM_CG_TOLERANCE` is mandatory and must also be defined with an acceptable error tolerance value, e.g.

```
#define EDM_CG_TOLERANCE  0.00001
```

We have implemented two ways of using conjugate gradient. The system can calculate the maximum amount of needed iterations and a sufficient error tolerance that is dependent of the given body and its amount of particles. The programmer can also explicitly tell the system how many iterations the conjugate gradient method should use. To explicitly set the iterations and the error tolerance define `EDM_CG_ITERATIONS` with an integer to indicate the amount of iterations, e.g.

```
#define EDM_CG_ITERATIONS  256
```

4.3 User Guide

This section will go through the different parts of the attached demonstration application. Our main implementation of the method is totally independent of the provided application, as we have implemented an “all-you-need-to-know” interface, that only public requisite functions. The application is simple in shape and does not include a fancy GUI. The user can interact with the system through the spawned OpenGL window. Feedback to the end user will be given graphically in the OpenGL window and information about the state of the system is given on the command prompt. In Table 4.3 is listed a brief explanation of the different possibilities of supported user interactions, that can be carried out once the system has been successfully initialized. Due to the fact that all keys are case insensitive, they are only showed in lowercase. The last two actions from Table 4.3 can be performed whether or not the simulator is running. If a particle is being pulled around while the simulator is running and visual debug has been activated, notice that the particle is colored red during the pull action. We treat the particle as if it has been constrained to a fixed location, and the location is the current position of the mouse pointer. As soon as the pulling ends the particle is yet again released, thus its color is turned back to grey. If the simulator is paused or was never started, particles can still be pulled around. However, upon particle release they will stay fixed, and continues to be fixed even though the simulation is started. Position constrained particles can be released by the same method, if the simulator is running, simply by clicking on them with the `LMB` while holding down `CTRL`.

To launch the demo application, run the executable with an XML scene configuration as the argument, e.g.:

```
deformation scene.xml
```

The configuration must be a valid XML document to initialize the system successfully. Please note that the application will not perform the actual XML validity check. It will only make sure that the rules of the scene layout are obeyed. There are currently no other way to start the application. The user *must* provide a scene configuration or the system will not start at all. The syntax of a scene configuration is described next.

Action/Key	Description
space	Simulation Start/Pause Default: Stopped
q	Quits/Terminates the demonstration
esc	Same as 'q'
y	Takes a screenshot of the current OpenGL window. The files are named: screen $nnnnn$, where $nnnnn$ is a 5 digit number in succession, beginning from 00000
d	Visual Debug On/Off Default: Off For more information see [Visual debug.REF]
w	Wireframe On/Off Default: Off Will show all the surface triangles in blue wireframe
r	Resets the simulation, but keeps the current action states
f	Frames Per Second (FPS) On/Off Default: Off
LMB	Rotates the viewport about the origin in WCS
ALT+LMB	Zooms the viewport while moving the mouse vertical
SHIFT+LMB	Pans the viewport
CTRL+LMB (running sim)	Pulls the selected particle
CTRL+LMB (paused sim)	Fix the selected particle

Table 4.3 Possible user interactions – LMB := press and hold Left Mouse Button, if not stated otherwise the action will first become effective upon mouse movement

4.4 Scene Configurations

During the development of the source code, we quickly came to the conclusion that it was too tedious to rebuild the project over and over again, each time we had a tiny modification to the actual test scene layout. The solution for the problem is not at all ground-breaking, as a matter of fact one of the things we are faced with prior to an entirely new project, is to always create an I/O interface as one of the first things. Typically a standard flat ASCII file interface is established and it enables external communication between the application and the developer. An ASCII file is often too linear in its construction, making it hard to change the structure of the file interface without modifying the code equally. To comply with both issues we have chosen to use the eXtensible Markup Language or XML. XML gives us a highly flexible way to set up our own rules for the scene configuration syntax.

In this section we will go though the syntax and the rules to create or modify a valid scene configuration that is necessary to communicate with our provided application. We will also list the concrete possibilities of models, objects, forces, and materials we have implemented as of this writing.

4.4.1 Common principles

We wanted to reuse as much syntax as possible throughout the different areas of the scene configurations in order to make the whole concept homogeneous. In this subsection we will introduce the XML nodes and attributes along with some specific formatting that we have used repeatedly.

4.4.1.1 Repeated formatting

Our naming convention follows these simple rules:

- All nodes begin with a capitalized letter, and all following new words are separated by a capitalized letter likewise, e.g. `ThisCouldBeANode`
- All attributes are lowercased and consist of a single word only, e.g. `temperature`
- All system attribute values use the same convention as nodes, e.g. `"OneHugeValue"`

We use 32bit unsigned integers for all indices. An integer index value is zero based, thus index `0` references the first item within a list of items. All other numbers are considered real, and for those we use 64bit precision floating point numbers. If used we require the decimal comma of a floating point to be a punctuation mark. We have chosen a specific formatting for other used data types:

- Vectors and/or Positions are clamped by brackets and their components are usually all treated as reals and separated by commas, e.g. a 3D vector `"[0, 0, -9.82]"`
- Matrices and/or Tensors are also clamped by brackets with comma separated components, which are row vectors, e.g. a 3x3 Matrix `"[[1,2,3], [4,5,6], [7,8,9]]"`
- Booleans are treated as strings and their values must be either `"Yes"` or `"No"` for true or false respectively

4.4.1.2 Repeated attributes

Some attribute names are used in several places and has the same meaning:

- `id` – an alphanumeric identifier or reference name, used to uniquely identify a concrete instance
- `type` – used both as abstract system types and specific generalizations, appears together with the `id` attribute upon creation of a concrete instance and for `Bind` nodes
- `particle` – a particle index reference, used to make an operation specialized to a single particle rather than for the whole system. Particle indices use vector notation, with unsigned integers as components, e.g. a surface particle `"[0,9]"` and a solid particle `"[1,1,1]"`

4.4.1.3 Repeated nodes

The following child nodes are used for almost all instances:

- `Properties` – this node describes all specific properties of the concrete instance, thus defines the behavior of the instance. The `Properties` child has no mandatory attributes as they are all dependent of the object instance at hand. They will be defined when they are needed. If more than one `Properties` node exists for a specific instance, only the first to appear will be utilized
- `Bind` – the `Bind` node is used to link different types of instances together. `Bind` has no children, but two mandatory attributes and one optional.
  The mandatory attributes are
  - `type` – this attribute is described above under 4.4.1.2 *Repeated attributes*. The value of `type` must be one of the following three supported identities
    - `"Material"` – binds a material to the instance at hand
    - `"Force"` – applies an external force to the instance
    - `"Collision"` – tells the system that the instance at hand should handle collision with an object

- `ref` – the reference attribute is used to point to the concrete user instance of the type that is given. All values of `ref` are user specified

The optional attribute is

- `particle` – this attributes is described in 4.4.1.2 *Repeated attributes*, and it is only available when `type` is `Force`

Two examples of the Bind node:
Example 1 - A collision handler is attached to the rigid object with `id = "Jack"`:

```
<Bind type="Collision" ref="Jack" />
```

Example 2 - A user wind force is applied to the particle `"[10,0]"`:

```
<Bind type="Force" ref="Wind" particle="[10,0]" />
```

4.4.1.4 Main configuration node

Each scene configuration must have the mandatory `EDMConfig` node, which is the main node and describes the whole scene. The `EDMConfig` node only requires the one mandatory attribute id, which uniquely identifies the scene. It is not necessary for `EDMConfig` to be the root node of the XML document, but in all our scene configurations we have made it so. All details about the scene are defined within the `EDMConfig` node, thus everything described in this section is about the interior of this node. The `EDMConfig` node must have exactly 4 distinct mandatory child nodes, in no particular order. These children are in turn described next, and as children of the main node it all looks like

```
<EDMConfig id="MyConfiguration">
+ <MaterialContainer>
+ <ExternalForces>
+ <RigidObjects>
+ <DeformableBodies>
</EDMConfig>
```

4.4.2 Material container

The `MaterialContainer` node is the main node for all materials in the scene, whether they are used or not. Materials are used to decorate various types of visual instances to give the simulated scene a more convincing look. Only one type of distinct `MaterialContainer` child node will be considered, and this child must be the `Material` node. The `Material` node requires two mandatory attributes; `id` and `type`. The `type` attribute must be one of two possible values; `"Color"` or `"Texture"`. Each type of material requires the `Properties` child node, and they are described next.

4.4.2.1 Color material

A color material will render the surface of the visual instance in a user selected color. The `Properties` child node of a `Material` node of `type = "Color"` requires one mandatory attribute `rgb`, which defines the red, green, and blue color channel, separated by commas. The color format is floating point RGB, thus the range for each color channel is $[0..1]$. As of writing we do not support the alpha channel. The `rgb` attribute is not a vector. Below are some examples of color materials:
Example 1 - Defines a pure red color with identification Red:

```
<Material id="Red" type="Color">
  <Properties rgb="1.0, 0.0, 0.0" />
</Material>
```

a)  b)  c) 

Figure 4.7 Texture samples **a)** An oriental carpet pattern **b)** A scrub **c)** One side of a crate

Example 2 - Defines a dark beige color with identification Pole:

```
<Material id="Pole" type="Color">
  <Properties rgb="0.78, 0.59, 0.39" />
</Material>
```

4.4.2.2 Texture material

A texture material goes even further than a color material. A texture or texture map is a collection of color pixels. The texture material will render the surface using an external user specified texture map. The `Properties` child node requires one mandatory attribute `file`, which points to the external texture map by file path and name. The path can both be absolute or relative to the working folder, which is the folder, from which the application has been launched. Always use the forward slash character when separating folders, no matter what the underlying operating system is, e.g. `file = "data/textures/map.bmp"`. As of writing we are only supporting two types of 2D texture formats; 24bit native Windows Bitmap (`*.bmp` or `*.dib`) and uncompressed 32bit Truevision Targa (`*.tga`). A texture material file must have a supported extension to be imported. We are using standard OpenGL functions to import texture maps, thus the dimensions of a 2D texture map must be

$$2^n \times 2^m, \;\; n,m \in \mathbb{N}\,, \tag{4.4.1}$$

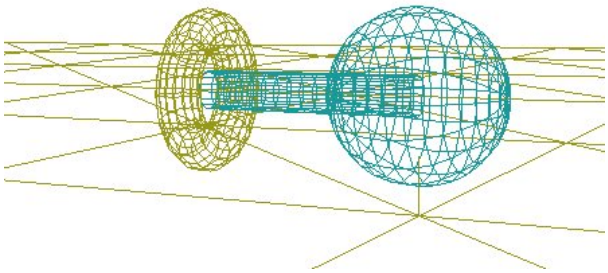
where $n,m \geq 2$, that is at least 4×4 pixels. Notice n and m does not have to be equal, thus the shape of a texture map is not required to be a square. Three visual examples of texture maps are illustrated in Figure 4.7, and below is an example of the texture material with identification `"Box"` that points to a crate texture Bitmap located in a textures folder within the working directory:

```
<Material id="Box" type="Texture">
  <Properties file="textures/crate.bmp" />
</Material>
```

4.4.3 External forces

All external forces must be defined as children of the main `ExternalForces` node. Only children of `Force` nodes will be utilized. A `Force` node also has the two mandatory attributes; `id` and `type`. The value of the `type` attribute for a `Force` node must be one of the following three possibilities; `"Gravity"`, `"Spring"`, or `"Viscous"`. No other type will be accepted. We have only implemented the external forces discussed in 3.5 *External Forces*. We will now briefly describe the needed attributes for each `Properties` child node.

a)



b)

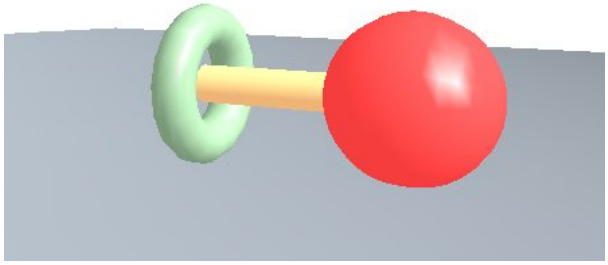


Figure 4.8 The four supported objects. From left to right the torus, cylinder, sphere, and the plane below **a)** shown in wireframe **b)** shown with color materials

4.4.3.1 *Gravity force*

A `Force` node with `type = "Gravity"` requires only a single mandatory `Properties` node attribute. The attribute is `g`, and it is a 3D vector that defines the gravitational acceleration field. The example below is used in many of our configurations, and it also demonstrates that a `type` value is not a reserved name:

```
<Force id="Gravity" type="Gravity">
  <Properties g="[0, 0, -9.82]" />
</Force>
```

4.4.3.2 *Spring force*

A spring `Force` node has `type = "Spring"`. Its `Properties` node requires two mandatory attributes; `k` and `r0`. The spring constant `k` is a real value, where `r0` is the fixed position in world coordinates. It is not common that a spring force is used globally, thus upon binding the `Bind` node, one typically specifies the `particle` attribute to only apply the force to a single particle. If more than one particle should be bound to a spring, it is likely more than one spring force is required. The example below defines a spring force that can be used to connect particles to the world center with a spring constant value of 10.0:

```
<Force id="Fixed" type="Spring">
  <Properties k="10.0" r0="[0, 0, 0]" />
</Force>
```

4.4.3.3 *Viscous force*

To use viscosity define a `Force` node with `type = "Viscous"`. The `Properties` node for a viscous force requires two mandatory attributes; `c` and `u`. The strength of fluid `c` is a real value, and the stream constant `u` is a vector. Below is an example of a heavy underwater viscous force, with a constant undercurrent:

```
<Force id="WildWater" type="Viscous">
  <Properties c="25.0" u="[8, 10, 0.75]" />
</Force>
```

4.4.4 Rigid objects

Rigid objects are as the term indicates not deformable. All rigid objects in the implementation have been rendered motionless and their primary purpose is for elastically deformable objects to have something to collide with, thus they could be called implicit immobile impenetrable rigid objects. The main node for the collection of all rigid objects is the `RigidObjects` node and the only important children must be `Object` nodes. The `Object` node supports several more attributes than the mandatory `id` and `type` attributes. Three optional attributes can be provided, these are `epsilon`, `scale`, and `visible`. The `epsilon` attribute is described in 3.6.1 *Penalty collision force* as the shape potential ε and it is also reused as the strength c of the projection method described in 3.6.2 *Projection*. It is a scalar and its default value is `"1.0"` if not provided. The `scale` attribute is a cosmetic hack, which describes a uniform scaling factor applied to the object when rendered. The scale hack is discussed in 3.9.2 *Particle density*. It is also a scalar and its default value is `"1.0"` if not provided. The last optional attribute is `visible`. It is a Boolean and defines if the object should be rendered or not. The visible attribute does not influence the behavior of an object other than visually.

The possible values of type are `"Cylinder"`, `"Plane"`, `"Sphere"`, and `"Torus"`. Some of the objects are described in greater detail at 4.5.2 *Objects*. The four objects are illustrated in Figure 4.8. Besides the mandatory `Properties` node of each `Object` node, an optional `Bind` node can be applied. It is only materials that can be bound to an object. The properties of the four different object types will be described next.

4.4.4.1 *Cylinder*

An `Object` of `type = "Cylinder"` creates an open-capped cylinder with both caps parallel to the XY-plane. The `Properties` node of the cylinder object takes three mandatory attributes; `c`, `h`, and `r`. The center of the cylinder `c` is a 3D position. The height `h` of the cylinder is a scalar, and the uniform radius `r` of the cylinder is also a scalar. Below is an example of a cylinder object, bound to a color material identified by Paint:

```
<Object id="Pole" type="Cylinder" epsilon="1.0"
        scale="1.0" visible="Yes">
  <Properties r="0.35" h="4.0" c="[2.0, 0.0, 0.0]" />
  <Bind type="Material" ref="Paint" />
</Object>
```

4.4.4.2 *Plane*

A plane `Object` node has `type = "Plane"`. Its `Properties` node requires two mandatory attributes; `n` and `x0`. The surface normal `n` is a vector and the fixed point `x0`, the normal is going through, is a 3D position. Although the rendering of a plane only visualizes a finite area, the plane is infinite upon collision. The example below defines a plane parallel to the XY –plane:

```
<Object id="Floor" type="Plane" visible="Yes">
  <Properties n="[0.0, 0.0, 1.0]" x0="[0.0, 0.0, -4.0]" />
</Object>
```

4.4.4.3 *Sphere*

The sphere `Object` node has `type = "Sphere"`. Its `Properties` node requires two mandatory attributes; `c` and `r`. The center of the sphere `c` is a 3D position and the radius `r` is a scalar. The example below defines a plastic ball with radius 2.0, but due to the scale is 0.9 the visualized sphere will only have a radius of 1.8:

```
<Object id="Ball" type="Sphere"
        scale="0.9" visible="Yes">
  <Properties c="[2.0, -1.0, 8.0]" r="2.0" />
  <Bind type="Material" ref="Plastic" />
</Object>
```

4.4.4.4 *Torus*

An `Object` of `type = "Torus"` creates a torus parallel to the XY-plane. The `Properties` node of the torus object takes three mandatory attributes; `a`, `c`, and `r`. The center of the whole torus object `c` is a 3D position, the outer radius from the center of the hole to the center of the torus tube `r` is a scalar, and the inner radius of the torus tube `a` is a scalar as well. Below is an example of a donut with frosting material:

```
<Object id="Donut" type="Torus">
  <Properties a="0.4" r="1.1" c="[2.0, 0.0, 4.0]" />
  <Bind type="Material" ref="Frosting" />
</Object>
```

4.4.5 Deformable bodies

The `DeformableBodies` node contains the deformable models that will be simulated. The needed children of the `DeformableBodies` node are `Body` nodes. Each `Body` node can use all the nodes we have listed up till now. As a `Body` node has several mandatory and optional child nodes, we will describe this node separately.

4.4.5.1 *The Body node*

The `Body` node itself takes three mandatory attributes; `id`, `model`, and `type`. The `model` attribute categorizes the supported `type` attribute values. The values of the `model` attribute can be `"Curve"`, `"Surface"`, or `"Solid"`. As deformable curves have not been implemented, we will in the following only consider surfaces and solids. When `model = "Surface"` the supported values of type are `"QuadraticBezierPatch"`, `"GenericBezierPatch"`, and `"EllipsoidPatch"`. When `model = "Solid"` the supported values of type are `"LinearBezierSolid"`, `"GenericBezierSolid"`, and `"EllipsoidSolid"`. A `Body` node has three mandatory child types, these are `Properties`, `NaturalShape`, and `Particles`, and two optional child types, which are `InitialShape` and `Bind`.

The `Properties` node has at least one mandatory attribute `timestep`, which is a scalar that defines the simulation step size. If `type` is either `"GenericBezierPatch"` or `"GenericBezierSolid"` another mandatory attribute is needed, which is the `order` attribute. The `order` attribute is an unsigned integer, and it defines the flexibility of the generic model. Please see 4.5.1.1 *Bézier patches* for more details. The `Properties` node also supports three optional attributes; `lcont`, `mcont`, and `ncont`. All three attributes are Booleans and define whether or not the model uses continuous boundaries, although the `lcont` attribute is only available when `model = "Solid"`. The three Booleans all have default value `"No"` if not provided. Below is an example of the `Properties` child node of a deformable Bézier patch of order 2, with a continuous 1st direction boundary:

```
<Properties timestep="0.02" mcont="Yes" order="2" />
```

The `NaturalShape` node defines the model in its rest shape and it only consists of `Node` children. The maximum amount of `Node` nodes is all dependent of the `model` and `type` of the `Body`. A `Node` only has two mandatory attributes; `index` and `pos`. The `index` attribute is an unsigned integer and `pos` is a 3D position. Note `pos` is used differently if `type` is "`EllipsoidPatch`" or "`EllipsoidSolid`". The behavior of the `Node` attributes is described later. The example of the `NaturalShape` node below defines the 4 control nodes of a Bézier patch of order 1 to be a square:

```
<NaturalShape>
  <Node index="0" pos="[-2.0, -2.0, 0.0]" />
  <Node index="1" pos="[2.0, -2.0, 0.0]" />
  <Node index="2" pos="[-2.0, 2.0, 0.0]" />
  <Node index="3" pos="[2.0, 2.0, 0.0]" />
</NaturalShape>
```

The last mandatory child node is the `Particles` node. This node defines the particles in the system and assigns them their properties. The `Particles` node has but one mandatory attribute `grid`, which is a vector of unsigned integers as components. The `grid` vector defines the amount of particles that will be used to discretize the deformable model and the dimension of the vector is equivalent to the dimension of the model, e.g. a 2D deformable surface with `grid="[8,4]"` will consists of 32 particles during the simulation and a 3D deformable solid with `grid="[5,5,5]"` will consists of 125 particles. The `Particles` node has two types of optional child nodes; `All` and `Single`. The `All` node sets the specified properties of all particles at the same time, where a `Single` node set the specified properties for a single particle. One mandatory attribute must be provided on a `Single` node. It is the `particle` attribute that specifies which particle the `Single` node will modify. Both nodes can specify several optional particle attributes. In no specific order these are

- `damping` – a scalar to damp the motion of the particle, default "`0.0`"
- `mass` – a scalar which defines the mass of the particle, default "`1.0`"
- `fixed` – a Boolean which defines if the particle is fixed or not, default "`No`"
- `tension` – a tensor which defines the metric properties η, all entries are default "`0.0`"
- `rigidity` – a tensor which defines the curvature properties ξ, all entries are default "`0.0`"

The dimension of the tensors is also dependent of the deformable model. If `model="Surface"` the tensors are 2×2 matrices, but if `model="Solid"` the `tension` tensor is a 3×3 matrix and `rigidity` is not used. The example below defines a piece of flexible, brisk clothing, which consists of 100 particles and is constrained in two of its adjacent corners:

```
<Particles grid="[10,10]">
  <All mass="5.0" damping="2.0"
       tension="[[1.0, 1.0], [1.0, 1.0]]"
       rigidity="[[0.01, 0.01], [0.01, 0.01]]" />
  <Single fixed="Yes" particle="[0,0]" />
  <Single fixed="Yes" particle="[9,0]" />
</Particles>
```

The two optional child nodes of a body are `InitialShape` and `Bind` respectively. The `InitialShape` node is used exactly as the `NaturalShape` node, but as `NaturalShape` describes the rest shape of the model, `InitialShape` describes its initial starting shape. This is useful when deformation wants to be achieved without external interaction with the system. If the `InitialShape` node is not provided, the system will reuse the rest shape control points as the

initial control points. The `Bind` children are used as described in 4.4.1.3 *Repeated nodes*. There are no restrictions on the bindings of a `Body`.

4.4.5.2 Bézier models

If the `type` attribute of a `Body` node is any of the Bézier models, all `Node` children of the `NaturalShape` and `InitialShape` nodes defines the control points of the given order. The amount of control points needed to successfully define any generic Bézier model of any dimension is

$$(n+1)^d\,, \tag{4.4.2}$$

where n is the order of the Bézier model and d is the dimension. As an example if `model="Surface"` and `type="GenericBezierPatch"` with `order="2"`, the shape nodes will need $(2+1)^2 = 9$ control points. A generic Bézier patch of order 2 is mathematically the same as the `"QuadraticBezierPatch"`.

4.4.5.3 Ellipsoid models

If the `type` attribute of a `Body` node is either `"EllipsoidPatch"` or `"EllipsoidSolid"` only two `Node` children of the `NaturalShape` and `InitialShape` nodes are needed. The two nodes must have `index="0"` and `"1"`. The `Node` with `index="0"` uses the three components of the `pos` attribute to define the a, b, and c parameters of the ellipsoid, which will be discussed shortly The `Node` with `index="1"` uses `pos` as the center of the ellipsoid.

4.5 Examples

In this section we present some examples of the models and objects that we have implemented into the system. First we will describe some examples of the models that can be simulated, and then we will show some of the rigid objects that are implemented.

4.5.1 Models

The models in the implementation are the deformable models that were the actual aim of this project. Previously we have briefly described the implemented models as a generalization of the `DeformableSurface` class. Here we will show three examples. The first two examples will show the generic Bézier patch in action, with discontinuous and continuous boundaries respectively. The third example illustrates an ellipsoid patch.

4.5.1.1 Bézier patches

Bézier patches are a parameterized surface function that can be used to control the initial and natural shapes of deformable surfaces. The examples shown here are done by use of the generic Bézier patch, which represents Bézier patches of any order. The shapes of these patches are defined by control points. The number of control points increases with the order. This means that the higher the order is, the more flexible the surface patch will be. The particles on the surface are linearly interpolated along the coordinate functions.

The most important property of the generic Bézier patch is the order attribute. This attribute controls the flexibility of a surface in its initial and resting state. A piece of cloth can essentially be created using order 1. This restricts the shapes of the surface to be specified only by four control points. These points need not lie in the same plane. In Figure 4.9 is illustrated an example of a patch of cloth that has been created with a generic Bézier patch of order 1. The dots represent the particles that define the surface. The four control points are located at the corner positions of the patch. Another example illustrates the use of continuous boundaries, which indicates that the surface will be cyclic in one or more of its directions. We are going to use a generic Bézier patch

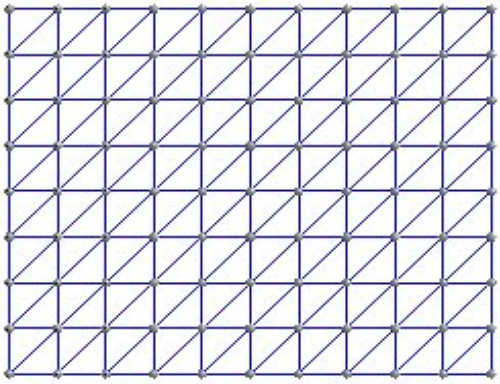


Figure 4.9 A patch of cloth

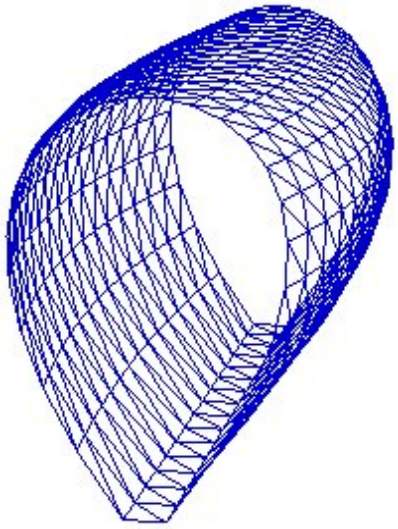


Figure 4.10 A surface of order 3 that is continuous in one direction

just as in the last example. We need to increase the order of the surface, t o gain more control points, to be able to specify the shape as something that is approximately round. If we use a Bézier patch of order 3, we will have 16 control points to define the surface, meaning 4 control points in each coordinate direction. By applying the use of continuous boundaries in one direction, the implementation automatically connects the two edges of the patch that are folded towards each other. In Figure 4.10 we have illustrated an example of a pipe with the described properties. To get an understanding of how to form the pipe, we degrade the surface to a cross-section of only four control points. The result is depicted on Figure 4.11. The cross-section is a curve where the end control points p_0 and p_3 will always lie on the curve, whereas the points in between, p_1 and p_2, are forming the actual shape of the curve. In [9] is the area of Bézier patches discussed in great detail, but for a more direct approach ready to be implemented look into [19].

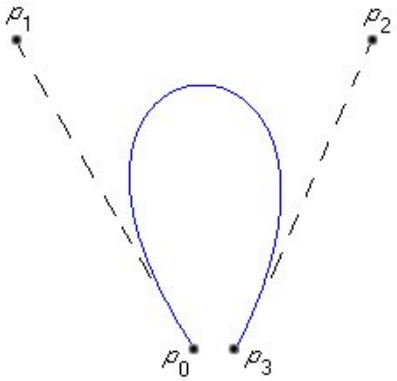


Figure 4.11 Cross section of the pipe, showing a standard Bézier curve of order 3

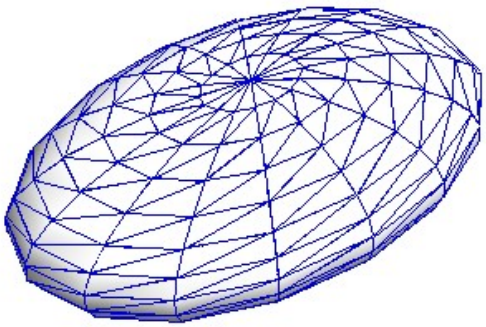


Figure 4.12 An ellipsoid patch

4.5.1.2 Ellipsoid patches

Another model which is possible to use in the implementation is an ellipsoid patch. This type of object is described by the equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1\,, \tag{4.5.1}$$

where the parameters a, b, and c are the lengths of the three semi-axes spreading orthogonally from the center of the ellipsoid. The parametric functions of the ellipsoid are

$$\begin{aligned} x &= a\cos\theta\sin\phi \\ y &= b\sin\theta\sin\phi \\ z &= c\cos\phi, \end{aligned} \tag{4.5.2}$$

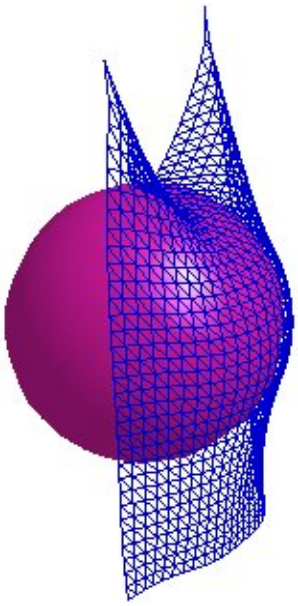


Figure 4.13 Hanging cloth colliding with a sphere object

where θ is the azimuth angle lying in the interval $[0, 2\pi)$, and ϕ is the polar angle in the interval $[0, \pi]$. In Figure 4.12 an ellipsoid is depicted. It is defined by the values $a = 1.5$, $b = 1.0$, and $c = 0.5$. In the situation where $a = b = c$, we will have a perfect sphere.

4.5.2 Objects

The objects in the implementation are the rigid objects, which primarily are used to simulate collision with the deformable models, and thus create interesting scene interactions. These objects are completely motionless but are useful as obstacles. The implementation of the objects is based on implicit functions, see 3.6.1.1 *Implicit surfaces*, and new kinds of objects can be implemented fairly easily. Here we will present two of the implemented objects and the math that is needed. Three mathematical functions are needed for an implementation of a new object in the framework; the implicit function describing the surface of an object, its gradient function, and a distance function, which should return the closest distance from any point to the surface of the object. Any new rigid object must inherit from the abstract base class `EDMObject`, which requires the 3 functions listed above to be implemented.

4.5.2.1 Sphere

A sphere is a simple mathematical object that is described by the equation

$$\left(x - c_x\right)^2 + \left(y - c_y\right)^2 + \left(z - c_z\right)^2 = r^2 , \tag{4.5.3}$$

where $\mathbf{c} = \left(c_x, c_y, c_z\right)$ is the center and r is the radius. The implicit function for the sphere object can be described as

$$f(\mathbf{p}) = |\mathbf{p} - \mathbf{c}|_2^2 - r^2 = 0 , \tag{4.5.4}$$

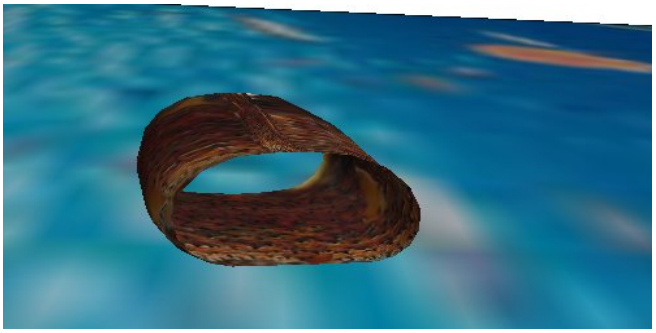


Figure 4.14 Sea cucumber colliding with plane

where $\mathbf{p} = (x, y, z)$ is any point to be verified. The gradient of the object function is

$$\nabla f(\mathbf{p}) = 2(\mathbf{p} - \mathbf{c}), \tag{4.5.5}$$

and the distance function is

$$d(\mathbf{p}) = \left| \|\mathbf{p} - \mathbf{c}|_2 - r \right|. \tag{4.5.6}$$

With these functions supplied to the sphere object, the only thing missing is a visualization function. For an insight into how this visualization is done for the sphere and how the class for this object is structured, consult the source files "`edm_sphere.h`" and "`edm_sphere.cpp`". An example of the object in use is depicted on Figure 4.13, where the sphere is colliding with the cloth.

4.5.2.2 Plane

A mathematical plane through the point $\mathbf{p}_0 = (x_0, y_0, z_0)$ with normal $\mathbf{n} = (n_x, n_y, n_z)$ is described by the equation

$$n_x x + n_y y + n_z z + d = 0\,, \tag{4.5.7}$$

where $d = -n_x x_0 - n_y y_0 - n_z z_0$. The implicit object function could then be described as

$$f(\mathbf{p}) = \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0\,, \tag{4.5.8}$$

and the gradient of the object function is simply

$$\nabla f(\mathbf{p}) = \mathbf{n}\,. \tag{4.5.9}$$

Finally the distance function describing the distance to the plane from any point is

$$d(\mathbf{p}) = \frac{\left|\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0)\right|}{|\mathbf{n}|_2}. \tag{4.5.10}$$

The source files "`edm_plane.h`" and "`edm_plane.cpp`" implement the described plane object. We have put the plane into use in the example on Figure 4.14, which illustrates a sea cucumber on the button of the deep pacific.

5 Comparison

We have been through all the mathematics and physics required to understand and get an idea of how to implement the method to simulate elastically deformable models. Further more, we have walked the reader through all the practically steps concerning the method itself, and also given the schematics of our implementation. Our work is a contribution to the ongoing OpenTissue project [6], which already provides a method to simulate soft and deformable bodies. It is now time to justify all the hard work that has been put into this project, and we are eager to get an idea of how convincing and practical this method actually is. In this chapter we will perform a few internal tests, while comparing our work against the existing method in OpenTissue. The comparisons will not be theoretical in nature, although we could possibly attract physicists by focusing on the energy levels of each and every particle in the system, and compare them against real world data and research from the field of material science. However, we are not engineers, but simply computer scientists that want to create convincing interactive animations from physics-based simulation. We will instead focus on areas that appeal more to our line of work and understanding. The preparation and maintenance of the systems will be considered, along with how realistic both methods can simulate the provided models, taking accuracy and stability into account. The last thing we are interested in is performance and memory usage. Next, we will present the opponent system, followed by the different cases of comparison.

5.1 Standard of Reference

The standard of reference is between the method of elastic deformation by Terzopoulos et al. [21] and the stick constraint system by Jakobsen [13]. In the rest of this chapter we will reference the two systems primarily by the names of their authors. Both methods are based on particle systems, and support simulation of cloth and soft bodies. The biggest difference between the methods is that constraints are stiff and explicitly given in Jakobsen, whereas constraints work like springs and are implicitly given in Terzopoulos. The constraint solver in Jakobsen is based on an explicit relaxation scheme, but in Terzopoulos an implicit solver scheme is employed.

5.1.1 Constraint types

We suggest that the reader pay attention to [13] for details about the constraint method. In this section we will describe how to use the explicit constraints to simulate the same types of constraints that are used in [21].

The finite difference operators in Terzopoulos applied in the computation of the discrete constitutive functions (3.2.3) and (3.2.4), actually provide enough information to design good approximating constraints to use with Jakobsen.

5.1.1.1 Alpha functions

The α_{ij} functions use elements from the 1st fundamental form. When $i = j$ the difference operators describe the length between adjacent particles, and when $i \neq j$ we get a description of shearing. Considering all particles ordered in some grid, like the discrete grid from Terzopoulos, the length constraints are defined as the stick constraints between adjacent particles parallel to the grid axes, and the shear constraints are defined as the stick constraints between a particle and its neighbor's opposite neighbor. Figure 5.1 illustrates the possible ways to create stick constraints to support the same constraints modeled from the α_{ij} functions. The particle surrounded by a circle, is the particle we are looking at compared to Terzopoulos. A dotted line indicates redundancy as this constraint is not necessary to model the desired effect. It is equivalent to the symmetric tensors in Terzopoulos, where the strictly upper triangular matrix is redundant.

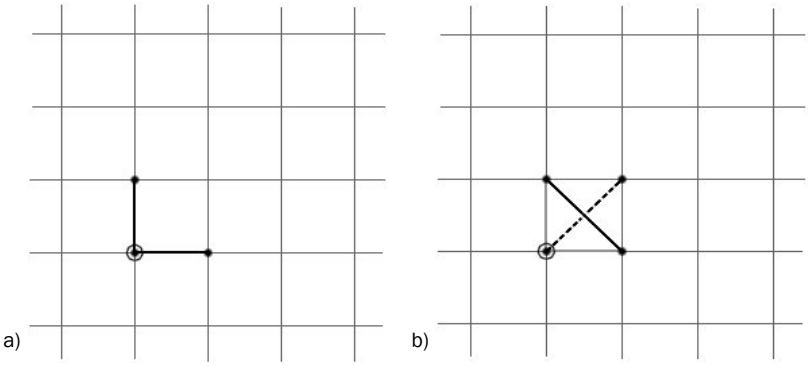


Figure 5.1 Alpha type constraints on a 2D grid - a dotted line indicates it is not directly necessary **a)** length constraints **b)** shear constraints

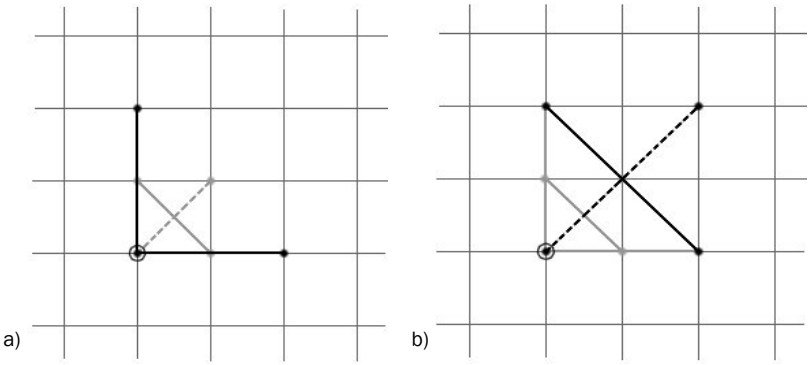


Figure 5.2 Beta type constraints on a 2D grid – the alpha constraints are preserved in a dimmed tone for reference purpose **a)** bend constraints **b)** twist constraints

5.1.1.2 Beta functions

The β_{ij} functions use elements from the 2nd fundamental form. When $i = j$ the difference operators describe the bending between particles, and when $i \neq j$, we get a description of the twist. The bend constraints are defined as the stick constraints between neighbors' neighbors parallel to the grid axes, and the twist constraints work in the diagonal within the same area. The possible ways to create stick constraints that support the same constraints modeled from the β_{ij} functions is depicted in Figure 5.2. However, since the bend and twist constraints differ more from the β_{ij} functions, than length and shear constraints differ from the α_{ij} functions, in practice, we must extend the amount and range of the beta constraints if we want to make an object more rigid in Jakobsen.

5.1.2 Comparison models

One thing we must bear in mind is that Terzopoulos is an elastic system in nature, where Jakobsen becomes more and more elastic as more particles are introduced into the system. Jakobsen is

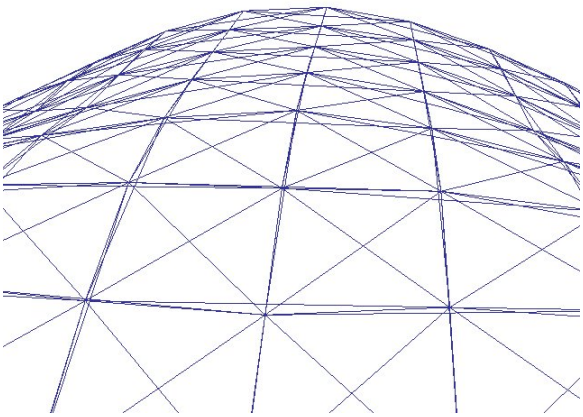


Figure 5.3 The Jakobsen cloth model. The cloth collides with an invisible object to better reveal the constraints

also able to become totally stiff, by connecting each particle to all other particles using stick constraints. Terzopoulos can never become completely stiff, but using large values for both tension and rigidity will do a decent job.

We will use two types of models for the tests and comparisons. These models will be used for both systems. We will set up the simulation configurations equally, to secure as fair a comparison as possible, e.g. we use the same timestep for the same simulation, and the same mass and damping for all particles. When some properties of a system cannot be changed freely, we will attempt to accomplish the same settings for the opponent system. The models we have chosen to focus on for the comparisons are a piece of cloth and a thin metal plate. Cloth simulations are always interesting to compare, and in one of the comparisons we have chosen to compare the cloth ability within both systems, focusing at details such as folds and flexibility. The thin metal plate will be used to compare the ability to use strong rigidity for both systems. Next we will describe how the models have been created for both systems.

5.1.2.1 Cloth

The cloth model is a standard quadratic surface patch, with an equal amount of particles in both coordinate directions. Because size does matter in Terzopoulos, the cloth model will only stretch out 4.0 units, in the world coordinate system, for both directions. The Terzopoulos cloth model is created using a generic Bézier patch of order 1. The 4 control points required to model the cloth, we have chosen as

$$\begin{aligned} p_0 &= \big(-2.0,\, -2.0,\, 4.0\big) \\ p_1 &= \big(-2.0,\, 2.0,\, 4.0\big) \\ p_2 &= \big(2.0,\, -2.0,\, 4.0\big) \\ p_3 &= \big(2.0,\, 2.0,\, 4.0\big), \end{aligned} \tag{5.1.1}$$

where the $z = 4.0$ is just a convincing initial starting point above different obstacles. For the particle properties we have chosen the following values

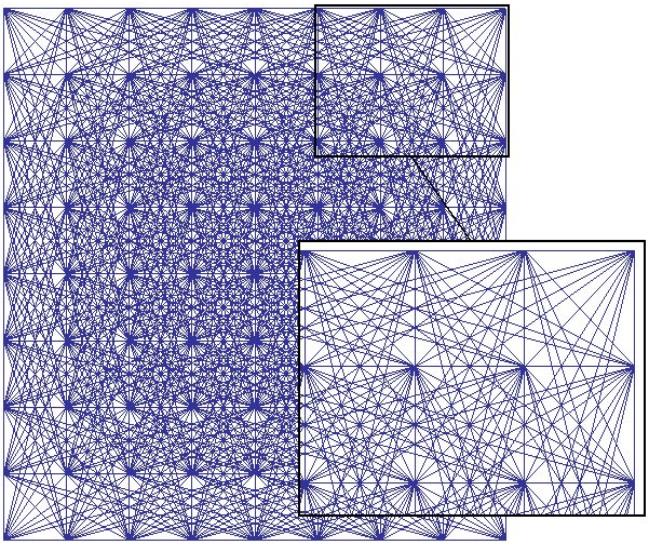


Figure 5.4 The Jakobsen thin metal plate model

$$\mu = 1.0, \quad \gamma = 0.85, \quad \boldsymbol{\eta} = \begin{bmatrix} 0.35 & 0.25 \\ 0.25 & 0.35 \end{bmatrix}, \quad and \quad \boldsymbol{\xi} = \begin{bmatrix} 7.5 \cdot 10^{-5} & 7.5 \cdot 10^{-4} \\ 7.5 \cdot 10^{-4} & 7.5 \cdot 10^{-5} \end{bmatrix}. \tag{5.1.2}$$

The cloth model in Jakobsen uses a default rigidity factor of 2, corresponding to a full use of the alpha constraints and the bending constraint of the beta types. The default particle properties are the same as Terzopoulos. The Jakobsen cloth model in action is illustrated in Figure 5.3.

5.1.2.2 Thin metal plate

The thin metal plate is modeled exactly like the cloth model, except the rigidity has been increased a great deal. For Terzopoulos this means that the tension and rigidity tensors for all particles will be

$$\boldsymbol{\eta} = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \quad and \quad \boldsymbol{\xi} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}, \tag{5.1.3}$$

and for Jakobsen the cloth type has been modified to use a default rigidity factor of 5. In Figure 5.4 the thin metal plate model is illustrated for Jakobsen, where the huge amount of constraints necessary to achieve the needed rigidity is revealed.

5.2 Comparison Cases

We have tested and compared the two systems with focus on several issues. Our findings and results of these comparisons are presented below in no prioritized order. As stated in the introduction of this chapter we have not been too theoretical, thus the presentations of the findings will vary in detail. Some results are pure observations, while others are based on hard facts.

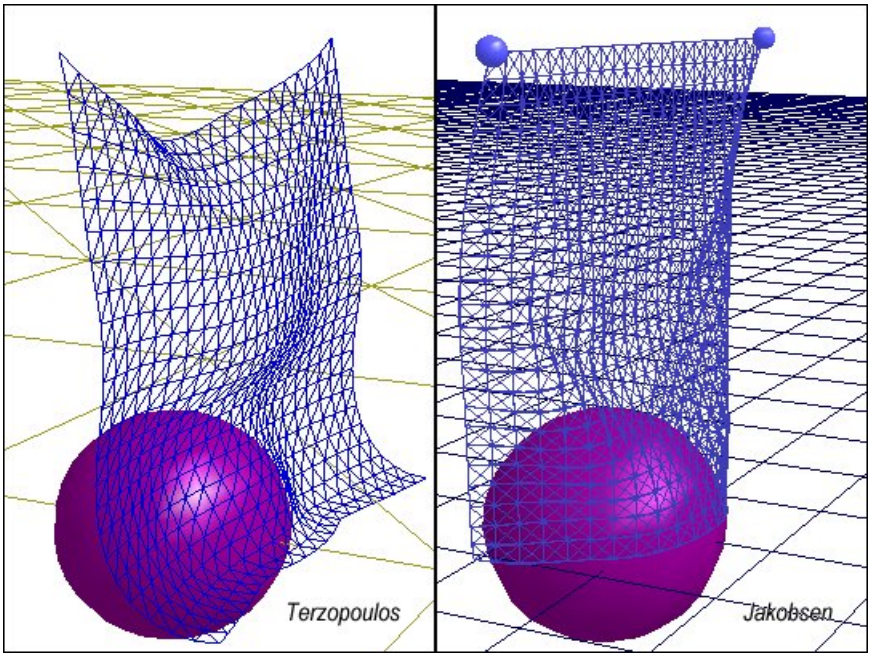


Figure 5.5 Synchronized frames from the same motion sequence simulated by Terzopoulos and Jacobsen

5.2.1 Maintenance

If a programmer starts from scratch and wants to develop a system to simulate deformable soft bodies, then some issues should be considered. Jakobsen is easily understood and easily implemented. However, hard tedious work is required to prepare simulations with Jakobsen. Assuming we have no functionality to create simulations for either system and the whole simulation must be set up manually. All particles must be created with their respective properties. This is actually enough to start a simulation using Terzopoulos, as all constraints are created implicitly from the discrete particles. The tension and rigidity tensors control the behavior of the constraints in Terzopoulos, and also the material property of the body. Even though our implementation creates both tensors for each particle, as the original model [21] states, we always use the same tension and rigidity values for every particle. This indicates that a global instance of each tensor is enough, and will not only make the creation faster, but also save memory. The creation of Jakobsen requires another process. All constraints must be created explicitly by the user or programmer. As a minimum all adjacent particles must be connected by length constraints to secure the correct shape of the body is maintained. Shear constraints are normally also required to secure that the body does not collapse on itself. The only possible way to control the material property of a body using Jakobsen is by controlling the creation of the constraints; which particles that should be connected and how many constraints that is required. Automating this process is not an easy task. OpenTissue uses a traverse algorithm that connects particles according to a rigidity factor. All though the process will get the job done, it does not always utilize a suitable connection strategy. As an example the cloth we simulate using Jakobsen has been created automatically using a hard coded rigidity value of 2, which makes the cloth a little too stiff. Modifying the algorithm to use a rigidity value of 1 will just make the cloth too floppy instead. The more rigid a body should be, the more constraints are required in Jakobsen, and thus memory usage is directly involved as a function of the material properties. The memory usage is kept sufficient and bounded in Terzopoulos

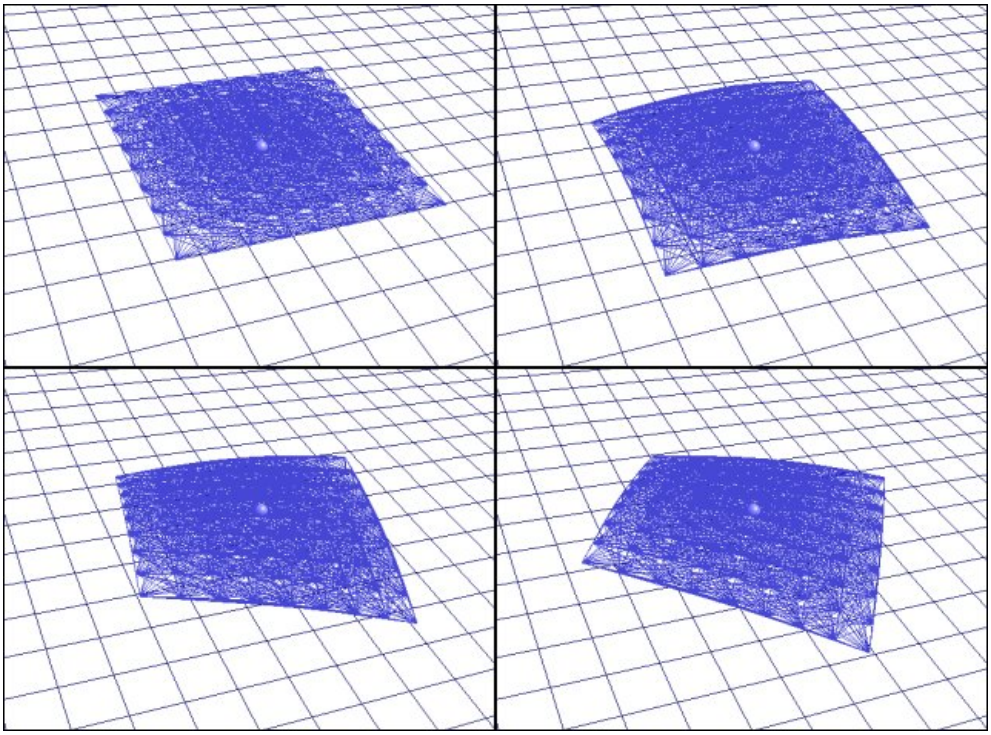


Figure 5.6 Stability issues in Jakobsen's constraint relaxation method

as changing the material property of a body, is a matter of changing some values that are already located in memory.

5.2.2 Accuracy

The accuracy tests will be looking at different kinds of details for the same configuration simulated on both systems. We have used the cloth model of 576 particles (24×24), and constrained the two corner particles to a fixed position. A plane and a half sphere placed under the cloth will be used for colliding. Gravity is applied to the system.

To compare the degrees of details of the systems, we have been focusing at how the cloth wrinkles and folds like cloths do in real life. Figure 5.5 shows a single pair of synchronized frames of the same simulation from both systems. The cloth is rendered in wire frame to reveal subtle details. The rest of the synchronized frame pairs are located in A.1 *Comparison Material*. It is worth mentioning Jakobsen is stiffer in nature than Terzopoulos, which is more elastic. This is why the cloth on Terzopoulos stretches and collides with the ground. However, it is also worth mentioning that Terzopoulos is much smoother and more flexible than Jakobsen, which is somewhat obvious counting the folds appearing and disappearing.

5.2.3 Stability

It is important to know about possible stability issues before implementing a new system. Some stability problems might affect precisely the kind of simulations a new system is developed for. We have focused on two types of problems that cause unstable systems. For the stability tests and comparisons we have used the thin metal plate model.

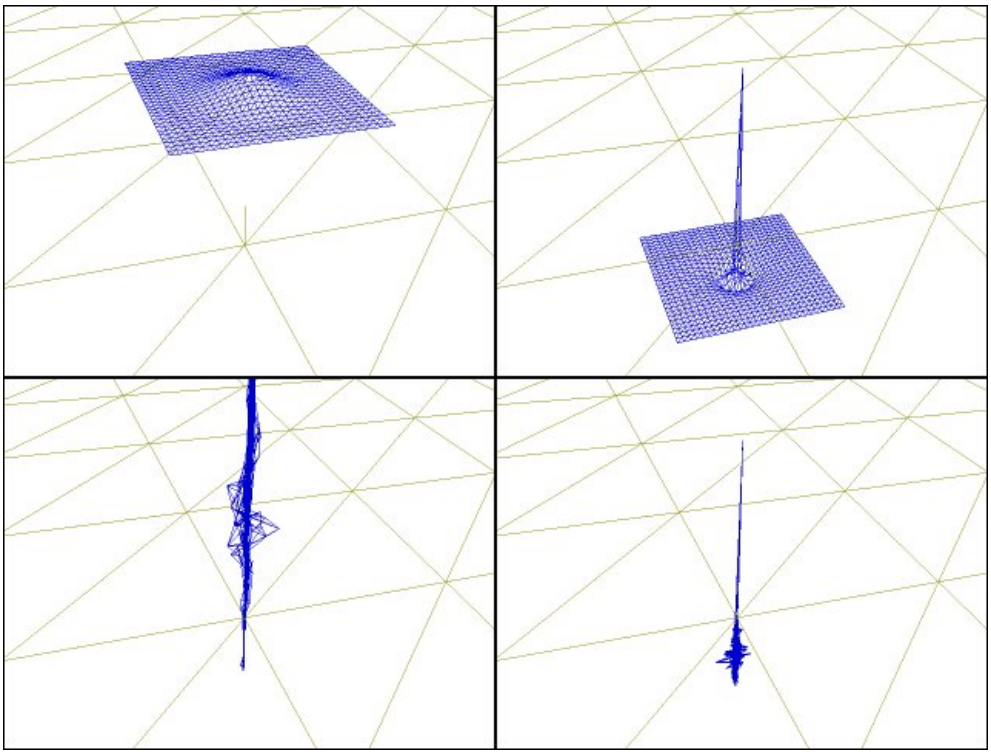


Figure 5.7 Stability issues in Terzopoulos using particle rich models with too large a timestep

We make sure that the thin metal plate is created with the same odd amount of particles in both coordinate directions. We end up with a particle in the center of the plate that will be constrained to its initial position. The plate is influenced by the force of gravity.

The first problem that we encounter is with Jakobsen. Because the constraints are solved explicitly in turn using a relaxation procedure, and because the whole system mass is distributed from the center particle and outwards, we will get a system that never converges. Why the relaxation method behaves like this can be explained, but the issue is beyond the scope of this paper. We will settle for the observation that the thin metal plate continues to rotate about the fixed center particle in Jakobsen, whereas the plate will fall into equilibrium in Terzopoulos. On Figure 5.6 we have illustrated 4 sequences from the Jakobsen system simulating the thin metal plate scene, which rotates clockwise about the center. The whole animation is called "`jakobsen_stability_wireframe.avi`" and it is available on the supplemental CD [3].

Jakobsen is not the only system that tends to behave badly. The next stability issue concerns Terzopoulos, the semi-implicit integration method, and too large timesteps. The last simulation used a timestep of $\Delta t = 0.01$ and Terzopoulos did not suffer from that. Extending the amount of particles from 9×9 to 29×29 and executing the simulation using the same timestep, does not look good for Terzopoulos, as the 4 still sequences in Figure 5.7 can confirm. The simulation of the thin metal plate explodes. This type of behavior is an exact proof that the semi-implicit integration method is not stable in all situations. When Terzopoulos becomes too large in different situations, the semi-implicit integration becomes unstable, and the system will diverge. Fortunately the instability can be solved quickly by reducing the timestep. For the thin metal plate model we

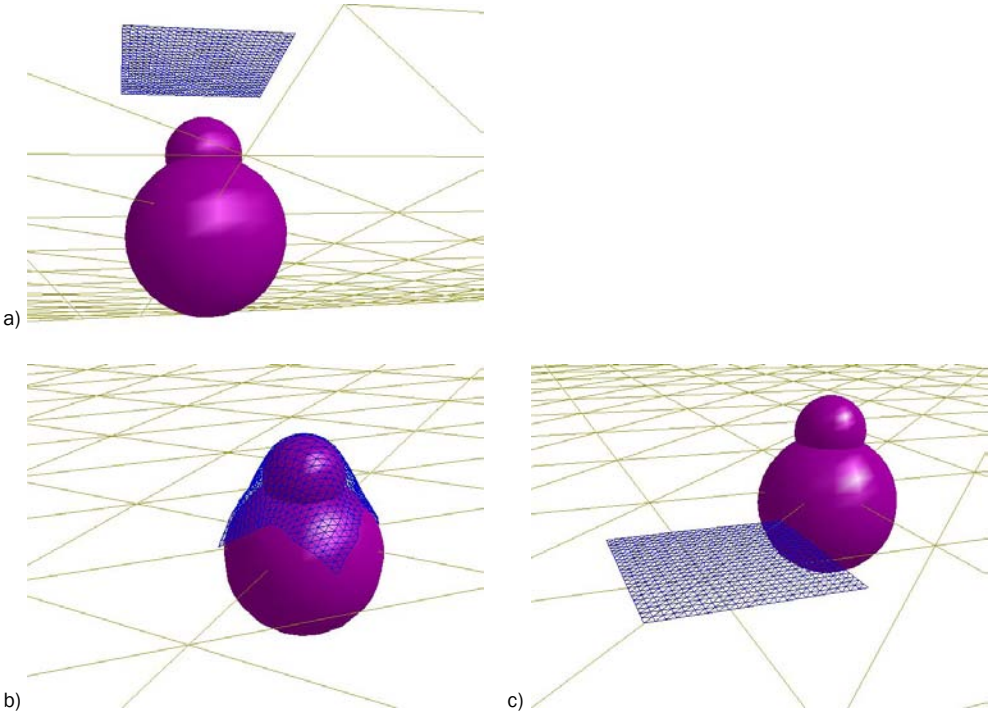


Figure 5.8 The configuration layout from the performance comparison, illustrating a Terzopoulos cloth of 24x24 particles **a)** initial starting state **b)** main collision state **c)** end resting state

successfully made the system stable by reducing the timestep with a factor of 10, thus setting $\Delta t = 0.001$.

5.2.4 Performance

The performance comparison is probably what interest most readers working with realtime animation, in contrast to accuracy and stability, which interest readers doing simulation. We know that constraint relaxation method using Verlet integration by Jakobsen [13], is one of the fastest methods to simulate cloth. Having that knowledge in mind, it will be an interesting task to monitor how well Terzopoulos will do against Jakobsen. We will only compare the runtime cloth simulation steps for Terzopoulos and Jakobsen. The rendering and visualization cost will not contribute to the frame computation, which is the fairest way to monitor performance, as rendering strategies can be implemented in many ways. The cloth model will be used again for these tests. Initially, it will be placed just above 2 spheres and a plane going through the bottom sphere. The system will be influenced by gravity, thus the cloth will collide with the spheres, and later slide aside on the floor away from the spheres. Finally it will fall to rest, due to damping and viscosity. The chosen scene configuration will show 3 different situations during the performance comparison:

1. The cloth model will fall unaffected from above the spheres, thus no collision will occur, and no deformation should occur, Figure 5.8a
2. The cloth model will collide with the spheres and deforms, Figure 5.8b
3. The cloth model will straighten out on the ground, to its rest shape, with equal collision weight on all particles, Figure 5.8c

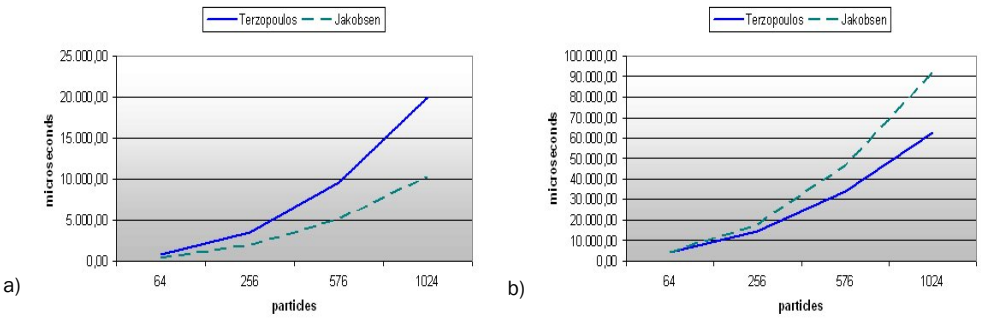


Figure 5.9 Connections between the average frame computation for Terzopoulos and Jakobsen **a)** 10 iterations **b)** 100 iterations

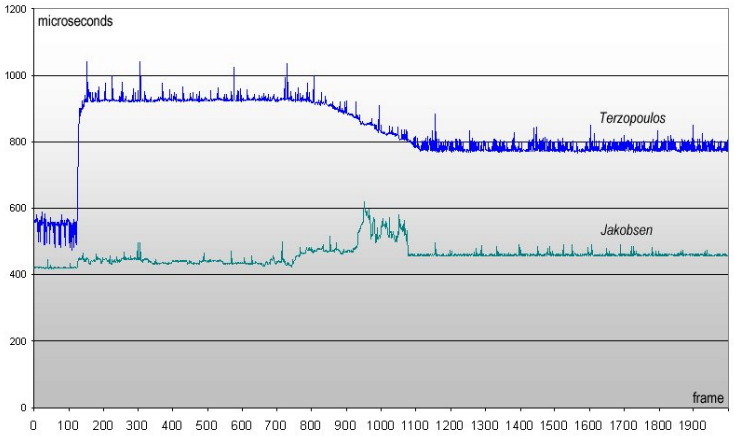


Figure 5.10 Terzopoulos and Jakobsen computation times of 2000 frames, simulating a 8x8 particle cloth using 10 iterations

Both systems possess the same two kinds of parameters that can be tweaked individually to compare the frame computation costs under different influences. The two parameters are the total amount of particles of a body, and the total amount of iterations used to solve the constraints by relaxation in Jakobsen, or the total amount of iterations used to solve the linear equations by the conjugate gradient method in Terzopoulos. We will vary this iteration parameter between 10 and 100 iterations, and the total amount particles will vary between 64, 256, 576, and 1024. The first 2000 frames will be monitored and recorded for analysis. The performance simulations are executed on a Pentium4 3GHz processor with 1GB of RAM running Microsoft WindowsXP operating system.

In Table 5.1 we have listed the average frame time calculations of the 2000 frames, from the 8 performance simulations. We have plotted the frame computation time onto 8 charts for easy visual comparison. All 8 charts are available in A.1 *Comparison Material*, and a motion sequence illustrating the scene can be found on the supplemental CD [3] under the file name `"jt_performance_wireframe.avi"`. On Figure 5.9 the two charts show the connection between the average frame times for 10 and 100 iterations respectively.

Particles	Iterations	Terzopoulos Avg. frame time	Jakobsen Avg. frame time
64 (8x8)	10	812.92 μs	448.83 μs
256 (16x16)	10	3,465.01 μs	2,014.58 μs
576 (24x24)	10	9,462.11 μs	5,141.24 μs
1024 (32x32)	10	19,916.83 μs	10,231.95 μs
64 (8x8)	100	4,032.82 μs	3,958.61 μs
256 (16x16)	100	14,466.96 μs	17,582.00 μs
576 (24x24)	100	33,832.97 μs	46,410.69 μs
1024 (32x32)	100	62,623.00 μs	91,484.99 μs

Table 5.1 The average frame calculation time for Terzopoulos and Jakobsen respectively, using 4 different particle configuration, and 2 different values for the number of iterations used in the different system solvers

Focusing on a single chart enables us to explain the appearance of the graphs for both systems. Figure 5.10 depicts the chart from the simulation of 8x8 particles using 10 iterations. The graph that describe the frame calculations of Terzopoulos is the most prominent to consider, as the 3 steps are very conspicuous. From frame 0 to frame 123 the model falls under gravity, thus performs a non-deformable rigid body motion. From frame 124 the model collides with the spheres, and about frame 800 the cloth begins to slide down on the floor. At approximate frame 1115 the cloth has slid all down on the ground and all particles are equally influenced by the plane collision, as the cloth slides away from the spheres until it is at rest. The reason the oscillation is distinctively larger for Terzopoulos than Jakobsen, when the sphere collisions occur, is due to the fact that the Jakobsen cloth is much more rigid than Terzopoulos, and thus not as many particles collide. The elastic cloth from Terzopoulos results in sphere collision for nearly all particles.

One final note about the chart on Figure 5.10, and the rest of the charts in the appendix, concerns the rising peeks that are emanating from the graphs. The peeks are unnatural and do not have anything to do with the actual simulation procedure, nor the frame computation cost. It is simply the unfortunate consequence of performing the simulations on the Microsoft Windows platform, as we are not guaranteed to be the only process in need of CPU power. We do not wish to blame Microsoft Windows in general, as we do not know for sure, if the pattern is repeated on other platforms.

6 Results

It is time to show the results that we have achieved during this project. A visual conviction is very powerful, and primarily we will show a number of visual examples that both will serve as test results, and as an illustration of the strengths of the implementation. We will close the chapter with an evaluation of our work and the method of elastically deformable models.

6.1 Visual Test Results

This section will present the results that we have been able to achieve. We will start off by testing some of the basic fundamental ideas behind the method [21], and continue with the showing of different situations from our implementation. The majority of the figures from this section originate from motion sequences generated from the graphical output of our application. All motion sequences are available on the supplemental CD [3].

6.1.1 Fundamental principles

The most important principles of the method are the abilities to simulate the tension and curvature constraints, and thus being able to restore a deformed body to its natural rest shape. Before we use the deformable models to animate exciting bodies, we must convince the reader that these fundamental principles work in the implementation.

6.1.1.1 Tension

We start by testing the metric constraints, using only the tension tensor. We disregard all external forces, choosing only to focus on the metrics. The best way to test the metric mechanics is to only affect the tension of the surface. To accomplish this, we define a small square surface of 7×7 particles and stretch all 4 corners away from the center. The displacement of the corner particles is performed in the same plane as the surface. The mass, damping, and tension tensor are defined as

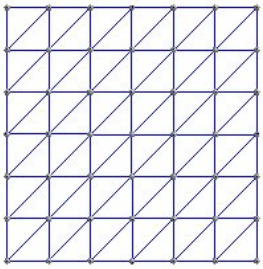
$$\mu = 5.0, \quad \gamma = 50.0, \quad \boldsymbol{\eta} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}, \tag{6.1.1}$$

for all 49 particles. Upon simulation start, the surface should shrink to its natural rest shape, which we have depicted on Figure 6.1a. As no external forces have been applied to the system, and all particles can move freely, within constraints, we want the shrinking of the surface to be similar in all corners, thus the center particle should stay in its initial position. Some of the interesting simulation steps are depicted on Figure 6.1b, which also indicates that our desired result seems to be achieved.

6.1.1.2 Rigidity

The next thing that we want to test is the curvature constraints. This implies that we only use the rigidity tensor. Again, all external forces are disregard to concentrate on the curvature alone. Unfortunately, we are only able to flatten the surface, not able to make it more curved, as discussed in 3.2.3 *Curvature*. To verify that a curved surface can be flattened with initial positive and negative curvatures, we reuse the square surface of 7×7 particles, but we now increate the control points at the 4 corners and the control point at the center upwards, along the positive z-axis. The mass, damping, and rigidity tensor are defined as

a)



b)

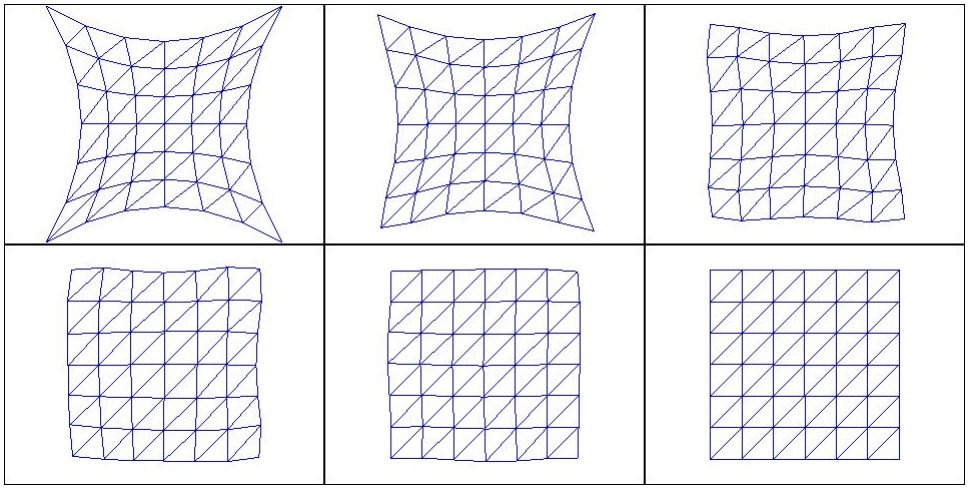


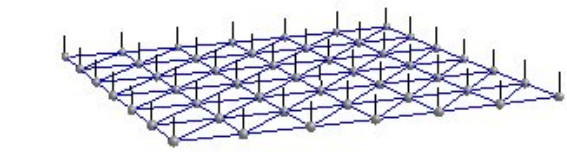
Figure 6.1 The tension constraint in action a) The natural rest shape of the surface b) 6 still frames from a motion sequence illustrating the tension constraint. The first frame is the initial deformed shape of the body

$$\mu = 5.0, \quad \gamma = 50.0, \quad \boldsymbol{\xi} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}, \tag{6.1.2}$$

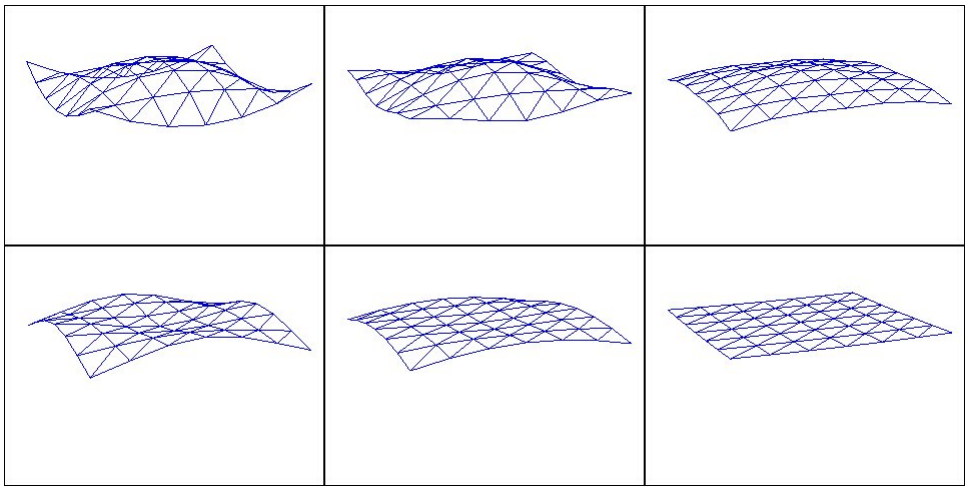
for all 49 particles. Upon simulation start, the surface should flatten towards its natural rest shape, which we have depicted on Figure 6.2a. As the curvature mechanics is somewhat more powerful than the metrics, we would expect the surface to behave like a damped oscillator in the most curved places, due to the elastic spring effect. The surface should stay in its initial place on average. Some of the highlights from the simulation test are depicted on Figure 6.2b. We believe that the behavior of the surface comply with our desired result.

6.1.1.3 Surface crumples

We believe that the asymmetric properties of the elasticity stencil contribute to the crumple features in the deformable surfaces. The crumples only arise at two of the opposite corners on the surface and are smoothed out across the surface diagonal. The crumple feature seems to have a lot in common with the curvature instabilities. On Figure 6.3a we have illustrated a piece of cloth affected by gravity and constrained in the 4 corners. The system has zero rigidity, thus the components of the rigidity tensor is zero, $\xi_{ij} = 0$, which introduce the crumple features. The paradox is

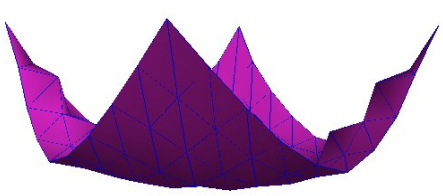


a)

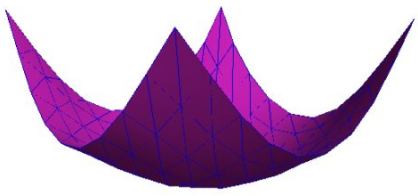


b)

Figure 6.2 The rigidity constraint in action **a)** The rest shape of the surface **b)** 6 still frames from a motion sequence illustrating the rigidity constraint. The first frame is the initial deformed shape of the body



a)



b)

Figure 6.3 Surface crumples **a)** Huge crumples appear when no rigidity is introduced to the system **b)** The result of applying a modest rigidity tensor to the surface, which eliminates the crumples

that a natural solution to the crumple problem is to introduce rigidity into the system. Just a modest value of the rigidity tensor, $\xi_{ij} > 0$, straightens out the surface crumples, as Figure 6.3b is the result of.

6.1.2 Deformable open surfaces

It is time to show all the strengths of the system. In this section we will visualize deformable open 2D-manifolds in different situations with different material properties. Only a few still frames will



Figure 6.4 A high resolution heavy curtain is sagging over a rigid sphere

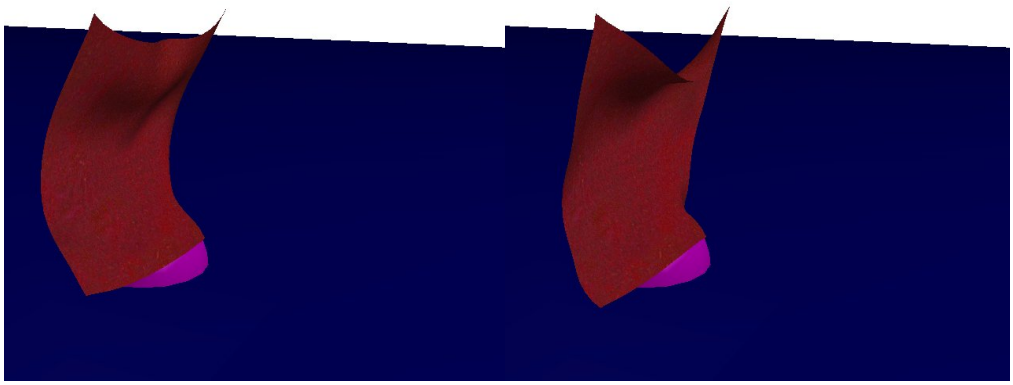


Figure 6.5 A tablecloth that renders smooth folds upon collision

be shown here for some of the examples. The rest of the frames from the motion sequences are available in A.2 *Results Gallaria*. Scene configurations are enclosed on the supplemental CD [3].

Figure 6.4 shows a curtain consisting of 1024 particles sagging over a rigid sphere. The curtain has been rendered resistant to metric deformation, and also made rather heavy to obtain a nice simulation of the sagging effect, upon collision with the sphere. The external force contribution is only provided by gravity. As no viscosity is applied to the system, the curtain will slowly slide down the sphere.

Figure 6.5 shows a tablecloth of 576 particles. It is constrained in the corners that lie on the topmost edge. The scene is just a modified version of the heavy curtain, making it more elastic and lighter than the curtain. Gravity pulls the tablecloth downwards, which stretches and collides with the sphere in the floor. The collision results in smooth folds in the cloth, which looks convincing.

Figure 6.6 shows a rubber flap with springs attached to each corner. We obtain the rubber property by using a high value for the tension, and a modest value of rigidity, while using a small damping factor for each particle.

Figure 6.7 shows a waving flag with its right edge constrained to the flag pole. The blowing wind is modeled using the viscous force, with the constant stream velocity different from zero.

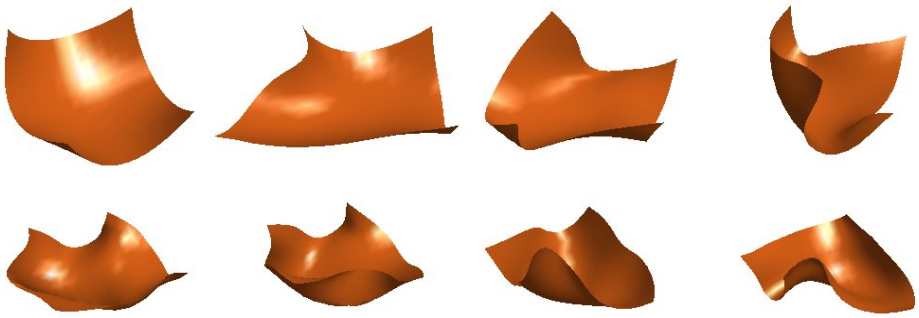


Figure 6.6 A rubber flap attached by springs is pulled around

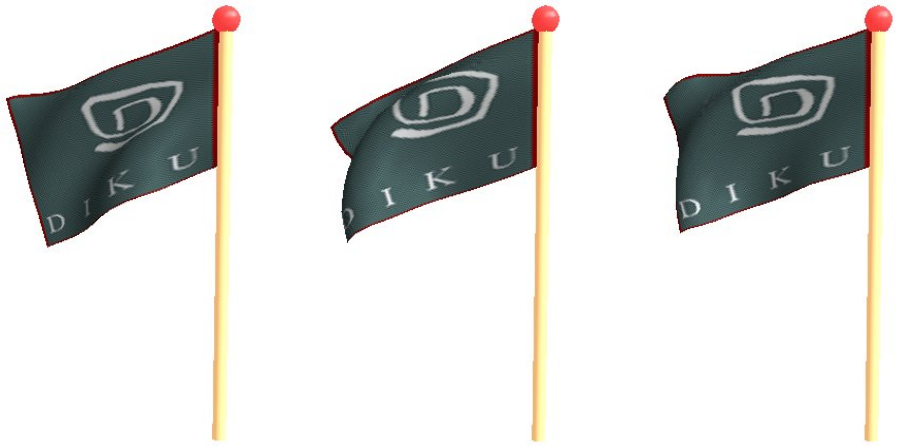


Figure 6.7 A waving flag, using the viscosity to model the wind force

6.1.3 Deformable semi-closed surfaces

We continue with surfaces that use continuous boundaries.

Figure 6.8 shows a thin elastic membrane, which has been blown up to surround a rigid jack. We use a high damping value compared to the mass, and no rigidity is used at all. Upon simulation start, the membrane will wrap itself around the jack as tight as is possible. We then pull one of the open ends away from the jack, which cause the membrane to slowly slide off the knots. High detailed rubber folds appear under the escape. The membrane model is based on the Ellipsoid patch and it consists of 1024 particles.

Figure 6.9 shows a very soft sea cucumber, which initially has tangled itself into two balls on the bottom of the deep pacific. It slides loose and sinks to the bottom. Someone is teasing it by pulling it by its rear end. The sea cucumber model is a generic Bézier patch with continuous boundaries. The particles are quite light in mass to enable the collaboration between tension and rigidity in maintaining the hollow shape of the Echinodermata.

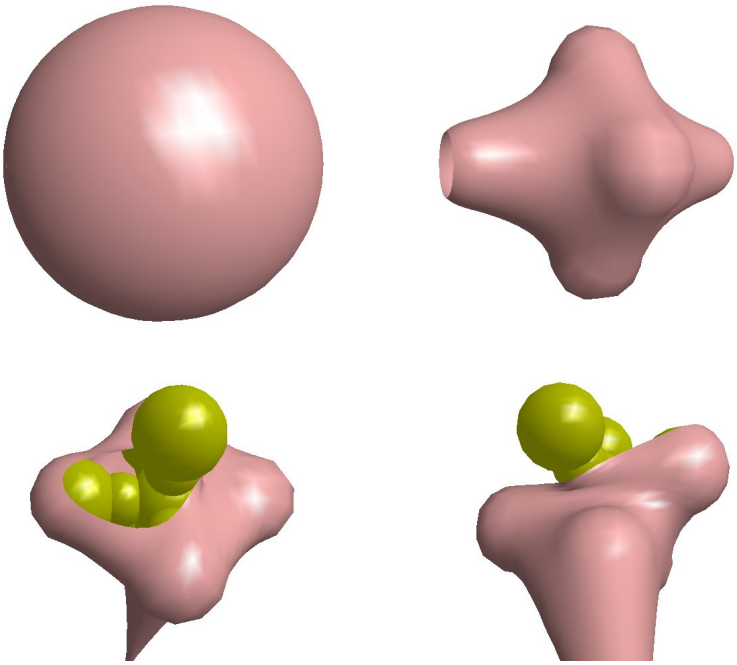


Figure 6.8 A thin open membrane is wrapping around a rigid jack. It slowly slides around the knots, as we are pulling it away from the jack



Figure 6.9 A sea cucumber playing around on the bottom of the pacific

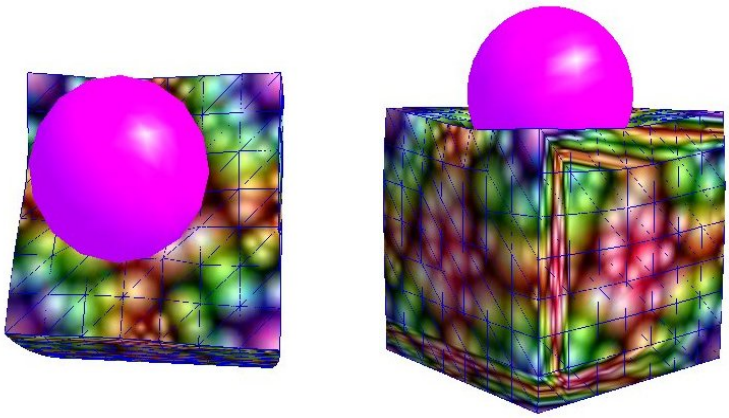


Figure 6.10 A rigid sphere resting on a deformable solid

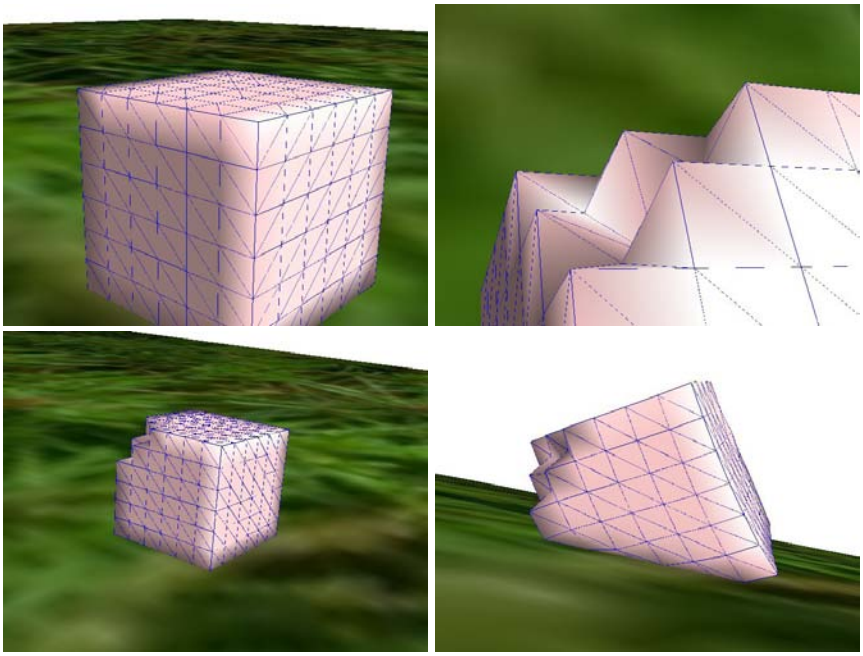


Figure 6.11 A pudding cube collapsing on a lawn upon concentrated deformation

6.1.4 Deformable Solids

In this section we illustrate the implementation of deformable solids. Although we assume the foundation of the model is a natural extension of the given model of deformable surfaces, we can observe strange happenings under different deformations.

Figure 6.10 shows a rigid sphere resting on a deformable solid. As all rigid objects are motionless, we have cheated a little to end up with the shown situation. The initial deformable cube is

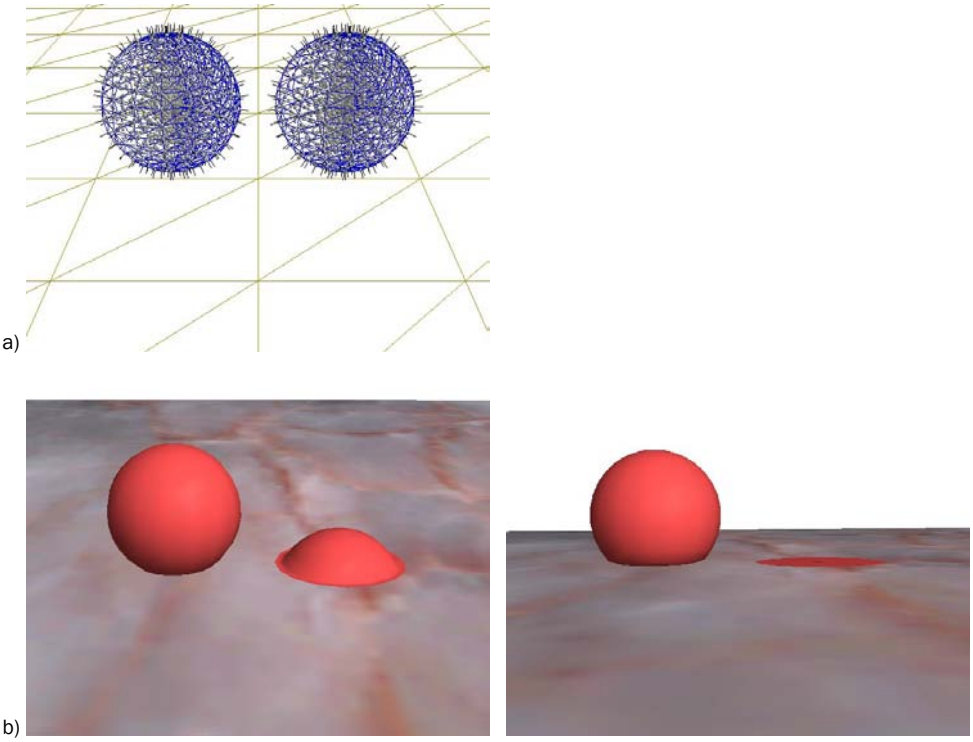


Figure 6.12 Rubber balls **a)** The particles are visualized to show that the balls are in fact solids **b)** The right ball is more heavy then the left, and the metrics fails to keep the integrity, thus the ball collapses on itself into 2 dimensions

squeezed under the sphere, and the whole bottom has been fixed. Upon simulation start, the cube will stretch out and deform when colliding with the sphere.

Figure 6.11 shows a highly elastic cube, when in motion has a similar behavior as pudding (on a lawn?!). The most obvious problem with deformable solids is also illustrated. We are able to push the corners inside the solid, making the solid collapse slowly. The solid is completely unable to recover from the deformation. We believe this is a real problem, as the continuation of the collapsing results in elimination of an entire dimension.

Figure 6.12 shows two rubber balls made of the ellipsoid solid model. The ball to the right has been rendered heavier than the left ball, and the same problem as in the above example, shows clearly. The integrity of the solid collapses in on itself and in the end becomes one dimension shorter.

6.2 Evaluation

Based on the different results obtained throughout the project, this section will contain an evaluation of our work.

6.2.1 Physical properties

As the elastically deformable models [21] are deeply anchored in the fundamental physical theory of elasticity, we think that it would be interesting to look at real-life materials, such as fabrics. The visual results from the heavy curtain example from the last section bring forward some nice prop-



Figure 6.13 A home-made photo of the real-life fabric Viscose, colliding with a bowl placed upside-down

erties from the physics-based model. We have tried to find some real-life fabric that is similar to the virtual fabric, depicted on Figure 6.4. We did come up with a sample of viscose fabric, and have tried to stage an arrangement that reminds of the collision situation in the figures. On Figure 6.13 we have shown a home made photo, which depicts the viscose fabric colliding with a spherical bowl. The size of the folds on Figure 6.4 is larger than the folds on the real-life cloth, which is all dependent on the fabric materials. We did not create the heavy curtain animation to simulate the viscose fabric. As a matter of fact the process was quite the opposite. The real-life cloth was assembled to simulate the virtual scene. The folds on the virtual curtain seem to have the same characteristics as the real-life viscose cloth.

6.2.2 Contribution

From 5 *Comparison* we can sum up the results and evaluate the whole comparison. The main question in focus was if our implementation would contribute anything to the OpenTissue project. This question will now be answered.

Like the Terzopoulos system, the Jakobsen system also runs interactively, but with a decrease in performance as the rigidity of the system rise. As both systems decrease in performance when more particles are introduced, this argument is disregarded. Technically, our implementation is asymptotically faster than Jakobsen, as a function of the amount of iterations. For 10 iterations Jakobsen was approximately twice as fast as Terzopoulos, but with 100 iterations we saw the opposite pattern. If we had increased the amount of iterations of yet another factor 10, we would, without a doubt, have seen that Jakobsen would deviate from Terzopoulos with more than a constant.

The process of changing material properties is very easily performed in Terzopoulos, whereas constraints either require being added or removed in Jakobsen to scale the rigidity. To introduce a more elastic system in Jakobsen, a new type of constraint must be added, or a simple hack must be applied, by using a very small amount of iterations, with the price of accuracy leaks.

Three reasons can justify why the method of elastically deformable models should become a part of OpenTissue:

- The system provides a high level of details, while still being able to run interactively
- Many different types of materials can be simulated easily, just by changing the parameters of tension and rigidity tensors
- The native elasticity of the model enables natural simulation of skin and tissue

The last thing about the contribution, that should be mentioned, is that Terzopoulos is relevant even today, although it originated almost two centuries ago. The facts are that the elastically deformable models are still based on a correct physical model, and the two centuries have provided us with a massive increase in computational power, which enables the method to run smoothly and interactively on a computer of today's standard.

6.2.3 The process

As we reach the end of this project we are finally able to take a step back and take a look at the process as a whole. Going in, we had only modest experience with the field of physics-based animation. The subject of deformable objects contained a number of mathematical areas, such as differential geometry and calculus of variations, which we needed to understand prior to the implementation phase. During the first half of the project period, we were able to gain a reasonable insight into these areas, making it possible to understand the deformable model put forward by Terzopoulos et al. During the implementation we experienced a number of pitfalls, such as choosing the wrong integration scheme, which resulted in a decrease in stability of the model. This could sometimes create results that were hard to interpret, in the sense that it was hard to know whether the problem was in our implementation or with the model itself. We have also learned a lot from taking the wrong turns, and we have gained a deeper knowledge of the problems that can arise in solving physical models.

7 Future Work

In this section we will talk about some of the possible future work that lies with the implementation of this project. Some of the things are actual work in progress at the time of writing, whereas other things are ideas for future extensions of the implementation.

7.1 Work in Progress

Some areas we define as important, and are currently looking into, are mentioned in this section.

7.1.1 Fixing curvature problems

In 3.2 *Elasticity* it was mentioned that problems arise when a deformable surface is specified as having a curved resting shape. This problem was never discussed in the original article about elastically deformable models [21]. The reason for this problem could be many, but two issues seem more plausible. Firstly, the derivation of the curvature term done in the original article, when trying to find the minimizing term, contains some generalizations to make the curvature term more like the tension term. It could be interesting to go back and take a second look at the calculus of variations done in this step, to see if some information of interest to this problem could have been thrown away. Secondly, this problem could simply be of a more fundamental nature with regards to the formulation of curvature through the second fundamental form and the discretization of it.

7.1.2 Instabilities in deformable solids

Although this project has mainly concerned itself with deformable surfaces, some efforts have also been put into the project to generalize the system into deformable solids. This can also be verified when looking in 4.1.1 *Class hierarchy*. When conducting experiments with deformable solids a weakness was noticed concerning the stability of the model. This weakness comes from the fact that solids only utilize tension constraints. The tension constraints present in the current model formulation for solids can only resist extremely modest deformations before collapsing. We would like to look into a new formulation of the model that is better suited to handle arbitrary deformations. Another issue regarding deformable solids was presented in 6.1.4 *Deformable Solids* where it was possible to provoke folds on the boundary corners of an elastic cube. This is not supposed to happen, and we are currently trying to formulate an improved model that can handle this problem.

7.1.3 Self intersection

It is possible for a surface in the current implementation to self intersect in any way possible. For more realistic animations of cloth, self intersection handling is required. We are currently looking into using methods already implemented in the OpenTissue framework to achieve this effect for the deformable objects.

7.1.4 Interface of the implementation

The current implementation is dependant on the user to supply a scene description through the XML scene configuration. We would like to extend the interface part to something that is more easily generalized into other types of user input. This will make the implementation easier to integrate into existing OpenTissue projects. For example a rigid body simulator that already exists in OpenTissue, could be integrated with our implementation, replacing the immobile rigid objects, and thus introducing more interesting scene interactions.

7.2 Ideas

We also have a few ideas for future extensions, some of them are briefly presented next.

7.2.1 Collision between deformable models

An interesting extension would be to implement collisions between two or more deformable objects. This could for example be implemented by use of AABB (axis aligned bounding box) trees.

7.2.2 Hybrid model

This project is based on the article from 1987 [21]. The authors have since done further work in the area of deformable models. This work was concentrated, among other things, on introducing new effects of non rigid objects into the original model which was of a perfectly elastic nature. Since hardly any real world objects are perfectly elastic they introduced inelastic deformation, using things such as viscoelasticity and plasticity, and creating what is called a hybrid model [20]. This extended model also incorporates fracturing of deformable objects, which could certainly be of interest since the OpenTissue project was originally created to be used as tools for creating a surgery simulator. The model of fracture could be used as a way of penetrating human skin. The deformable objects that we have worked with throughout this paper are very elastic in nature. A method that improves the motion of elastically deformable objects by integrating rigid body mechanics is introduced in [22].

7.2.3 Generic mesh structure

The method that is currently used for the discretization of a deformable surface is a rectangular grid structure with evenly spaced particles. This does not invite the use of more complex geometries, such as mesh structures of real life objects. It could be interesting to be able to somehow apply the deformable model to arbitrary surface meshes. If a surface mesh could somehow be unwrapped onto the rectangular grid, the finite differences of the deformable model could fairly easily be applied. Another approach could be to create an algorithm that could analyze a surface mesh, and for every particle find out what its next and previous particle were. This should be done for both coordinate directions as is currently done in the parameterized surface model. The finite differences of the deformable model could still be applied using a new flexible data structure, and thus introduce import of meshes from 3rd party software.

8 Conclusion

The project is at its end, and we have completed an implementation of the method of elastically deformable models, based on the theory and understandings of the model. The implementation runs interactively, with more than 1024 particles, and can be used to simulate different types of elastic surfaces and solids.

During the project we have, among other things, gained knowledge of new mathematical areas, such as differential geometry. Such knowledge will be very useful in our further studies of physics-based animation and simulation. We have addressed several problems, which appeared during the implementation, and even corrected problems found in the article by Terzopoulos et al.

A comparison of our implementation was made with a fast relaxation based particle system. The comparison showed that the elastically deformable models are a reasonable performance contender in the field of cloth and soft body simulation. When it comes to the level of details in the simulations, the method of elastically deformable models is superior.

Although tiny issues still exist in the final implementation, the motion sequences produced with our implementation were of such quality that they have been presented in public. With the addition of our work in progress, we believe the elastically deformable models will be a gratifying contribution to the OpenTissue project.

References

[1] ATLAS Homepage (2004), *"Automatically Tuned Linear Algebra Software (ATLAS)"* http://math-atlas.sourceforge.net/

[2] J. Bloomenthal, C. Bajaj, J. Blinn, M. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill (1997), *"Introduction to implicit surfaces"*, Morgan Kaufmann Publishers, Inc.

[3] M. K. Christensen and A. Fleron (2004), *"Implementation of Deformable Objects"*, Enclosed supplementary CD, Department of Computer Science, University of Copenhagen, DIKU

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf (2000), *"Computational Geometry: Algorithms and Applications"*, Second edition, Springer-Verlag

[5] D. Eberly (2003), *"Derivative Approximation by Finite Differences"*, Magic Software, Inc., January 21, 2003

[6] K. Erleben, H. Dohlmann, J. Sporring, and K. Henriksen (2003) *"The OpenTissue project"*, Department of Computer Science, University of Copenhagen, DIKU, November 2003 http://www.diku.dk/forskning/image/research/opentissue/

[7] K. Erleben, J. Sporring, H. Dohlmann, and K. Henriksen (2003) *"Simulation of Deformable Objects"*, Lecture Notes, Department of Computer Science, University of Copenhagen, DIKU, Spring 2003

[8] R. Fedkiw and S. Osher (2003), *"Level Set Methods and Dynamic Implicit Surfaces"*, Springer-Verlag

[9] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes (1996), *"Computer Graphics, Principles and Practice"*, Second edition in C, Addison-Wesley

[10] H. Goldstein, C. P. Poole, and J. L. Safko (2002), *"Classical Mechanics"*, Third Edition, Addision-Wesley

[11] E. B. Hansen (19??), *"Variationsregning"*, Polyteknisk Forlag (in Danish)

[12] M. T. Heath (2002), *"Scientific Computing, An Introductory Survey"*, Second Edition, McGraw-Hill

[13] T. Jakobsen (2001), *"Advanced Character Physics"*, Proceedings, Game Developer's Conference 2001, San Jose
http://www.ioi.dk/Homepages/thomasj/publications/gdc2001.htm

[14] J. J. Koenderink (1990), *"Solid Shape"*, MIT Press

[15] T. Larsson and T. Akenine-Möller (2001), *"Collision Detection for Continuously Deforming Bodies"*, In Eurographics 2001, pp. 325-333, 2001

[16] T. Möller (1997), *"A Fast Triangle-Triangle Intersection Test"*, Journal of Graphics Tools 2(2), pp. 25-30, 1997

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2002), *"Numerical Recipes in C++: The Art of Scientific Computing"*, Second edition, Cambridge University Press

[18] S. Redon, A. Kheddar, and S. Coquillart (2002), *"Fast Continuous Collision Detection between Rigid Bodies"*, in Proceedings of Eurographics, September 2002

[19] M. Skinner (2001), *"Bezier Patches"*, GameDev.net online article
http://www.gamedev.net/reference/articles/article1584.asp

[20] D. Terzopoulos and K. Fleischer (1988), *"Modeling inelastic deformation: Viscoelasticity, Plasticity, Fracture"*, Computer Graphics, volume 22, Number 4, August 1988, pp 269-278

[21] D. Terzopoulos, J. C. Platt, A. H. Barr, and K. Fleischer (1987), *"Elastically deformable models"*, Computer Graphics, volume 21, Number 4, July 1987, pp 205-214

[22] D. Terzopoulos and A. Witkin (1988), *"Physically-based Models with Rigid and Deformable Components"*, Craphics Interface '88, Jule 6-10 1988, pp 146-154

[23] G. van den Bergen (1998), *"Efficient Collision Detection of Complex Deformable Models using AABB Trees"*, Journal of Graphics Tools 2(4), pp. 1–14, 1998

[24] L. Verlet (1967), *"Computer Experiments on Classical Fluids I: Thermodynamical Properties of Lennard-Jones Molecules"* Phys. Rev., vol. 159, pp. 98-103, 1967

[25] A. Watt and M. Watt (1992), *"Advanced Animation and Rendering Techniques, Theory and Practice"*, Addison-Wesley

[26] E. W. Weisstein (2004), *"Arc Length"*, MathWorld, A Wolfram Web Resource
http://mathworld.wolfram.com/ArcLength.html

[27] E. W. Weisstein (2004), *"Manifold"*, MathWorld, A Wolfram Web Resource
http://mathworld.wolfram.com/Manifold.html

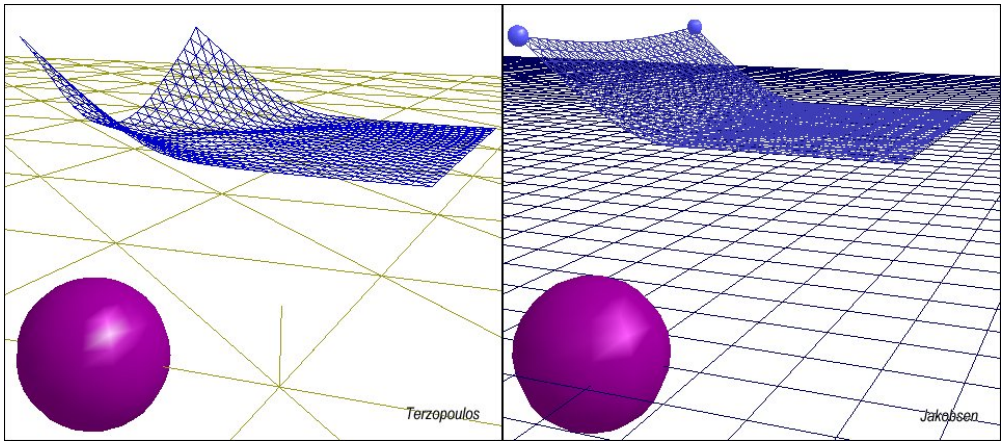
[28] D. R. Williams (2001), *"Planetary Fact Sheets"*, NASA Goddard Space Flight Center, National Space Science Data Center
http://nssdc.gsfc.nasa.gov/planetary/planetfact.html

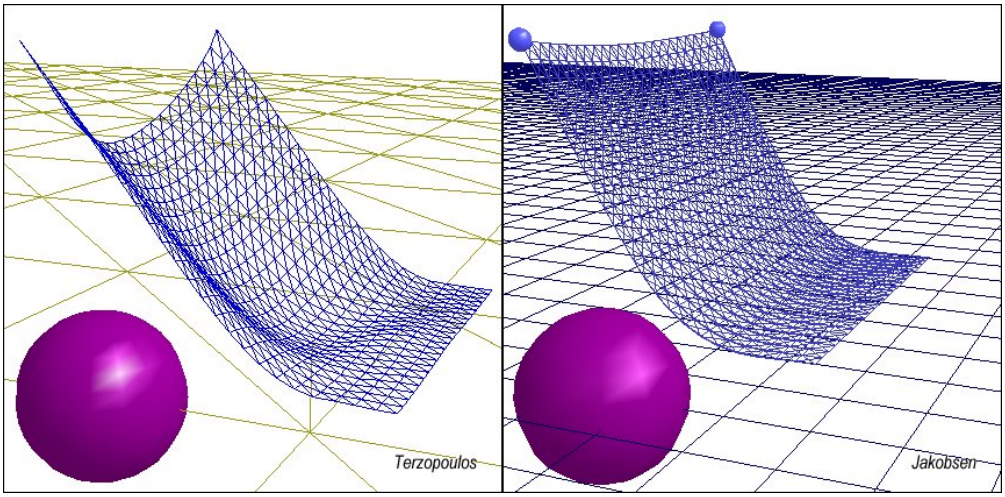
Appendices

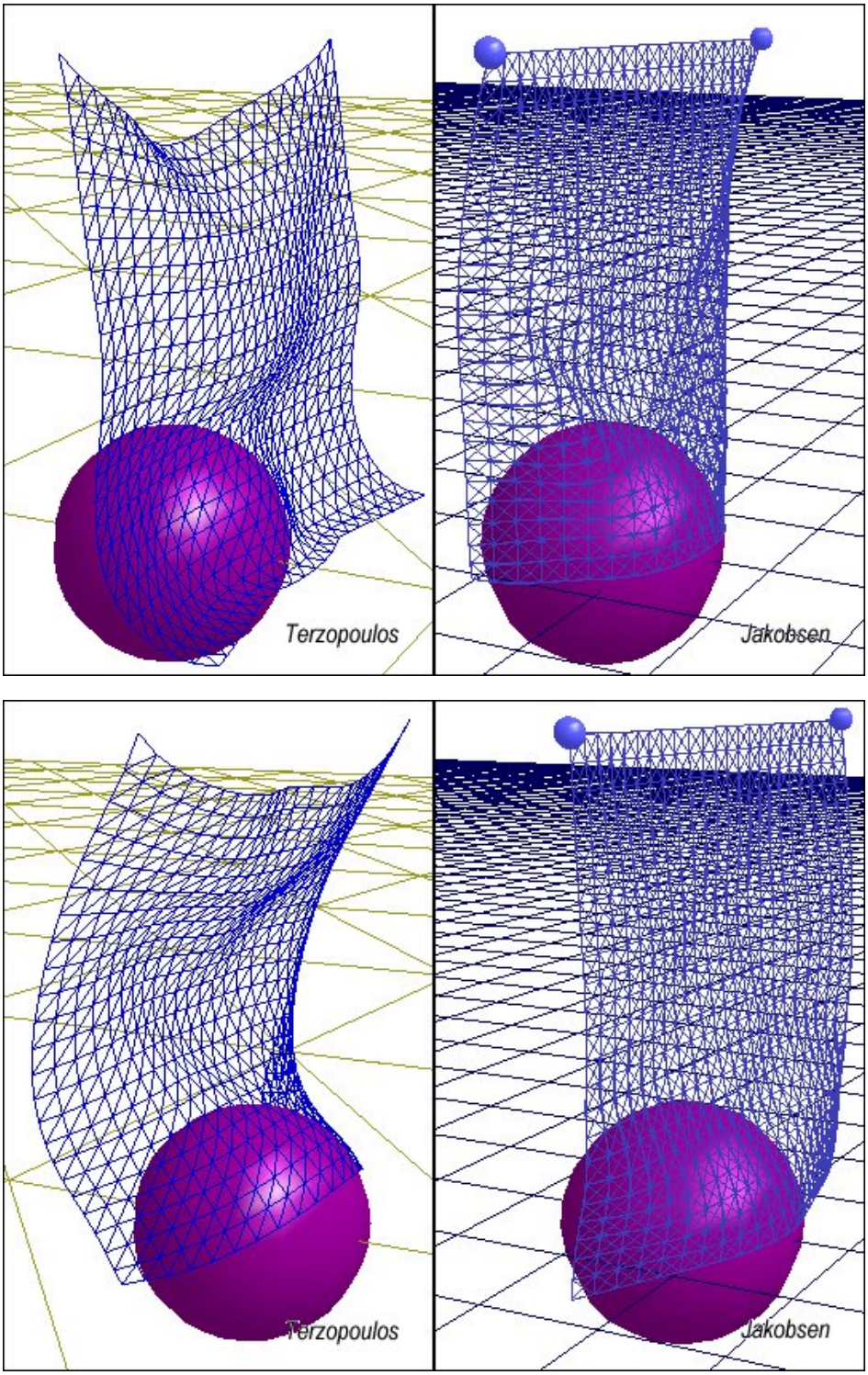
A.1 Comparison Material

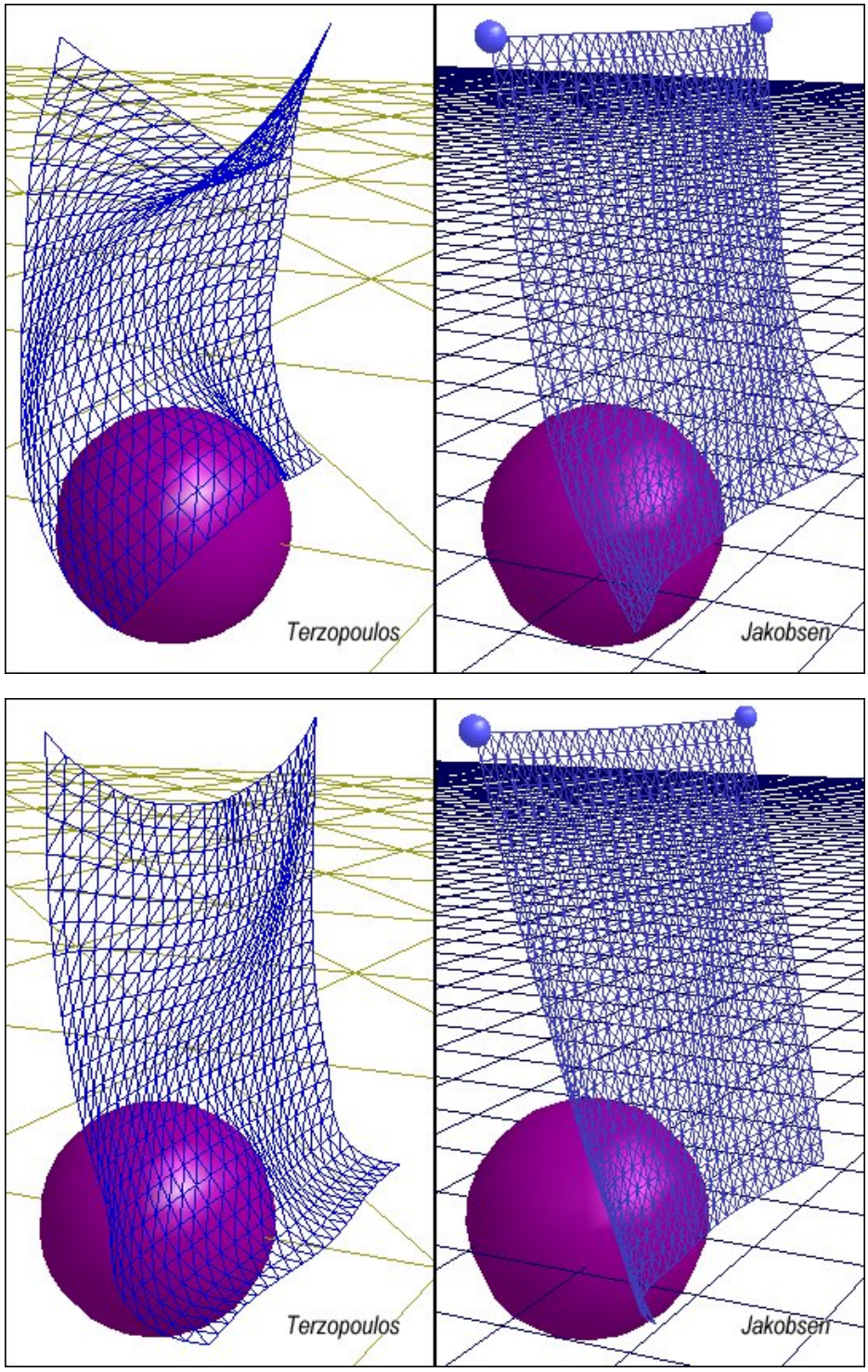
Accuracy test

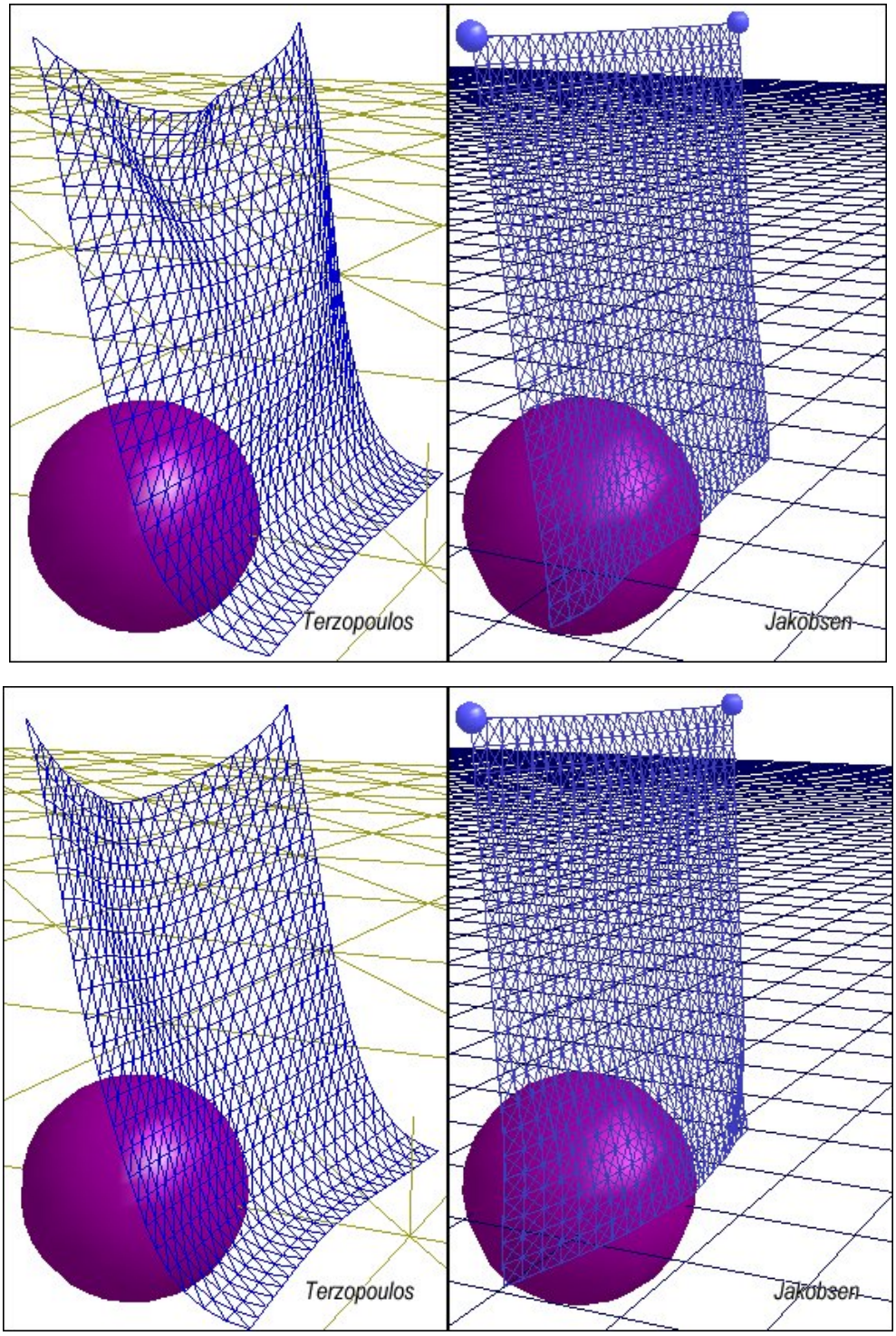
Below are the 8 pairs of still frames as referenced from 5.2.2 *Accuracy*. The frames arise from 2 synchronized motion sequences, and are aligned for easy comparison. The frames, as they appear below, are in chronologic order with respect to time. The cloth is rendered in wire frame to compel hidden details. Both animations can be found on the supplemental CD [3].





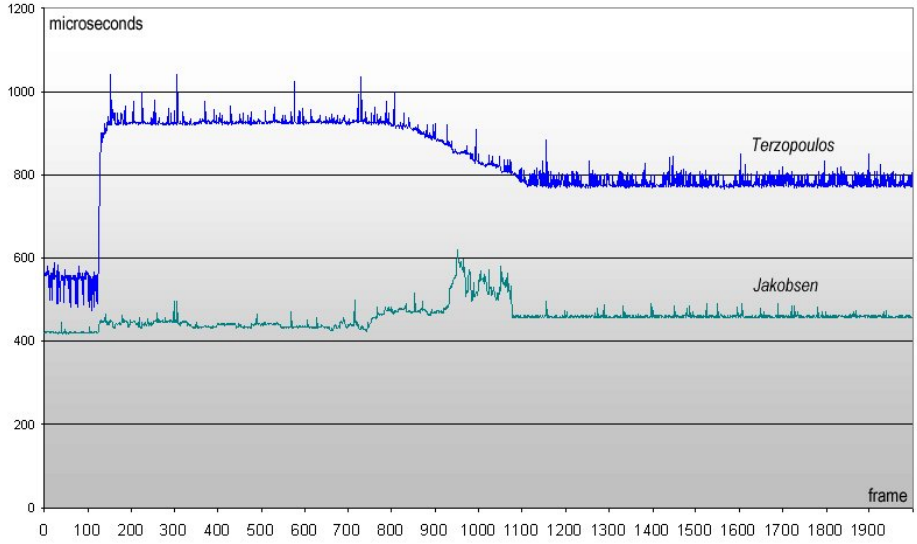

Terzopoulos
Jakobsen
Terzopoulos
Jakobsen


Terzopoulos
Jakobsen
Terzopoulos
Jakobsen

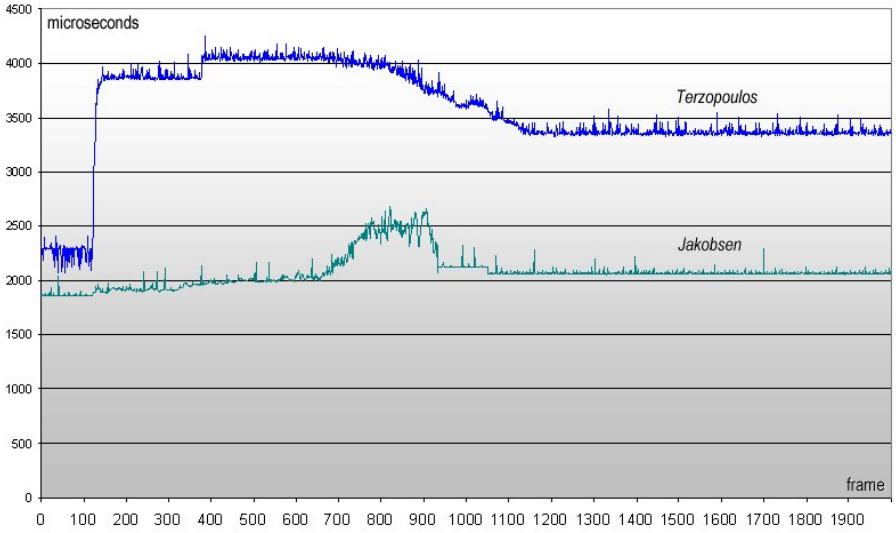

Terzopoulos
Jakobsen
Terzopoulos
Jakobsen

Performance comparison

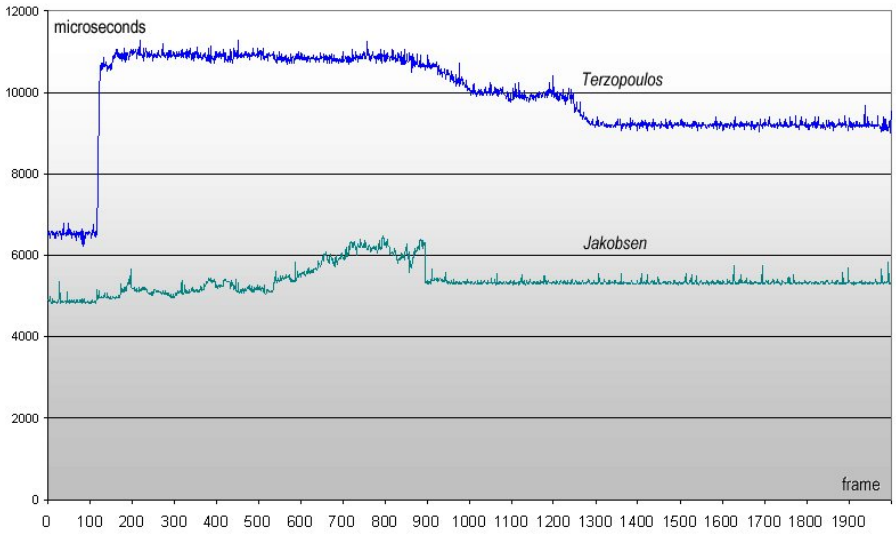
The charts from the 8 performance comparisons as referenced from 5.2.4 *Performance*. The first 4 charts show the computation time using 10 iterations, for the conjugate gradient solver from Terzopoulos and for the constraint relaxation solver from Jakobsen. The last 4 charts show the same tests, but extended to 100 iterations. All native frame computation data can be found on the supplemental CD [3], both as the native ASCII output format from the comparison application, and as Microsoft Excel sheets. A motion sequence showing both systems working on the same performance scene layout, in the case of 24×24 particles using 10 iterations, is also to be found on the CD.



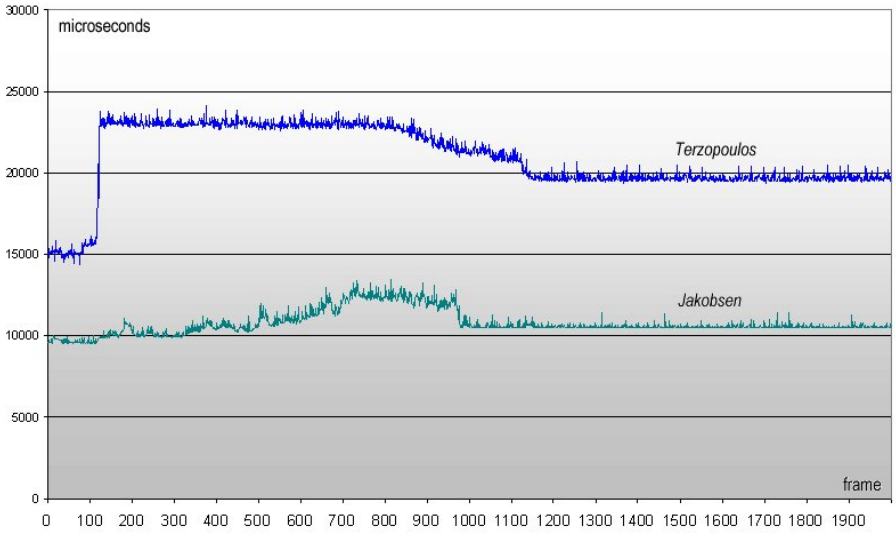
Particles: $8 \times 8 = 64$ Iterations: 10



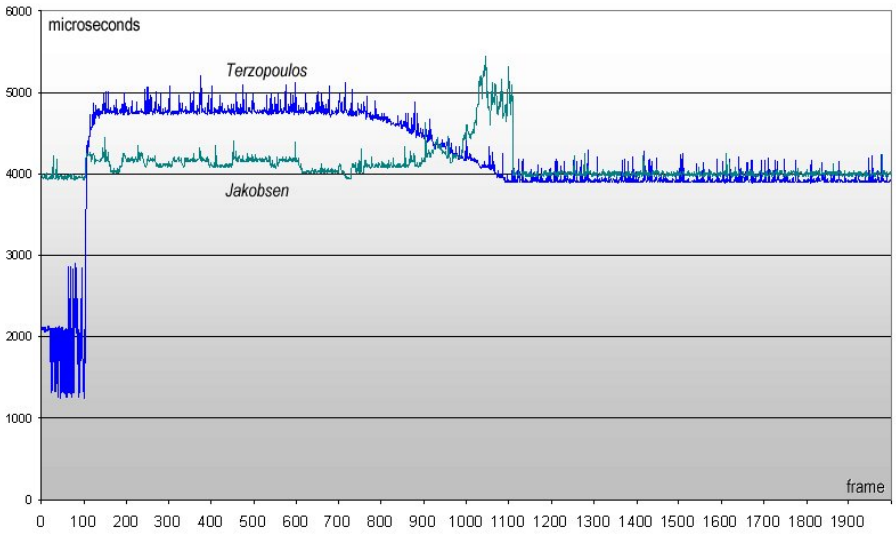
Particles: $16 \times 16 = 256$ Iterations: 10



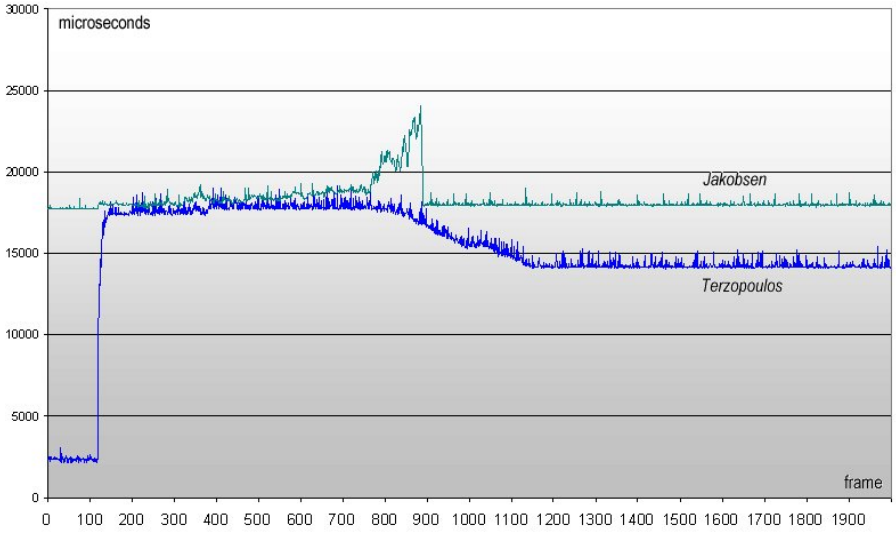
Particles: $24 \times 24 = 576$ Iterations: 10



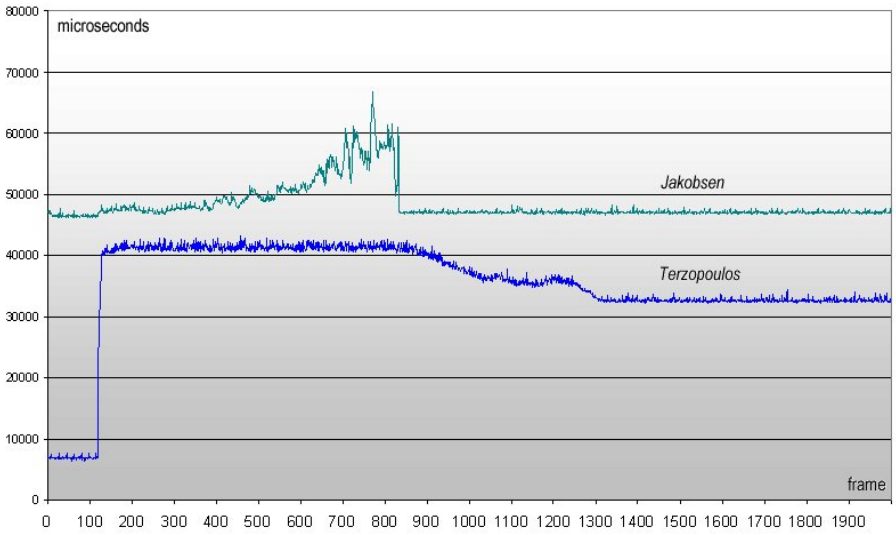
Particles: $32 \times 32 = 1024$ Iterations: 10



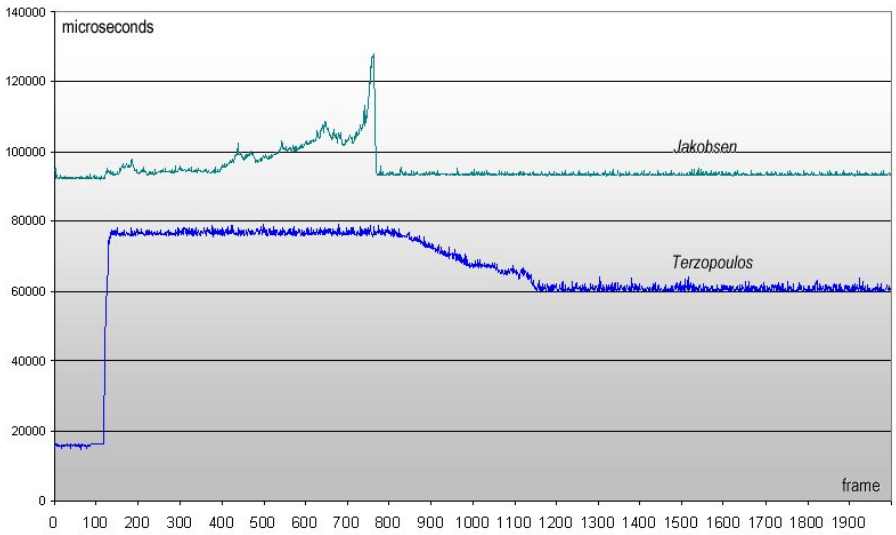
Particles: $8 \times 8 = 64$ Iterations: 100



Particles: $16 \times 16 = 256$ Iterations: 100



Particles: $24 \times 24 = 576$ Iterations: 100



Particles: $32 \times 32 = 1024$ Iterations: 100

A.2 Results Gallaria

The animations that originate from the still frames in 6.1 *Visual Test Results* are available under on the supplemental CD [3].

Heavy curtain

8 frames from the motion sequence of a heavy curtain sagging over a rigid sphere.











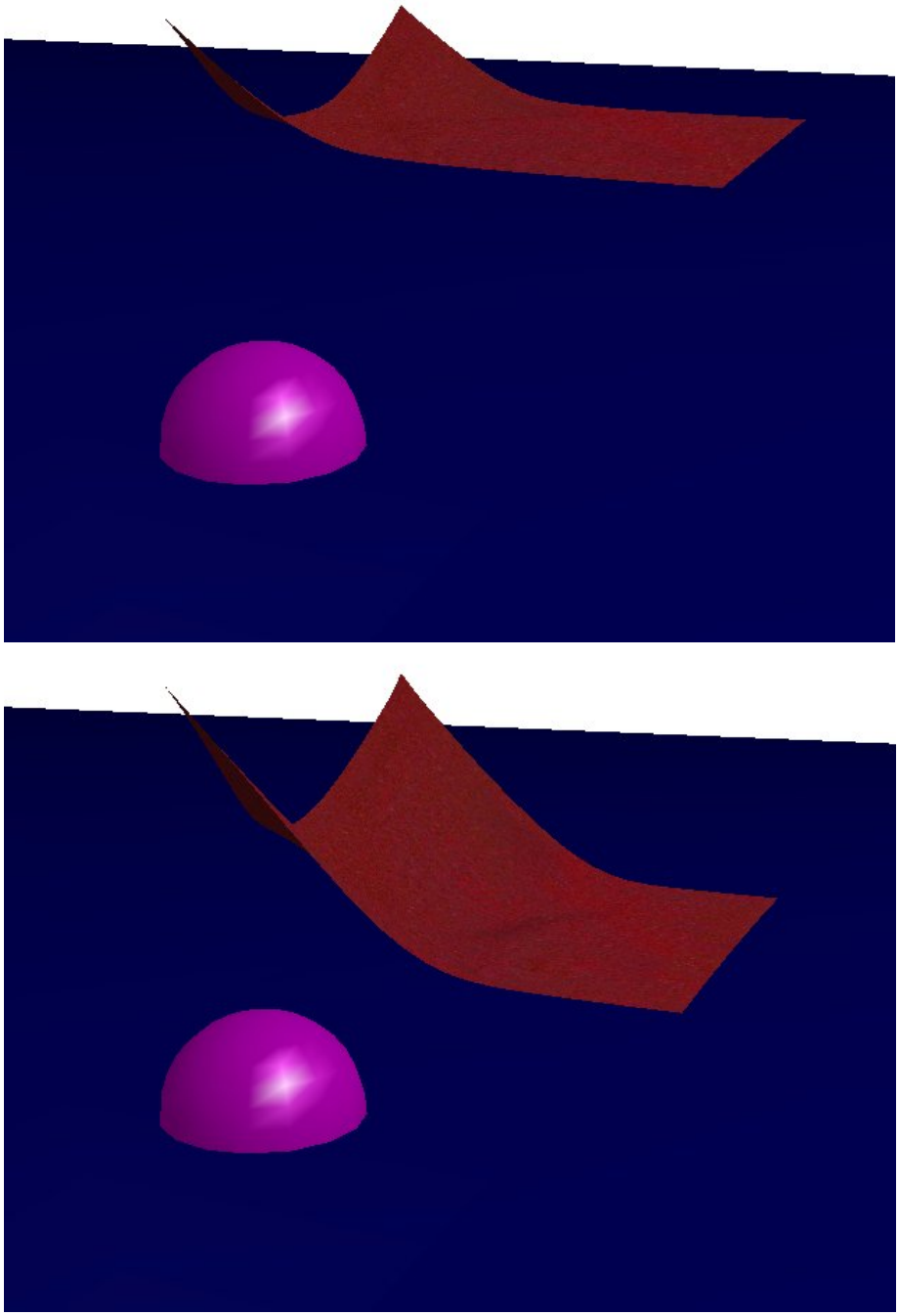


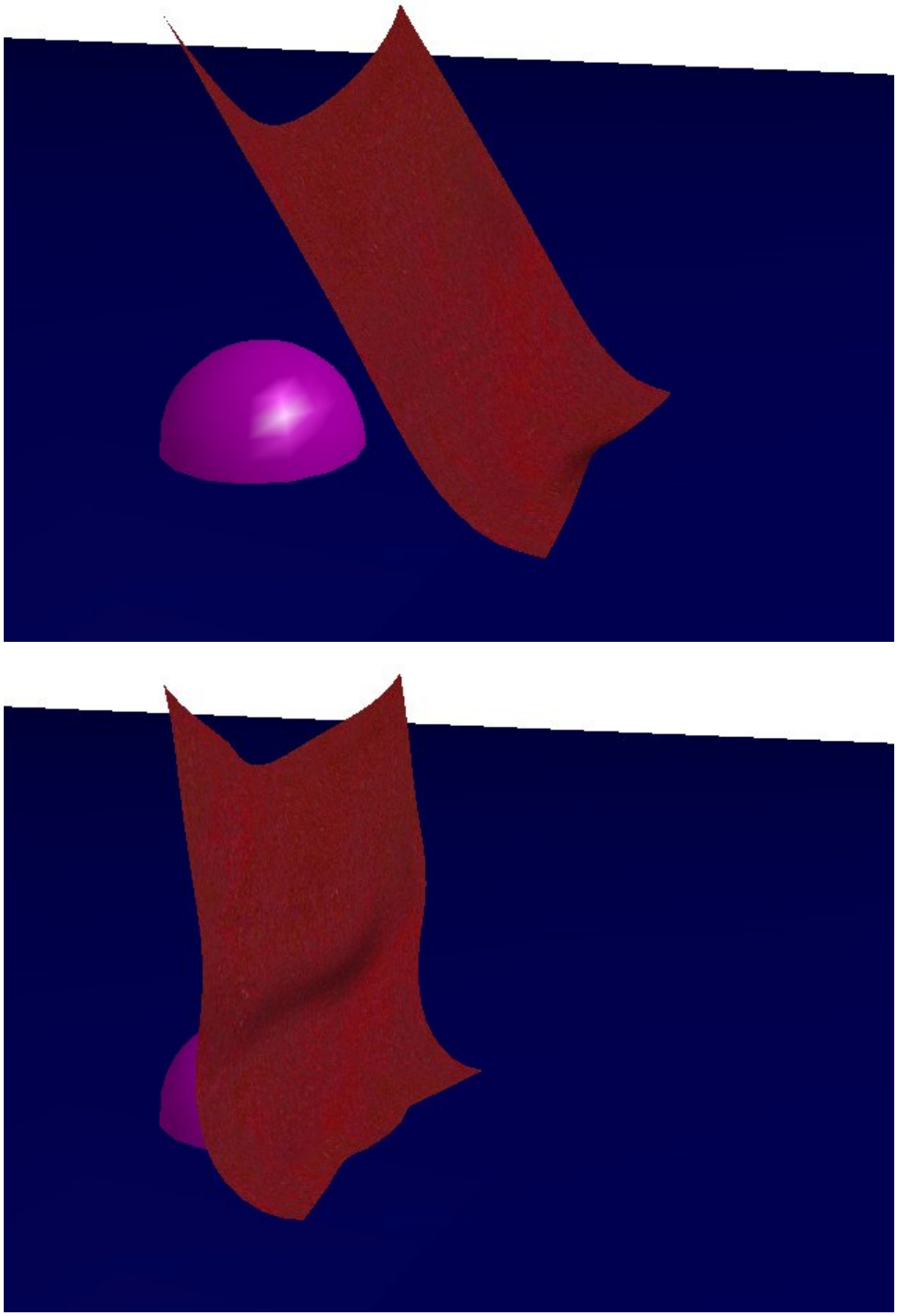


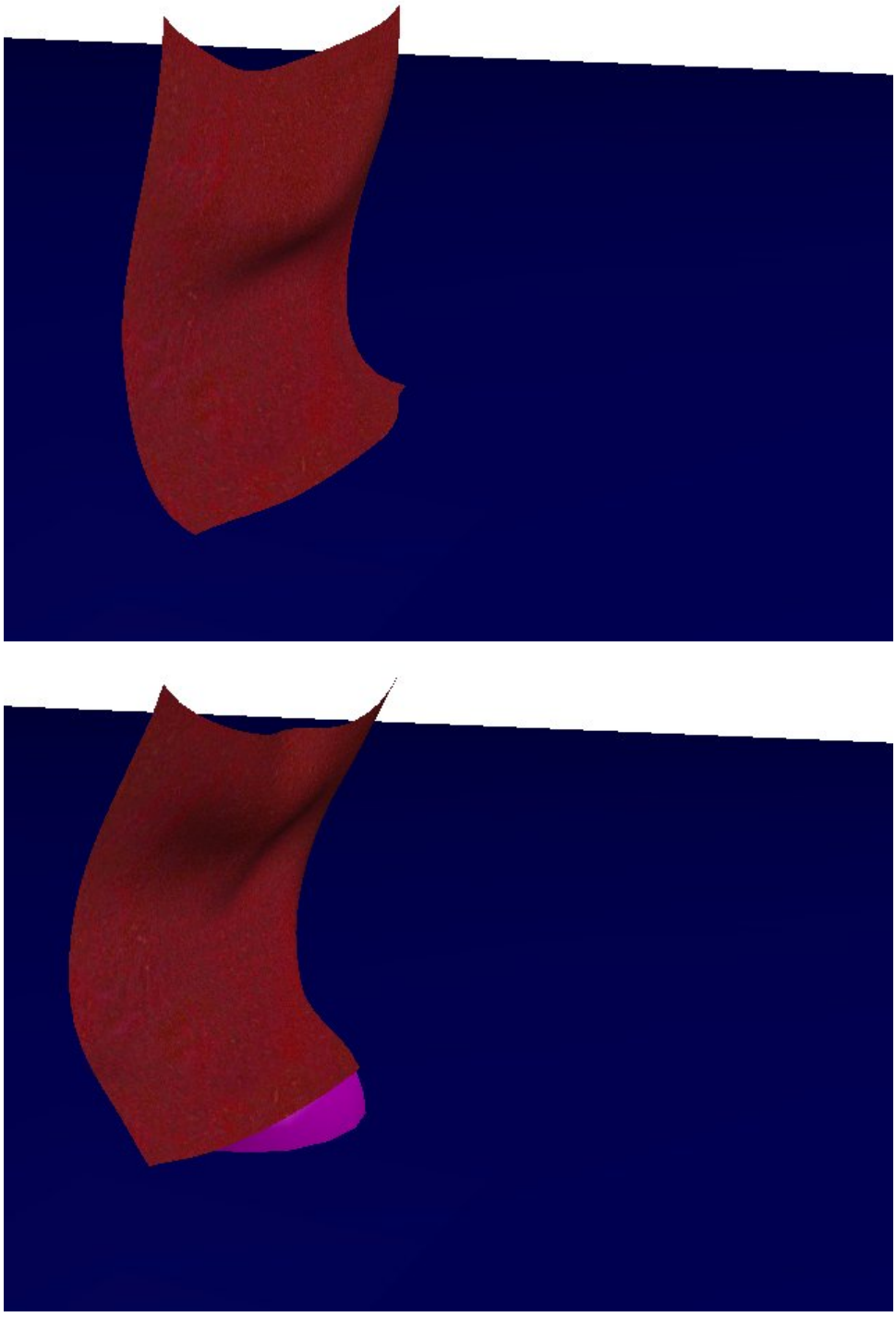


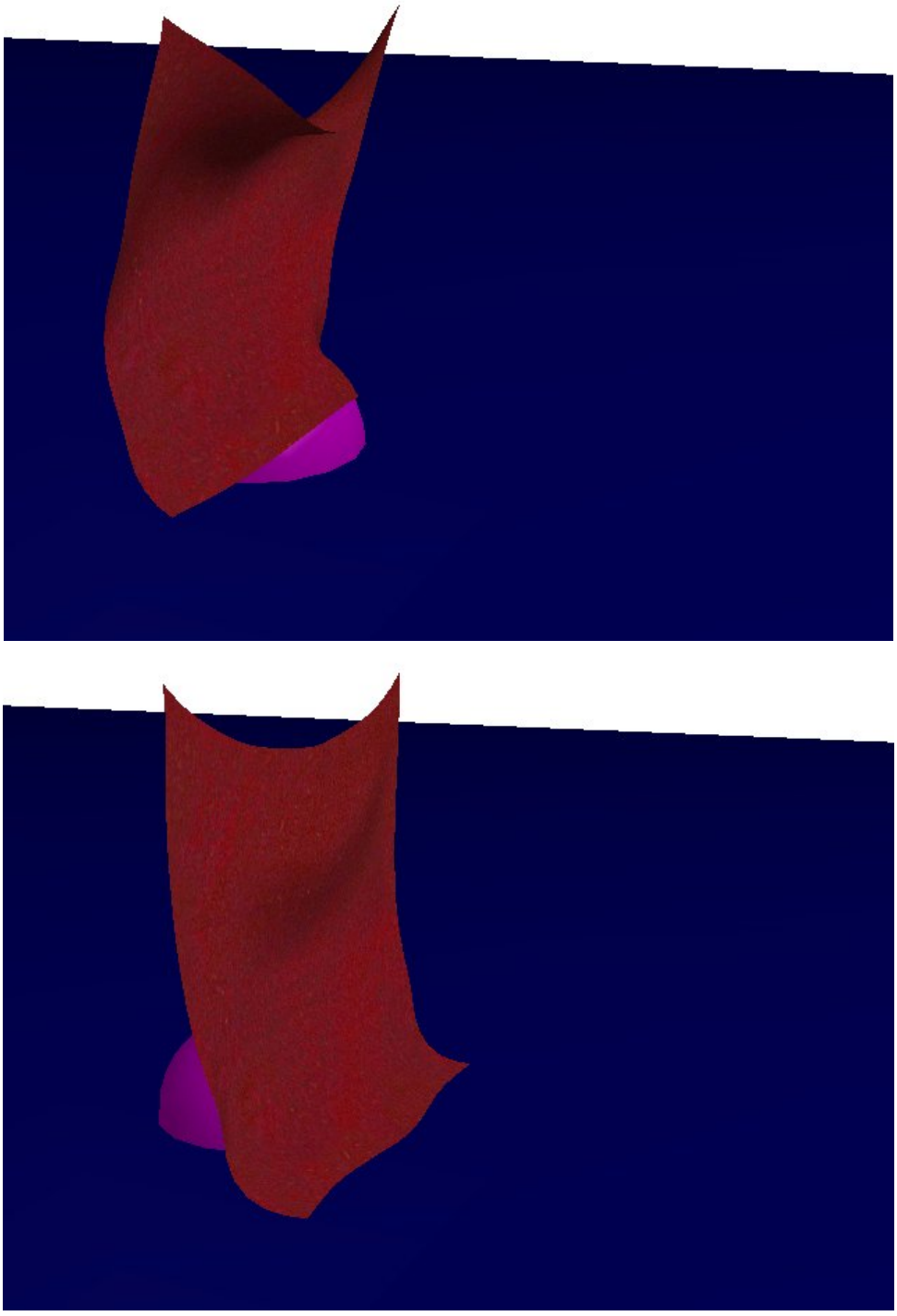
Tablecloth

Still frames the motion sequence of a corner constrained tablecloth, revealing beautiful folds in the fabric upon collision.



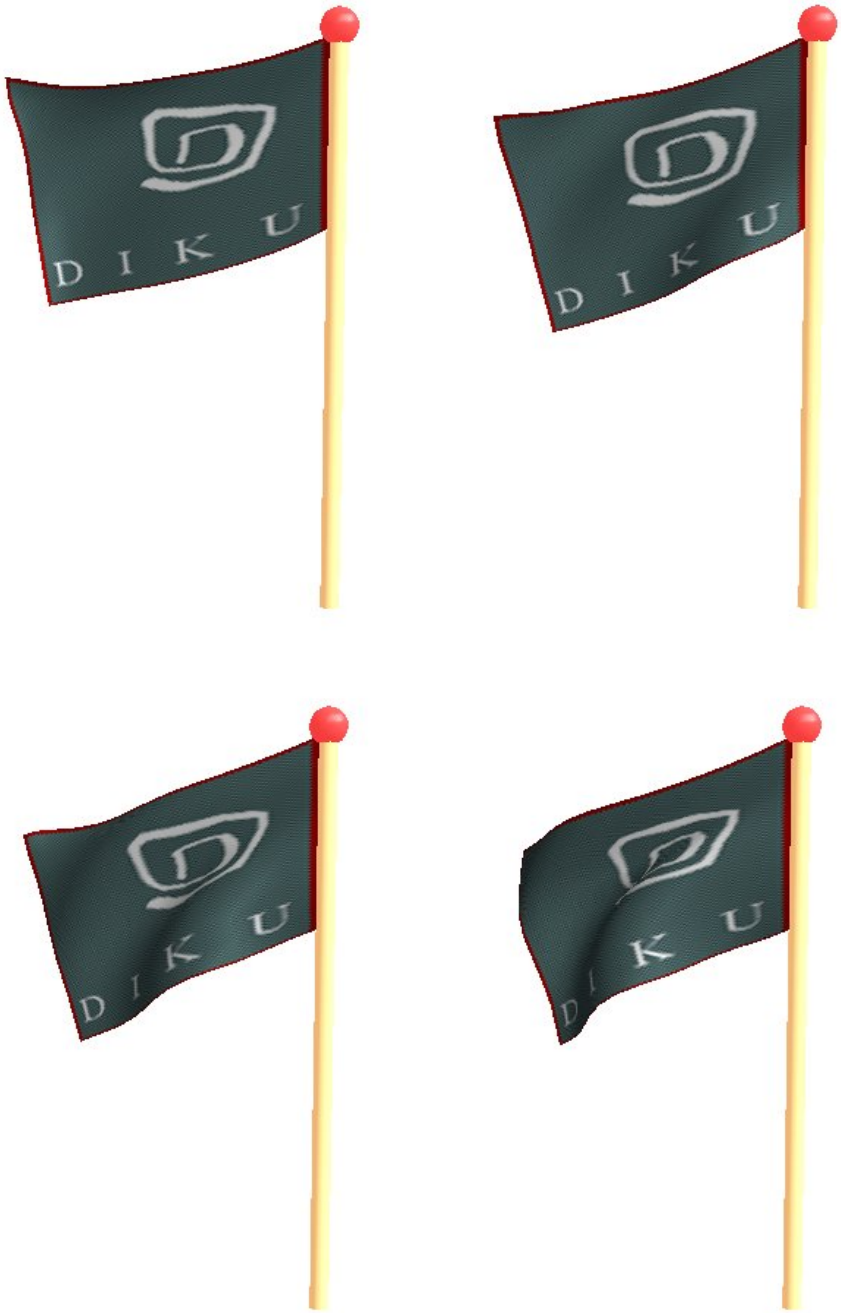






Waving Flag

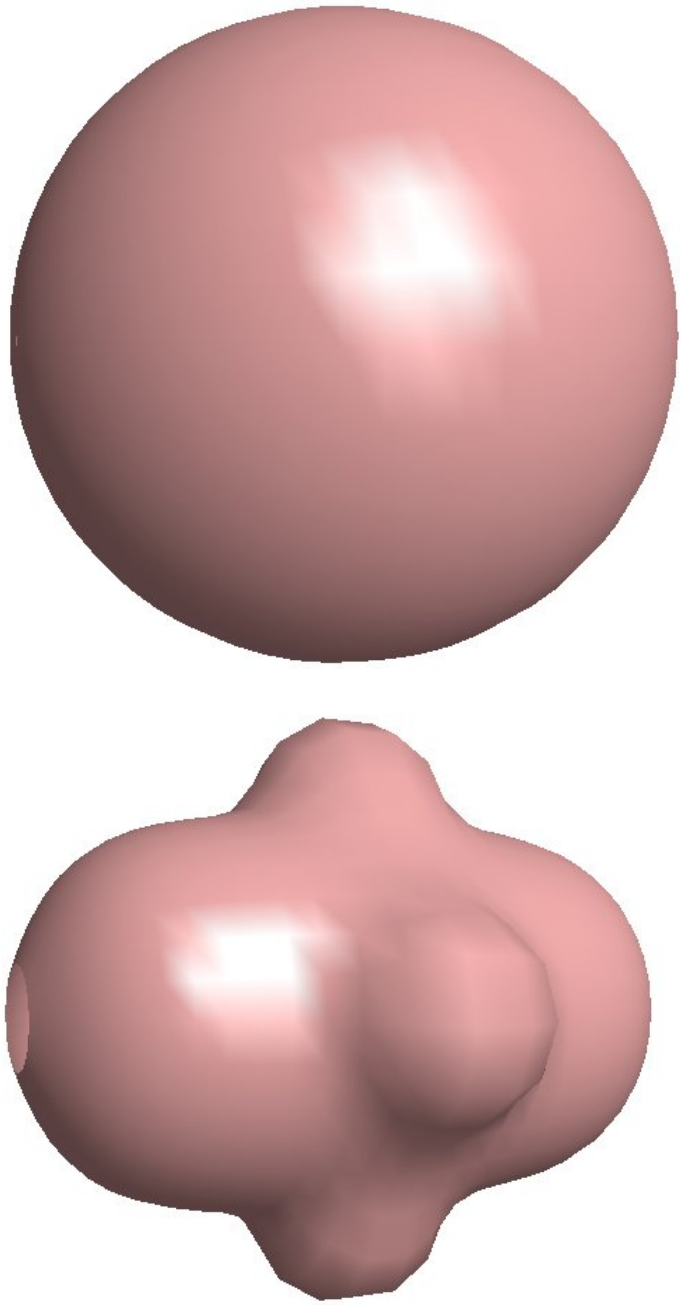
Still frames from the motion sequence of flag waving in the wind.

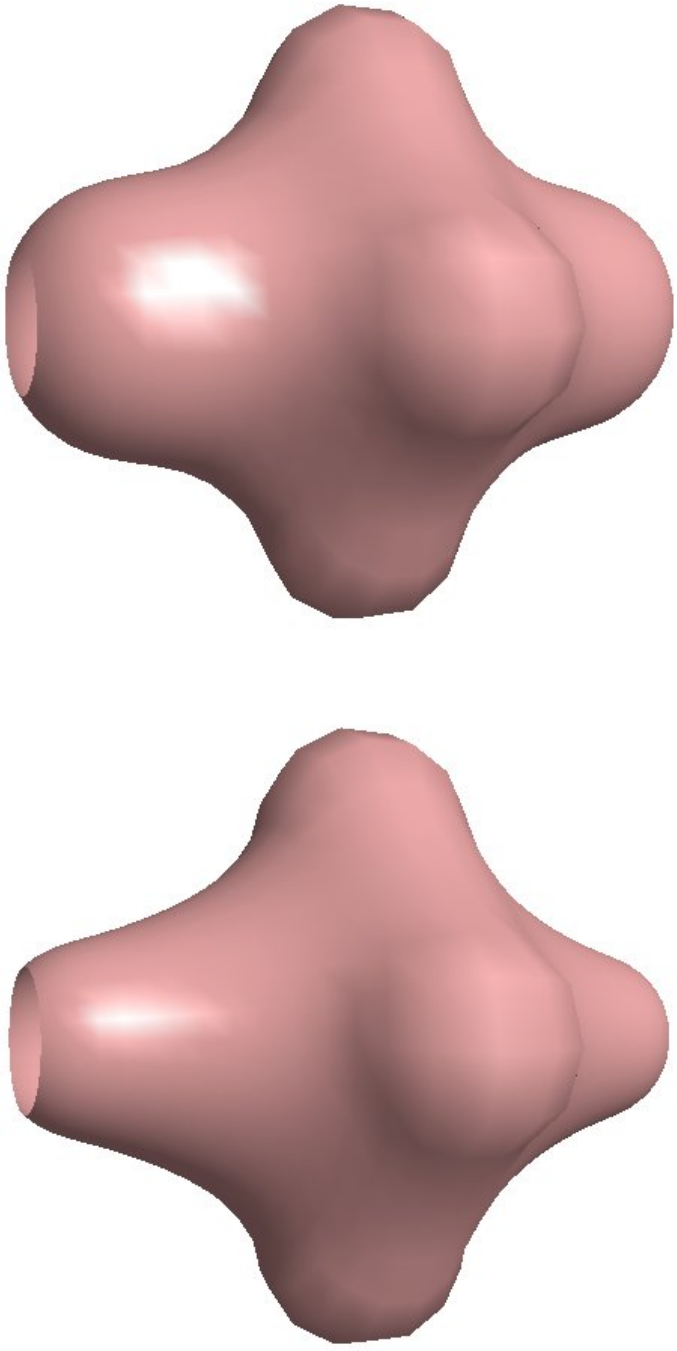


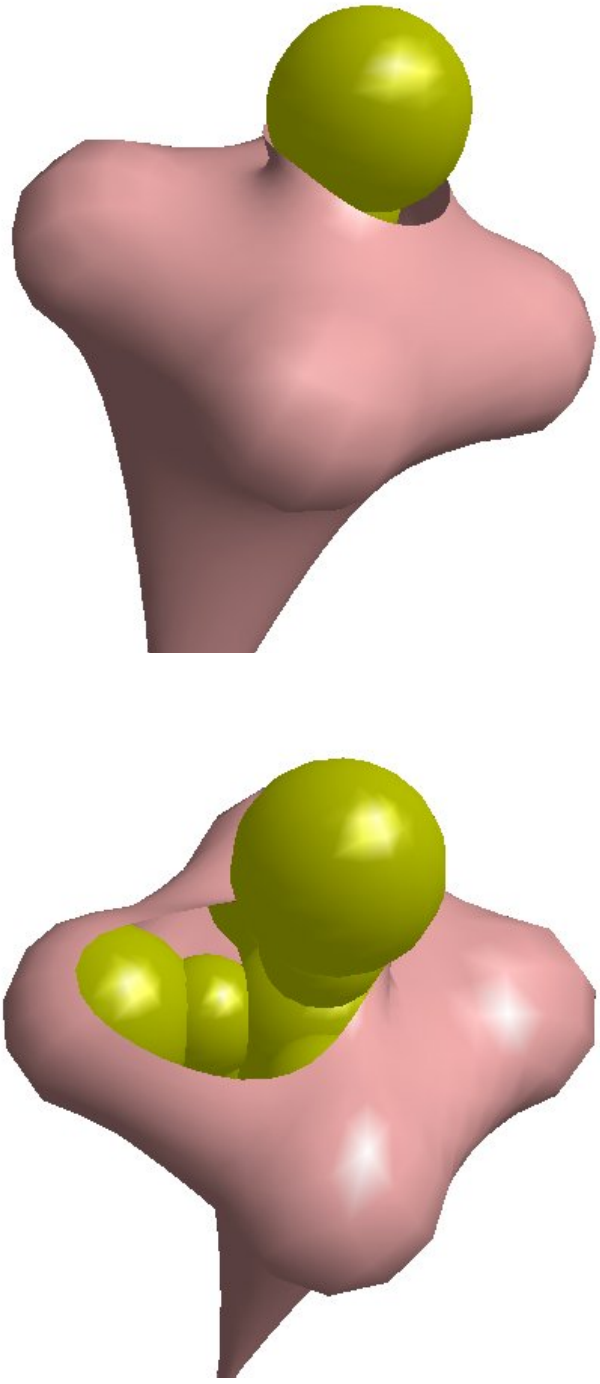

I K U
I K U
D I K U
D I K U

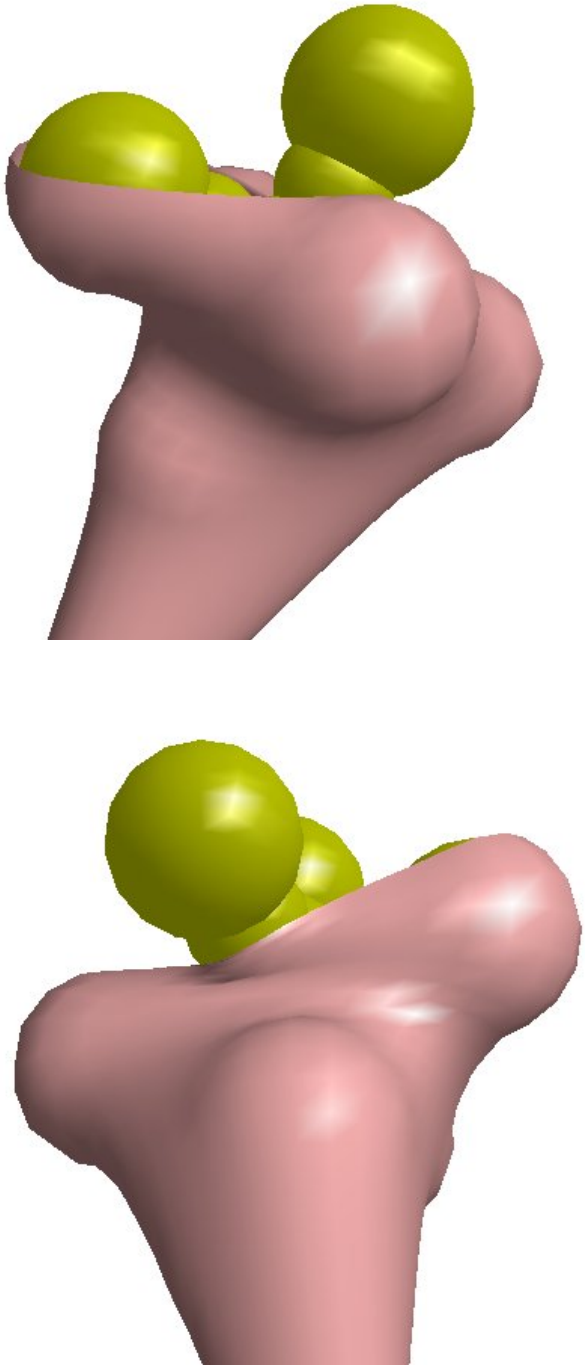
Thin membrane

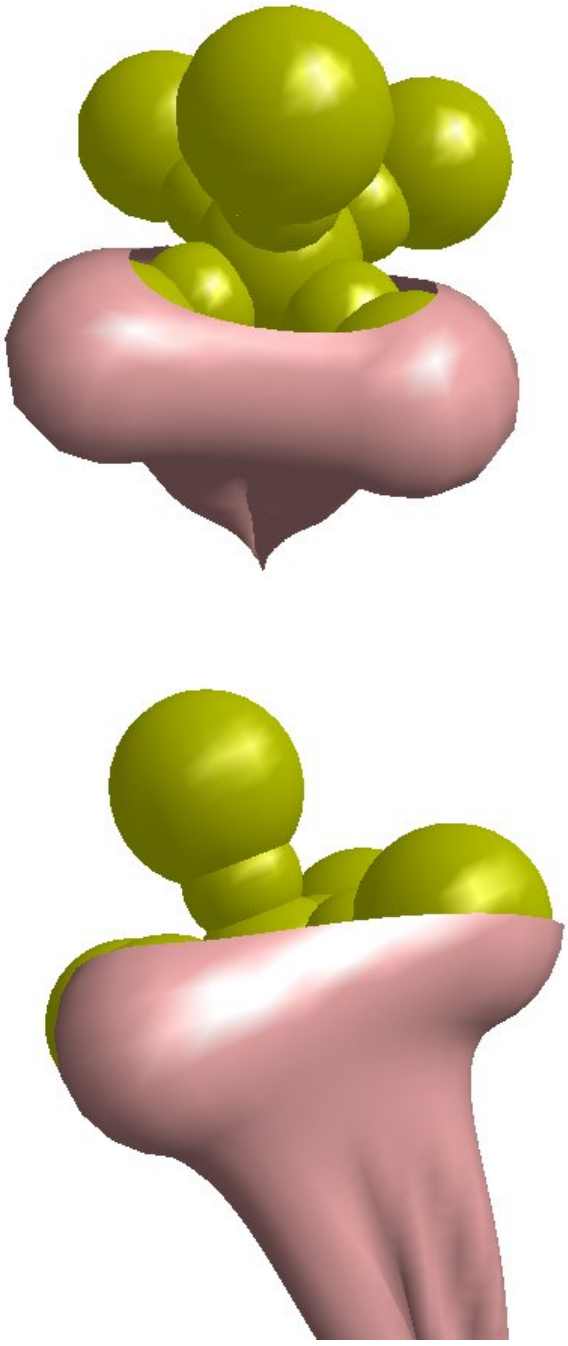
Still frames the motion sequence of the thin elastic membrane wrapping around the rigid jack. Notice the detailed folds in the rubber material, as we pull the membrane off the jack.











Sea Cucumber

Frames from the motion sequence of the poor sea cucumber that is breaking loose from two furious under-water spheres. To make things worse, someone is pulling its rear end around, making it all dizzy.

