Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Andreas Ess

Virtual cutting using smoothed particles

Summer Term 2004

Advisor:
Simone Hieber

Institute:
Prof. Dr. Petros Koumoutsakos
Institute of Computational Science

23.7.2004

Abstract

This thesis presents a particle–based simulation of a solid material to be used in a virtual surgery simulator. The simulation of the material is based on the generalized Hooke's Law, extended by the Kelvin–Voigt damping model, and is discretized using Smoothed Particle Hydrodynamics (SPH). The boundary conditions are solved by the use of ghost particles, representing fixed and stress–free boundaries.
The idea of ghost particles for stress–free boundaries is extended to support cutting of the material. Here, two different methods are examined that preserve geometry of the computational elements and thus only increase the computational overhead slightly. Cut surfaces are introduced to translate the movement of the virtual surgery instrument into information needed for splitting the particles.
The results obtained by using the methods developed within this semester thesis underline the usefulness of using SPH for a virtual surgery simulator.

Contents

1 Introduction **1**

2 Particle Method Approach **3**
2.1 Smoothed Particle Hydrodynamics . . . 3
2.1.1 Spatial Derivatives . . . 4
2.1.2 Evaluation Based on Initial Positions . . . 4
2.1.3 Visualization of the surface . . . 4
2.2 Governing Equations and Particle Discretization . . . 5
2.2.1 Integration Method . . . 6
2.2.2 Cell Lists . . . 7
2.3 Boundary Conditions Solved by Ghost Particles . . . 7
2.3.1 Ghost Particles . . . 7
2.3.2 Fixed Boundary . . . 8
2.3.3 Stress–free Boundary . . . 8

3 Virtual Cutting **11**
3.1 Basic Idea . . . 11
3.2 Cutting by Converting Particles . . . 12
3.3 Cutting by Splitting Particles . . . 13
3.3.1 Cut Surface . . . 15
3.3.2 Splitting the Particles . . . 17

4 Implementational Issues **19**
4.1 Class overview . . . 19
4.1.1 Particle . . . 19
4.1.2 ParticleList . . . 20
4.1.3 Actions on all particles . . . 21
4.1.4 Other classes . . . 21
4.1.5 Integration into user interface . . . 22
4.2 Marching Cubes . . . 22

5 Results **23**
5.1 Experimental Results . . . 23
5.1.1 Cutting by Converting . . . 23
5.1.2 Cutting by Splitting Particles . . . 24
5.2 Performance . . . 24

6 Conclusion **31**

7 Future Work **33**
7.1 Performance 33
7.2 Physically–based simulation 33
7.3 User interaction 34

8 Contents of CD **35**

Bibliography **37**

Chapter 1

Introduction

Virtual surgery simulation has nowadays gained in importance due to promising advantages against current surgery training, like reproducibility of individual operations, low cost and the possibility of rapid exchange of information.

A very important point in virtual surgery simulation is the geometric modification of the biological tissue. While other works focus on accurate modelling of tissue interaction with a scalpel using tetrahedral meshes [2], this thesis inspects the possibilities of using Smoothed Particle Hydrodynamics (SPH) for virtual surgery.
SPH is a mesh–free method for solving fluid dynamic equations. It provides a larger degree of accuracy than standard mass–spring systems at a lower computational overhead than FEM. This is an optimal basis for a surgery simulation, which needs to provide means for efficient modification of the organ's geometry in real–time, while still offering a sufficiently accurate model of the tissue. In contrast to FEM models, where a cut can heavily increase the size of the tetrahedral mesh, the actual topology is not affected by a cut in the approaches presented here.
The focus of this semester thesis lies in the physically–based simulation of the tissue, including boundary conditions, as well as basic cutting functionality. For the latter, two approaches are examined and discussed.

This semester thesis is based on two previous publications—in [7], SPH is discussed as a possibility for virtual surgery. In [4], volumetric organ modelling using SPH, as well as the handling of the virtual scalpel is described. While [4] focuses on the visualization aspect, this thesis first describes the modelling of the governing equations for a linear, viscoelastic solid using SPH in chapter 2. This also includes a description for handling boundary conditions using ghost particles. Chapter 3 forms the integral part of the thesis. Both the implementation of two cutting algorithms, as well as the integration into the existing framework are presented. Next, chapter 4 describes some of the implementational measures taken to improve performance and simplify further extension of the code. Then, results are presented and discussed in chapter 5. Chapter 6 gives a conclusion and chapter 7 an overview of things yet to be done. Finally, chapter 8 gives an overview of the accompanying CD.

Chapter 2

Particle Method Approach

In this chapter, the equations for Smoothed Particle Hydrodynamics (SPH) are presented. After a description of the method, its application to the governing equations of the simulation of a linear, viscoelastic material is shown. Then, boundary conditions and a possible implementation using so–called ghosts particles are pointed out.

2.1 Smoothed Particle Hydrodynamics

SPH [9] approximates any field quantity A at a position $\mathbf{x}$ by a weighted sum of contributions from all particles:

$$A(\mathbf{x}) = \sum_b A_b V_b W(\mathbf{x} - \mathbf{x_b}, h) \tag{2.1}$$

Here, the summation extends over all particles, with A_b denoting the function value at the location $\mathbf{x_b}$ of the b–th particle, and V_b its volume.
The function $W(\mathbf{x}, h)$ is called the kernel—a weighting scheme to quantify the influence of the particles in the superposition. It is an approximation of the dirac function:

$$\delta(\mathbf{x}) = \lim_{h \to 0} W(\mathbf{x}, h) \tag{2.2}$$

The parameter h is called the smoothing length and is used to control the size of the support radius, i.e. the number of neighboring particles that are considered.

For this semester thesis, a quartic spline kernel of second order (2.3) is used. It is normalized at the beginning of the simulation to ensure condition (2.4) on a particle p.

$$W(\mathbf{x}, h) = M_5(\mathbf{r}, h) = \frac{1}{\pi} \begin{cases} \frac{s^4}{4} - \frac{5s^2}{8} + \frac{115}{192} & 0 \leq s < \frac{1}{2}, \quad s = \frac{|r|}{h} \\ -\frac{s^4}{6} + \frac{5s^3}{6} + \frac{5s^2}{4} + \frac{5s}{24} + \frac{55}{96} & \frac{1}{2} \leq s < \frac{3}{2}, \\ \frac{(2.5-s)^4}{24} & \frac{3}{2} \leq s < \frac{5}{2} \\ 0 & s \geq \frac{5}{2} \end{cases} \tag{2.3}$$

$$\sum_j V_j W(\mathbf{x_p} - \mathbf{x_j}, h) = 1 \tag{2.4}$$

2.1.1 Spatial Derivatives

In SPH, the spatial derivative of any approximated field quantity on a particle a can be calculated *exactly* by derivating the kernel:

$$\left\langle \frac{\partial A}{\partial x_i} \right\rangle_a = \sum_b V_b (A_b - A_a) \frac{\partial}{\partial x_i} W(\mathbf{x_a} - \mathbf{x_b}, h) \tag{2.5}$$

$$\left\langle \frac{\partial^2 A}{\partial x_i \partial x_j} \right\rangle_a = \sum_b V_b (A_b - A_a) \frac{\partial^2}{\partial x_i \partial x_j} W(\mathbf{x_a} - \mathbf{x_b}, h) \tag{2.6}$$

The symmetric term $(A_b - A_a)$ in the sum is introduced for numerical reasons.

2.1.2 Evaluation Based on Initial Positions

In order to secure the convergence of the SPH method, the particle map must be regular. One way to ensure this is by using remeshing, as seen in [7].

In the scope of this semester thesis, a simpler approach was chosen: particles carry both an initial, Lagrangian, position ξ as well as the current, Eulerian, position $\mathbf{x}(\xi, t)$. These are related through

$$\mathbf{x}(\xi, t) = \xi + \mathbf{d}(\xi, t) \tag{2.7}$$

, where $\mathbf{d}(\xi, t)$ is called the displacement of the particle.

For evaluating the constitutive model, Eqs. (2.1) and (2.5) are calculated based on the initial positions of the particles. Since these are initialized on a regular map and don't change over time, the above condition is ensured at all times.

2.1.3 Visualization of the surface

As described in the previous thesis [4], the marching cubes algorithm is used to derive an iso–surface from the particles. It is based on the mass m_b, which is chosen to be $m_b = V_b \rho_{init}$. The initial density ρ_{init} is assumed to be 1. Together with Eq. (2.1) this yields Eq. (2.8)—which uses the current positions $\mathbf{x}$ as opposed to the actual simulation.

$$\rho(\mathbf{x}) = \sum_b m_b W(\mathbf{x} - \mathbf{x_b}, h) \tag{2.8}$$

Since the smoothness of the kernel is a minor issue in the computation of the iso–surface, a simplified kernel (2.9) was used for performance reasons. It has the same order as M_5.

$$W(\mathbf{x}, h) = \frac{1}{h^3} \left(1 - \frac{\|\mathbf{x}\|}{h}\right) \tag{2.9}$$

2.2 Governing Equations and Particle Discretization

As in [7], the mechanical behavior of soft biological tissue is modeled as a linear viscoelastic material for small strains. The governing equation is the momentum equation,

$$\rho \frac{D\mathbf{u}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{f_{ext}} \tag{2.10}$$

For simplicity's sake, the particles' density ρ is assumed to be 1 and the divergence is taken with respect to the initial positions. $\frac{D}{Dt}$ denotes the material derivative

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla \tag{2.11}$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{u} \tag{2.12}$$

where $\mathbf{u}$ is the velocity.

$\mathbf{f_{ext}}$ is an external body force—a simple gravitational force was used here. $\boldsymbol{\sigma}$ is the Cauchy stress tensor. It depends on the constitutive model of the material used. The solid model is based on the generalized Hooke's law [3] and is extended by the Kelvin–Voigt damping model [8]. This results in the components σ_{ij} of the stress tensor $\boldsymbol{\sigma}$ to depend linearly on the components ϵ_{ij} of the Cauchy Green strain tensor $\boldsymbol{\epsilon}$,

$$\sigma_{ij} = 2\mu(\epsilon_{ij} + T\dot{\epsilon}_{ij}) + \lambda\delta_{ij}(\epsilon_{kk} + T\dot{\epsilon}_{kk}) \tag{2.13}$$

The indicies $i, j, k = 1, 2, 3$ follow the Einstein's summation convention and δ_{ij} is the Kronecker symbol. The time constant T is the relaxation time used for the damping, and μ and λ are the Lame constants,

$$\begin{aligned} \mu &= \frac{E}{2(1+\nu)} \\ \lambda &= \frac{\nu E}{(1-2\nu)(1+\nu)} \end{aligned} \tag{2.14}$$

where E represents the Young's modulus and ν the Poisson's ratio.

The components ϵ_{ij} of the strain tensor are linearly dependent on the spatial derivative of the displacement $\mathbf{d}$,

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial d_i}{\partial \xi_j} + \frac{\partial d_j}{\partial \xi_i}\right) \tag{2.15}$$

Putting Eqs. (2.10), (2.13) and (2.15) together for a particle p with $\mathbf{d_p} = \mathbf{x_p} - \xi_\mathbf{p}$ results in Eqs. (2.16)–(2.18).

$$\begin{aligned}
\frac{Du_1}{Dt} &= (2\mu+\lambda)\frac{\partial^2 d_1}{\partial \xi_1^2} + \lambda\frac{\partial^2 d_2}{\partial \xi_1 \partial \xi_2} + \lambda\frac{\partial^2 d_3}{\partial \xi_1 \partial \xi_3} \\
&+ \mu(\frac{\partial^2 d_1}{\partial \xi_2^2} + \frac{\partial^2 d_2}{\partial \xi_1 \partial \xi_2}) + \mu(\frac{\partial^2 d_3}{\partial \xi_3 \partial \xi_1} + \frac{\partial^2 d_1}{\partial \xi_3^2}) \\
&+ T\Big((2\mu+\lambda)\frac{\partial^2 u_1}{\partial \xi_1^2} + \lambda\frac{\partial^2 u_2}{\partial \xi_1 \partial \xi_2} + \lambda\frac{\partial^2 u_3}{\partial \xi_1 \partial \xi_3} \\
&+ \mu(\frac{\partial^2 u_1}{\partial \xi_2^2} + \frac{\partial^2 u_2}{\partial \xi_1 \partial \xi_2}) + \mu(\frac{\partial^2 u_3}{\partial \xi_3 \partial \xi_1} + \frac{\partial^2 u_1}{\partial \xi_3^2})\Big)
\end{aligned} \tag{2.16}$$

$$\begin{aligned}
\frac{Du_2}{Dt} &= \lambda\frac{\partial^2 d_1}{\partial \xi_1 \partial \xi_2} + (2\mu+\lambda)\frac{\partial^2 d_2}{\partial \xi_2^2} + \lambda\frac{\partial^2 d_3}{\partial \xi_2 \partial \xi_3} \\
&+ \mu(\frac{\partial^2 d_1}{\partial \xi_2 \partial \xi_1} + \frac{\partial^2 d_2}{\partial \xi_1^2}) + \mu(\frac{\partial^2 d_2}{\partial \xi_3^2} + \frac{\partial^2 d_3}{\partial \xi_2 \partial \xi_3}) \\
&+ T\Big(\lambda\frac{\partial^2 u_1}{\partial \xi_1 \partial \xi_2} + (2\mu+\lambda)\frac{\partial^2 u_2}{\partial \xi_2^2} + \lambda\frac{\partial^2 u_3}{\partial \xi_2 \partial \xi_3} \\
&+ \mu(\frac{\partial^2 u_1}{\partial \xi_2 \partial \xi_1} + \frac{\partial^2 u_2}{\partial \xi_1^2}) + \mu(\frac{\partial^2 u_2}{\partial \xi_3^2} + \frac{\partial^2 u_3}{\partial \xi_2 \partial \xi_3})\Big)
\end{aligned} \tag{2.17}$$

$$\begin{aligned}
\frac{Du_3}{Dt} &= \lambda\frac{\partial^2 d_1}{\partial \xi_1 \partial \xi_3} + \lambda\frac{\partial^2 d_2}{\partial \xi_2 \partial \xi_3} + (2\mu+\lambda)\frac{\partial^2 d_3}{\partial \xi_3^2} \\
&+ \mu(\frac{\partial^2 d_1}{\partial \xi_3 \partial \xi_1} + \frac{\partial^2 d_3}{\partial \xi_1^2} + \mu(\frac{\partial^2 d_2}{\partial \xi_3 \partial \xi_2}) + \frac{\partial^2 d_3}{\partial \xi_2^2}) \\
&+ T\Big(\lambda\frac{\partial^2 u_1}{\partial \xi_1 \partial \xi_3} + \lambda\frac{\partial^2 u_2}{\partial \xi_2 \partial \xi_3} + (2\mu+\lambda)\frac{\partial^2 u_3}{\partial \xi_3^2} \\
&+ \mu(\frac{\partial^2 u_1}{\partial \xi_3 \partial \xi_1}) + \frac{\partial^2 u_3}{\partial \xi_1^2} + \mu(\frac{\partial^2 u_2}{\partial \xi_3 \partial \xi_2} + \frac{\partial^2 u_3}{\partial \xi_2^2})\Big)
\end{aligned} \tag{2.18}$$

Eqs. (2.16)–(2.18) can now be discretized using SPH—the spatial derivatives of the displacement based on the initial position can be calculated exactly using Eq. (2.6). As the displacement is by means of Eq. (2.7) only dependent on current and initial positions, this also defines the set of attributes needed per particle:

- Current position $\mathbf{x}(\xi, t)$
- Initial position ξ
- Initial volume V_{init}

Since the evaluation of the Eqs. (2.16)–(2.18) is based on initial positions, $\mathbf{u}$ is 0 at all times, and the material derivative of Eq. (2.11) turns into a standard time derivative.

2.2.1 Integration Method

Integration of the particles is done using a third order Beeman integrator [1].

$$\mathbf{u}(t+\delta) = \mathbf{u}(t) + \delta(\frac{5}{3}\mathbf{a}(t) + \frac{2}{3}\mathbf{a}(t-\delta) - \frac{1}{12}\mathbf{a}(t-2\delta)) \tag{2.19}$$

$$\mathbf{x}(t+\delta) = \mathbf{x}(t) + \delta(\mathbf{u}(t) + \frac{2}{3}\mathbf{a}(t-\delta)\delta - \frac{1}{6}\mathbf{a}(t-2\delta)\delta) \tag{2.20}$$

Note that this scheme only uses one evaluation of the functional (2.16)–(2.18) per time–step, as opposed to methods like Runge–Kutta. However, to obtain a higher order, the integration is based on an interpolation of the acceleration in time. This means that sudden changes in the topology (e.g. by cutting or pushing particles) can result in instabilities.

However, as the cutting simulation is supposed to work in real time, this integration scheme is used whenever possible. A simpler Euler–Cromer integrator, seen in Eqs. (2.21)–(2.22), is used temporarily when the particles are subject to sudden changes, as well as to to initialize Eqs. (2.19)–(2.20) at the beginning of the simulation.

$$\mathbf{u}(t+\delta) = \mathbf{u}(t) + \mathbf{a}(t)\delta \tag{2.21}$$
$$\mathbf{x}(t+\delta) = \mathbf{x}(t) + \mathbf{u}(t+\delta)\delta \tag{2.22}$$

2.2.2 Cell Lists

The kernel from Eq. (2.3) has a limited support. Thus, when evaluating the superposition of Eq. (2.1), the sum only has to be calculated over a local neighborhood of the current particle.
This is done using cell lists, i.e. the particles are sorted into a regular grid depending on their initial position. The size of the cells is chosen in such a way that the 26 direct neighbors of a cell are within the kernel's support radius.
Given a specific particle, one can easily determine the cell it is located in, and then evaluate Eq. (2.1) only using the particles located in the neighboring 26 cells—instead of taking all particles into account. Depending on the support radius of the kernel, this can give an immense speed–up.

Section 4.1.3 deals with the actual implementation, and how new functions can easily be defined to take advantage of cell lists.

2.3 Boundary Conditions Solved by Ghost Particles

In order to solve Eqs. (2.16)–(2.18), one needs to know both the initial state of the variables, as well as the boundary conditions on the spatial edge of the domain. Two different types of boundaries are considered.

- **Fixed boundaries**, used to clamp a material, enforce a given displacement at the boundary.
- **Stress–free boundaries** are boundaries where the surface traction is 0.

2.3.1 Ghost Particles

In the scope of this thesis, a discretization of these boundary conditions using ghost particles (or short, ghosts) is used, inspired by [10].
Ghosts are particles residing outside of the actual domain. They are purely passive particles that don't evolve. However, ghosts are accounted for in the superposition of Eq. (2.1). Furthermore, while they have a specific initial position ξ, their displacement $\mathbf{d}$ is adjusted on the fly depending on the particle a whose field is currently calculated.

The displacement $\mathbf{d}$ of the ghost is dependent on both the boundary condition and the particle a whose field is calculated.

The advantage of this method lies in its efficiency, since it circumvents the need of one–sided differentiation.

In the following sections, the displacement field for a particle a is calculated. Its displacement is denoted as $\mathbf{d_a}$. Similarly, g denotes a ghost particle, with $\mathbf{d_g}$ being its displacement.
As a simplified organ, a cube was chosen. Ghost particles are added in layers around the cube, where the number of layers needed is determined by the support radius of the kernel.

2.3.2 Fixed Boundary

The goal of a fixed boundary is for the displacement field to have a given value at a specific position. Since particles aren't actually located on the boundary, the idea is obtain the desired displacement by adapting the ghost particles outside of the domain accordingly.
During the calculation of the Eqs. (2.16)–(2.18) for a particle a, whenever a fixed boundary ghost particle g is consulted for the calculation, $\mathbf{d_g}$ is linearly extrapolated such that the displacement on the boundary corresponds to the specified value.

For instance, if the material is clamped to a horizontal $y-z$ plane at position ξ_{plane}, with a displacement of $\mathbf{d_{boundary}}$, one obtains Eq. (2.23). ξ and ξ_a denote the x component of the initial position.
Figure 2-1 shows a one–dimensional equivalent.

$$\mathbf{d_g} = \frac{\xi_g - \xi_a}{\xi_{plane} - \xi_a}(\mathbf{d_{boundary}} - \mathbf{d_a}) + \mathbf{d_a} \tag{2.23}$$

2.3.3 Stress–free Boundary

On a stress–free boundary, one has to ensure that the displacement quantities remain constant, such that no force acts on the surface. The stress–free boundary ghosts behave like a passive, unstretched material that is glued to the actual soft material.

This is accomplished by linking each ghost particle to an actual particle on the boundary of the domain, called neighbor. Whenever g is consulted, $\mathbf{d_g}$ takes the displacement of this neighbor.

$$\mathbf{d_g} = \mathbf{d_n} \tag{2.24}$$

Figure 2-2 shows how ghost particles are linked to actual material particles for different boundaries. Note that for the third case, it is not clear what g should adjust to when considered during the calculation of particle d. Best results were achieved by using an average of g's two orthogonal neighbors a and b.

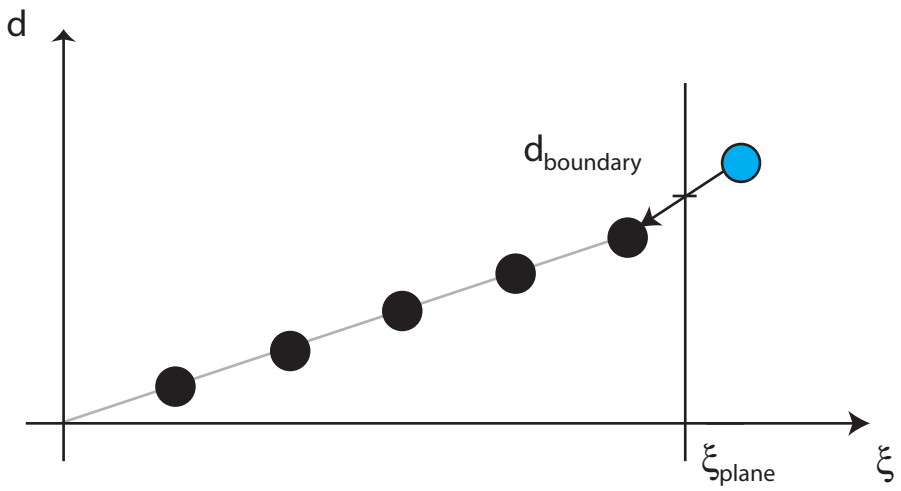

Figure 2-1: Fixed boundary using ghost particles in 1D

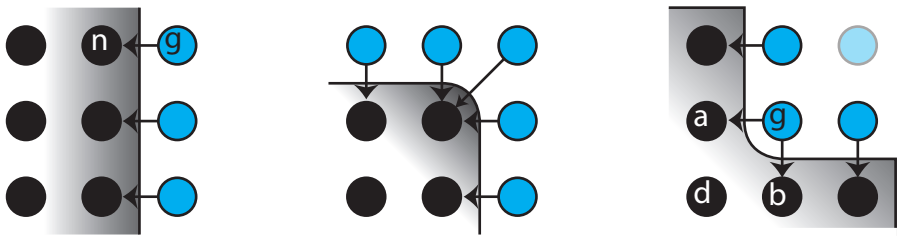

Figure 2-2: Stress–free boundary using ghost particles

Chapter 3

Virtual Cutting

The central part of this semester thesis was to investigate approaches to cut objects within the SPH model. In this chapter, two approaches are described. Both rely on a technique similar to the ghost particles used to solve the stress–free boundary conditions. First, a description of this technique is given, followed by the two actual approaches.
The last section describes the implementation into the graphical user interface of the surgery simulation provided by [4].

3.1 Basic Idea

For a stress–free boundary, a ghost particle has one possible neighbor it adapts to.[1] However, if there is a cut between two particles, a new stress–free boundary is created *between* the two. Thus, when describing a cut via ghost particles, such a ghost has at least two neighbors, as seen in figure 3-1. In the following, $\mathcal{N}$ denotes the set of neighbors for a ghost particle.

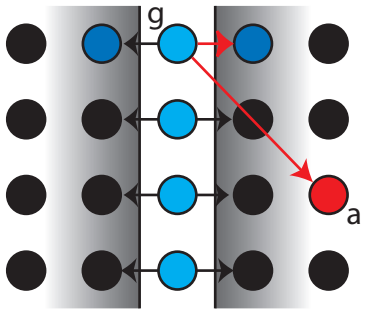

Figure 3-1: Ghost particles in a cut

[1] Disregarding the third case of figure 2-2, the solution to which has been introduced along with this technique.

The set $\mathcal{N}$ of a ghost particle is defined by the cut.

When such a ghost is considered during the evaluation of Eqs. (2.16)–(2.18) for a particle a, the displacement from the best neighbor is considered. To determine the best neighbor, two normalized directions are defined: first, the vector from the ghost particle g to the particle a, seen in Eq. (3.1). The set of direction vectors from the ghost particle to its neighbors is defined by Eq. (3.2).

$$\mathbf{d1} = \frac{\xi_\mathbf{a} - \xi_\mathbf{g}}{\|\xi_\mathbf{a} - \xi_\mathbf{g}\|} \tag{3.1}$$

$$\mathbf{d2_n} = \frac{\xi_\mathbf{n} - \xi_\mathbf{g}}{\|\xi_\mathbf{n} - \xi_\mathbf{g}\|}, \text{for all } n \in \mathcal{N} \tag{3.2}$$

Upon evaluating the constitutive model for particle a, the ghost g chooses the neighbor maximizing the dot product of Eqs. (3.1) and (3.2).

$$m = \arg\max_{n \in N} \mathbf{d1} \cdot \mathbf{d2_n} \tag{3.3}$$

If there is no clear maximum (i.e., two or more neighbors have the same dot product up to an additive factor ϵ), the displacements of the corresponding neighbors are averaged. See the following sections for examples when this is necessary.

3.2 Cutting by Converting Particles

A first approach for cutting is to convert particles inside the scalpel into ghost particles and add links to their immediate neighbors. Figure 3-2 shows an examples for the two–dimensional case:
The red particle p to be cut is converted into a ghost particle in a first step. This means that a new stress–free boundary is created for all of its nearest neighbors, to which it is thus linked. Furthermore, the ghost particle g that previously kept p as a neighbor has to search for a new set of neighbors, since it can't adapt to a ghost particle.

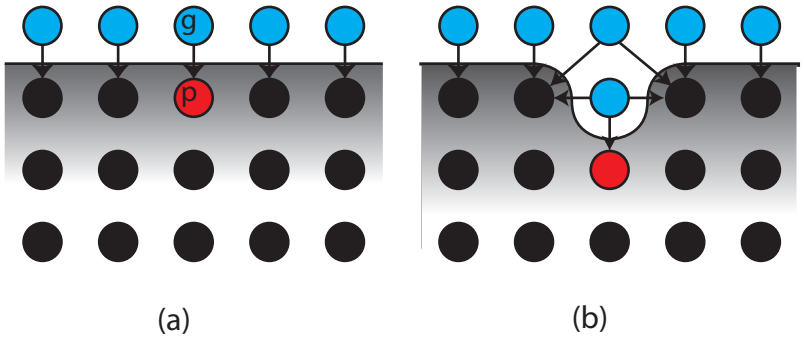

Figure 3-2: Cutting by converting particles

This yields the following algorithm when converting a material particle p to a ghost particle:

1. Change type of p to ghost particle.
2. Find immediate neighbors of p.
3. For all ghost particles g that had a link on p:
   (a) Clear neighborhood of g.
   (b) Using cell lists, find nearest material particle(s), link them.

The immediate neighbors of step 2 are defined as the 6 direct neighbors of the particle in all main directions. Extended, diagonal linkage was investigated as well but did not yield better results with respect to stability.
Note that in step 3 (b), it's possible to link g to more than one actual particle, if the distances are approximately the same. This is necessary for instance in case (b) of figure 3-2. Here, g can adapt to either side of the cut, as desired.

As mentioned above, it can happen that Eq. (3.3) has no clear maximum—as already seen in case (c) of figure 2-2. As an approximation, the averaged displacement of the two most fitting neighbors is chosen. Numerical instabilities can ensue due to this, making the use of damping necessary.

Obviously, volume is going to be lost when transforming material particles into ghost particles. Introducing additional particles might be considered as a solution for this. However, this is critical with respect to the regularity of the map.
Because of this, cutting by converting was only regarded in a first step. Cases with conversion at fixed boundaries, or with kernels that have a larger support radius than $0.75h^2$—where h denotes the particle spacing—were not considered.

3.3 Cutting by Splitting Particles

Another method is to treat particles that surround a cut as *hybrid particles*. Such particles can behave as material or ghost particles, depending on the situation. The idea is to assume the cut to be between two such hybrid particles as opposed to at a particle itself.
A cut is now defined by a cut surface, as described in section 3.3.1.

When considering a hybrid particle h during the evaluation of the constitutive model for a particle a, one has to check on which side of the cut h is located. If it is on the same side as a, the ghost will behave just like a standard particle, otherwise, it mirrors the displacement of the particle on the other side of the cut, thus behaving like a ghost particle for a stress–free boundary. (compare figure 3.3)

Contrary to before, where a ghost particle only has links to direct neighbors, a hybrid particle keeps track of *every* particle it can be consulted by, and stores the appropriate action to take. This is necessary as the condition of Eq. (3.3) can not be used for diagonal cuts between the particles. Thus, each hybrid particle h has a list

[2] Equaling to the 26 direct neighbors of a particle

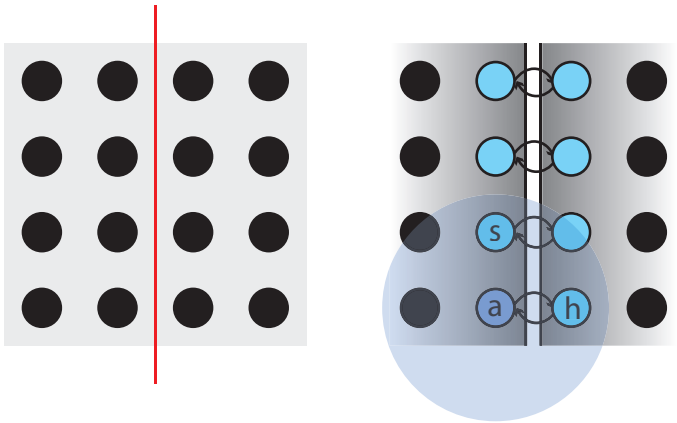

Figure 3-3: Cutting between particles using hybrid particles

of all its possible "invokers" (particles in the support radius of the simulation kernel), containing the following fields:

- Reference to invoker i (particle a for which the superposition of Eq. (2.1) is calculated).
- Action to be taken when invoked by i—this can either be none (h behaves like a material particle with respect to i) or copy (h behaves like a ghost particle with respect to i).
- A list of one or more reference particles that are used to copy the (possibly averaged) displacement from.

When a hybrid particle h is considered in the superposition of Eq. (2.1), it checks the list of invokers for the reference particle a, and then either returns its own displacement, meanings it is on the same side of the cut as a, or returns an averaged sum of the displacements of its reference particles (usually, only one).

In figure 3.3, the hybrid particle h keeps a list of all the particles within its support radius. These are all the particles a that can consult h when calculating the superposition of Eq. (2.1).
For all the particles residing on the same side of the cuts as h (5, 7, 8), the action is set to none, i.e. h behaves like a standard material particle. For the other particles, h will return the displacement of the particles indicated by the arrows. Thus, h will behave like a stress–free boundary in x–direction for particles 4 and 6, and like a stress–free boundary in y–direction for particles 2 and 3. For the outer edge particle 1, h will mirror itself, corresponding to case (b) of figure 2-2.
The information for the invokers is obtained in step 4 of the algorithm presented in subsection 3.3.2.

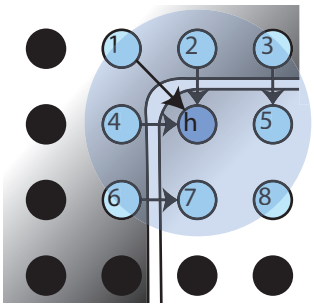

Figure 3-4: Hybrid particle h and its possible invokers

3.3.1 Cut Surface

The cut surface is defined as the surface that separates two parts of an object that are to be splitted [5]. It is inferred from the movement of the scalpel, as can be seen in figure 3-5.

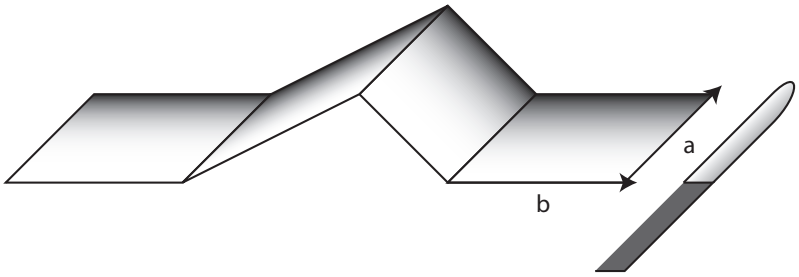

Figure 3-5: Cut surface defined by scalpel

While cutting, the algorithm has to decide which particles are to be separated by the cut surface. To simplify this calculation, the cut surface is assumed to be planar and rectangular.
The virtual surgery framework [4] does not support translation and rotation of the scalpel at the same time. In this thesis, only the generation of cut surfaces by translation is supported, but the addition of the rotation should not entail any major changes. Due to the way translation is handled by the framework, the cut surface can only be a parallelogram.

Thus, the cut surface is defined by two vectors $\mathbf{a}$ and $\mathbf{b}$, as well as a position $\mathbf{x}$. For $\mathbf{a}$, the cutting edge of the scalpel is chosen. $\mathbf{b}$ is defined as $\mathbf{b} = \mathbf{x_2} - \mathbf{x_1}$, where $\mathbf{x_1}$ denotes the start point and $\mathbf{x_2}$ the end point of cutting. These are obtained from the position of the scalpel at two subsequent time–steps.

Both $\mathbf{a}$ and $\mathbf{b}$ are assumed to be of unit length.

To test whether a particle a with position $\mathbf{x}$ resides at the cut surface, it is projected onto the surface using the dot product. That is, if both $(\mathbf{x} - \mathbf{x_1}) \cdot \mathbf{a}$ and $(\mathbf{x} - \mathbf{x_1}) \cdot \mathbf{b}$ are within the bounds specified by the cut surface's size, particle a is affected by the cut.

To guarantee the condition of having a rectangular planar surface, two measures were taken: first, new cut surfaces are generated after large enough movements (meaning the spacing of the particles h). Second, the parallelogram is approximated by a rectangle by means of Eq. (3.4) is used for $\mathbf{x_2}$. Refer to figure 3-6 for a graphical explanation.

$$\mathbf{x_{2a}} = \mathbf{x_2} - (\mathbf{a} \cdot (\mathbf{x_2} - \mathbf{x_1}))\mathbf{a} \tag{3.4}$$

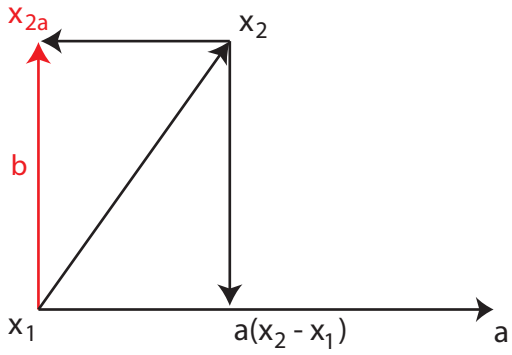

Figure 3-6: Graphical explanation of Eq. (3.4)

As the scalpel usually has only one main direction of moving, and cut surfaces can't become too large, Eq. (3.4) hardly impacts the results.

For the cutting process, the collision detection routine presented in [4] is used to check whether the scalpel is inside the organ. If this is the case, cut surfaces are generated as described above and stored in a list. This list is then handled as soon as possible by the actual simulation.

Note that the separation of the particles using the cut surface can either be based on the initial positions ξ or the current positions $\mathbf{x}(\xi, t)$. The disadvantage of the former is that the cut performed by the user is not reflected exactly in the material. The latter however can entail problems when the particles move heavily while moving the scalpel. This could result in particles being left out from the cut.
Thus, in the current version, cutting is done based on the particles' initial positions ξ.

Another note concerns the size of the time–step between sampling the positions of the scalpel. A too large time–step can introduce heavy lags for the user. On the other hand, a too small time–step can not only impact performance, but also affect the detection of diagonal edges. Two measures were taken: first, a cut surface is issued when the scalpel moves by an amount larger than $h\sqrt{3}$, where h denotes the particles'

spacing. Second, cut surfaces are chosen to overlap, i.e. if for the first cut surface, $\mathbf{b} = \mathbf{x_2} - \mathbf{x_1}$, the second cut surface will encompass $\mathbf{b} = \mathbf{x_3} - \mathbf{x_1} + \frac{1}{2}(\mathbf{x_2} - \mathbf{x_1})$. This improves detection of diagonal edges without introducing too large artefacts into the cut surfaces.

3.3.2 Splitting the Particles

The following summarizes the algorithm that is used to infer the linking information from the cut surface. Only the current cut surface is considered—previous cuts are inherent in the hybrid particles, as they keep track of what to do for their invokers.

1. Find particles whose projection fall into the cut surface, discarding particles that are farther away than the kernel's support radius.
2. Sort particles into two sets $\mathcal{A}$ and $\mathcal{B}$, depending on the side of the cut surface they are.
3. Find subsets $\mathcal{A}_{min}$ and $\mathcal{B}_{min}$ of particles that are nearest to the cut surface—these define the actual cut in the material.
4. For each particle $a \in \mathcal{A}$

   For each particle s within particle a's support radius, check particle s with respect to the cut surface.
   - Not at the cut surface or on same side as particle a: the relationship between the particles s and a doesn't change.
   - On other side of cut surface: particle s has to act like a ghost particle with respect to particle a. Thus:
     (a) If s is not yet a hybrid particle, convert particle s into a hybrid particle.
     (b) Find nearest neighbor of particle s on particle a's side of cut, $n \in \mathcal{A}_{min}$.
     (c) As an action when invoked by particle a, particle s takes the displacement $\mathbf{d_n}$ of particle n, i.e. behaves like a stress–free boundary particle.
5. Do the same for each particle $b \in \mathcal{B}$

Finding Edges

If two subsequent cuts are orthogonal to each other, something as seen in figure 3-7 can happen: even though the link information should represent an edge, the diagonal links are missing. This can result in instabilities in the simulation. To prevent this, the following algorithm is run over all particles after issueing a cut. Its goal is to find a diagonal edge when a particle has to adapt to two or more neighbors that are orthogonal with respect to to the particle whose field they mirror. This is the case for particle a in figure 3-7.

For each particle $a \in \mathcal{A}_{min} \cup \mathcal{B}_{min}$

1. Build set $\mathcal{S}$ of ghost particles with respect to a.

2. For each combination of two or three ghost particles $g_1 \dots g_3 \in \mathcal{S}$

   (a) Check that the particles $g_1 \dots g_3$ behave as ghosts to the same particle, i.e. take the displacement from the same reference particle r.[3]

   (b) Find diagonal particle h with position $\mathbf{x} = \mathbf{x}^{(n_1)} + \mathbf{x}^{(n_2)} - \mathbf{x}^{(a)}$ or $\mathbf{x} = \mathbf{x}^{(n_1)} + \mathbf{x}^{(n_2)} + \mathbf{x}^{(n_3)} - 2\mathbf{x}^{(a)}$.

   (c) Convert particle h into a hybrid particle.

   (d) h has to behave like a ghost particle with respect to particle a, thus h has to take the displacement $\mathbf{d_r}$ of particle r when invoked by particle a.

   (e) If the behavior of particle a is undefined for particle h, particle a should behave like an inner edge for particle h, i.e. particle a has to average over the displacements of the ghost particles $g_1 \dots g_3$ when invoked by h.

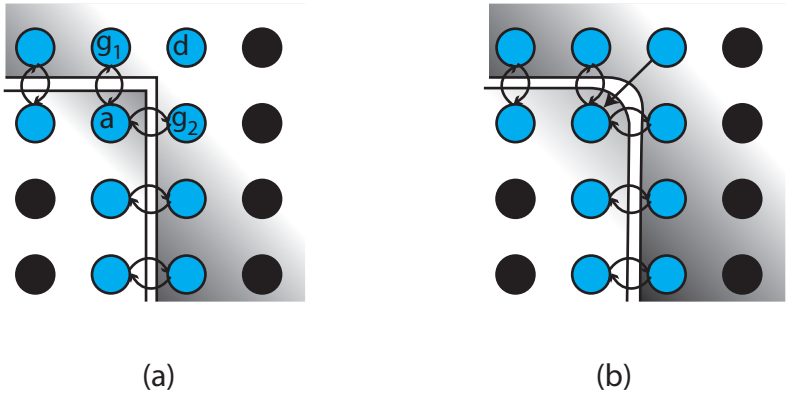

Figure 3-7: Finding edges

Stress–free boundary ghost particles

Actual stress–free boundary ghost particles are only indirectly affected by the cut, and thus will keep their neighbor at all times. That is, they behave exactly like their neighbor, and thus also adapt to cuts automatically when their neighbor becomes a hybrid particle.
This also applies to ghost particles in inner edges that have more than one neighbor: they return an averaged sum of the displacements of their hybrid neighbors.

[3] This usually is a, but doesn't have to be: consider figure 3-7. Assume that g_1 and g_2 lie in the same z–plane, but not in the same as a. r, however will lie in the same plane as g_1 and g_2, sharing the x and y coordinates with a.

Chapter 4

Implementational Issues

This chapter contains a few notes concerning the implementation of the algorithms. First, an overview of the most important classes is given. Next, the implementation of cell lists is described, with a focus on how to define new functions to operate on them. Finally, a performance improvement for the marching cubes algorithm is presented.

These descriptions concern the implementation for cutting by splitting, i.e. functions for converting are not included here due to the very different nature of the algorithms.

4.1 Class overview

4.1.1 Particle

The class `Particle` is an integral part to the simulation. It keeps track of the variables needed for the physically–based simulation, boundary conditions as well as displaying of the particles. The functions and variables offered by the class can be divided into six parts.

- **Simulation**. The variables $\mathbf{x}(\xi, t)$, ξ and V_{init} are the attributes of the particle. Furthermore, this also includes variables for the acceleration at the current and previous two time steps for Eqs. (2.19)–(2.20).
  Also, to support the improvement for the marching cubes algorithms, a threshold for a change flag can be set using `setEpsChanged`—when the change in position is larger than this, `getChanged` will return **`true`**. Use `updatePosOld` to reset the changed flag.

- **Integration**. `Particle` offers both a function `integrate` and `integrateEuler` to advance a particle according to Eqs. (2.19)–(2.20) and (2.21)–(2.22), respectively.

- **Invokers**. An auxiliary class, `ParticleInvoker` defines the invokers for a hybrid particle. As described in section 3.3, when considered by a particle a, the hybrid particle will search the list of its invokers for a, and then perform the action connected with it.
  Besides the functions to manage the set of invokers, the important ones for the

simulation are `getDisplacement` and `getVelocity`, which adjust the hybrid particle according to the invoker.

- **Neighbors**. For a stress–free ghost particle, a neighbor is a material particle inside the domain. In most cases, there is just a single neighbor, unless the ghost particle is situated in an inner edge (compare the third case of figure 2-2). Upon evaluation of a ghost particle, these neighbor(s) are consulted.

- **Fixed boundaries**. For fixed boundary ghost particles, the functions `setXPos` and `setBorderDisplacement` can be used to define a fixed boundary in the $y-z$ plane.

- **Coin3D**. Each particle also can hold a Coin3D data structure for it to be displayed as particle sphere. To initialize this data structure, the function `init` has to be called. To update the structure according to the particle's current position, use `render`. A color for the particle can be specified as well by using the function `setColor`.
  Use the `#define` statement `COIN3D` to include parts pertaining Coin3D in the compilation.

4.1.2 ParticleList

One of the most important classes in the simulation is the `ParticleList` class. It holds the set of all particles needed for the simulation, and contains functions for traversing all particles or a neighborhood of a given particle. The latter takes advantage of cell lists for increased performance.
Again, the functions can be divided into five sections.

- **Particles**. These are auxiliary functions to operate on single particles, including `makeHybrid`, which transforms a material particle into a hybrid particle, adding links to all particles within the support radius. `findParticle` can be used to find a particle according to its initial position.

- **Traversal**. Functions for traversing the entire list of particles with a functional operator are provided. Please refer to section 4.1.3 for a detailed description.

- **Cutting**. `cutBetween` will insert a cut in the particles, according to the algorithm given in 3.3.

- **Integration**. Functions `integrate` and `integrateEuler` are provided to integrate the entire list of particles. Note that the `integrate` automatically changes between Beeman and Euler–Cromer integration scheme if the topology changed (flag `topologyChangeCount`, e.g. modified by `cutBetween`).

- **Visualization**. On one hand, the function `getSpheres` will return a Coin3D separator with spheres for all particles. On the other hand, the functions `getDensity` and `getChangedDensity` can be used to calculate the density at a given location, needed for the marching cubes algorithm. Latter depends on the `getChanged` function of the `Particle` class introduced before.

4.1.3 Actions on all particles

To apply a function to a set of particles, the abstract class `ParticleAction` was defined. Its idea is to decouple the actual algorithm from the particle list data structure. Any descendent of this class can be passed to the traversal functions of the `ParticleList` class, which will then apply the functional operator to each particle.

```
class ParticleAction {
public:
  ParticleAction() { }
  virtual ~ParticleAction() { }

  // Applies the collected data to a particle
  virtual void apply(Particle& p) = 0;

  // Sets a reference particle
  virtual void setRef(const Particle* p) = 0;

  // Resets the aggregation variables
  virtual void reset() = 0;

  // Executes an action on the particle
  virtual void operator()(Particle& p) = 0;
};
```

Here is a quick rundown on how Eqs. (2.16)–(2.18) is evaluated using a descendant of `ParticleAction`, `ParticleForce`:

1. Set a reference particle a using `setRef`.
2. Invoke `ParticleList`'s `forParticleNeighborhood` with a as parameter. This will then apply the functional operator **`operator()`** to the entire neighborhood of a, summing up the field into private variables of the `ParticleForce` class.
3. Call `apply`, again with a as parameter. This applies the aggregated value (i.e. the force) to the particle. The particle can the update its position using Eqs. (2.19)–(2.20).

The `ParticleList` class actually offers a function `forEachDoForce` that combines the three above steps and applies them to *every* particle in the list.

After that, a call to the `integrate` method of `ParticleList` will update all particle positions.

4.1.4 Other classes

This is a quick overview of the other classes written within the course of this semester thesis. Most are based on `ParticleAction`.

- `DisplayKernel`. Class that contains the kernels that can be used for the calculation of the iso–surface, containing the kernel of Eq. (2.8), among others.
- `Knife`. Definition of the knife / cut surface, including functions for setting the distance (`distance`) and whether a particle projected onto the surface falls within the given bounds (`inside` and `setBounds`)
- `ParticleContainsNeighbor`. Checks whether the list of neighbors of a particle contains another particle.

- `ParticleDisplay`. Update the color field of a particle sphere according to its velocity.
- `ParticleIndex`. Given a vector of particles, this class converts between the index of an element and the reference to the element.
- `ParticleInvoker`. Defines a possible invoker for another particle.
- `ParticleLink`. Update link information of particles according to cut surface.
- `ParticleNearest`. Find particles nearest to another particle. This is used for linking stress–free boundary ghosts and for cutting by converting.

4.1.5 Integration into user interface

Integration into the user interface is done using callbacks—see `callbacks.cpp`.

The simulation itself runs in an idle callback. Here, `forEachDoForce` is invoked using an instance of `ParticleForce`, followed by `ParticleList`'s `integrate` function. A call to `updateOrgan` will then update the organ's surface.

A list of cut surfaces is kept in `CutSurfaces`, these are handled using `cutBetween`.

Cut surfaces are generated by the functions `cutter` and `issueCutSurface`, with the help of `insideOrgan`. This is done according to the description in section 3.3.1.

4.2 Marching Cubes

As described in the preceding semester thesis [4], the marching cubes algorithm is used to generate an iso–surface based on the density distribution described by Eq. (2.8).

Note that Eq. (2.8) is based on the Eulerian coordinates. Since these change after each step of the simulation, the surface has to be recalculated as well, which poses a heavy computational load on the CPU, both the evaluation of Eq. (2.8), as well as the marching cubes algorithm itself.

However, since usually only particles near to the cut are moving considerably, Eq. (2.8) is only recalculated for those particles whose change in location is significant.

Furthermore, calculation of is also sped up considerably by using cell lists—this can however yield artifacts due to largely displaced particles, since the cell lists are based on the initial position.

Chapter 5

Results

In this chapter, some experimental results are presented, followed by a performance test of the surgery simulator.

5.1 Experimental Results

The following two subsections contain a few examples of the cutting algorithms. For all examples, the same basic setup is chosen:
The particles are arranged in a cube with $9 \times 9 \times 9$ particles and an additional margin of 2 ghost particles on all sides, resulting in 2197 particles in total. The kernel's support radius is $0.75h$, thus involving only the 26 direct neighbors of a particle.
The cube is prestretched along the x–axis (denoted by the two quads in the pictures), with a small displacement of $d(x) = 0.1x$. The other boundaries are stress–free.
The time–step of the simulation is chosen to be 0.01.

For the visualization, the kernel's support radius is h, and the marching cubes' resolution is set to $20 \times 20 \times 20$. The iso–value of the iso–surface is set to 0.8, as 1 is the maximum due to the assumption $\rho = 1$.

5.1.1 Cutting by Converting

Figure 5-1 showcases the process of cutting by converting particles. An incomplete perforation of the cube is done along the z–axis. Two particles are left untouched. As material properties, the following settings were used:

- Poisson's ratio $\nu = 0$
- Young's modulus $E = 1$
- Relaxation time $T = 0.01$
- Gravity coefficient = 0

As expected, volume is lost at the location of the perforation. However, the hole doesn't collapse in itself, due to the stress–free boundary ghosts replacing the original material particles. The simulation stays stable, even with a low relaxation time of $T = 0.01$.

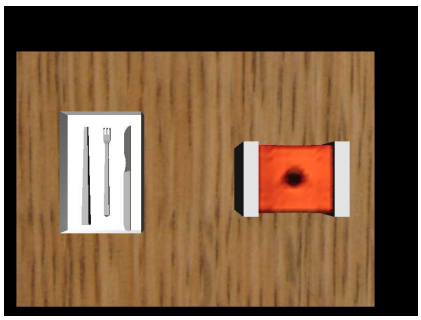

Figure 5-1: Cutting by converting particles

5.1.2 Cutting by Splitting Particles

Two examples are presented for this case. First, figure 5-2 demonstrates the process of cutting the cube apart along the z–axis. The material properties used were:

- Poisson's ratio $\nu = 0.3$
- Young's modulus $E = 0.7$
- Relaxation time $T = 0.2$
- Gravity coefficient = 0.1

Note the higher relaxation time—this is due to $\nu > 0$.
The cube is separated nicely due to the clamping of the material, and the simulation stays stable. At an optimization level of `O2`, the simulation ran at about 17 frames per second.

Another example involves a diagonal cut on the surface, seen in figure 5-3. Contrary to the cut before, this is not very symmetrical. Still, the simulation stays stable and produces the expected results. However, ν is set to 0 to increase the visibility of the diagonal cut. Thus, T is lowered again as well. Again, about 17 frames per second were achieved in the simulation.

- Poisson's ratio $\nu = 0$
- Young's modulus $E = 1$
- Relaxation time $T = 0.1$
- Gravity coefficient = 0.1

5.2 Performance

All performance tests were conducted on a Pentium 4 machine with 3 GHz. The setup was exactly the same as in the previous section. Compilation of the code was done

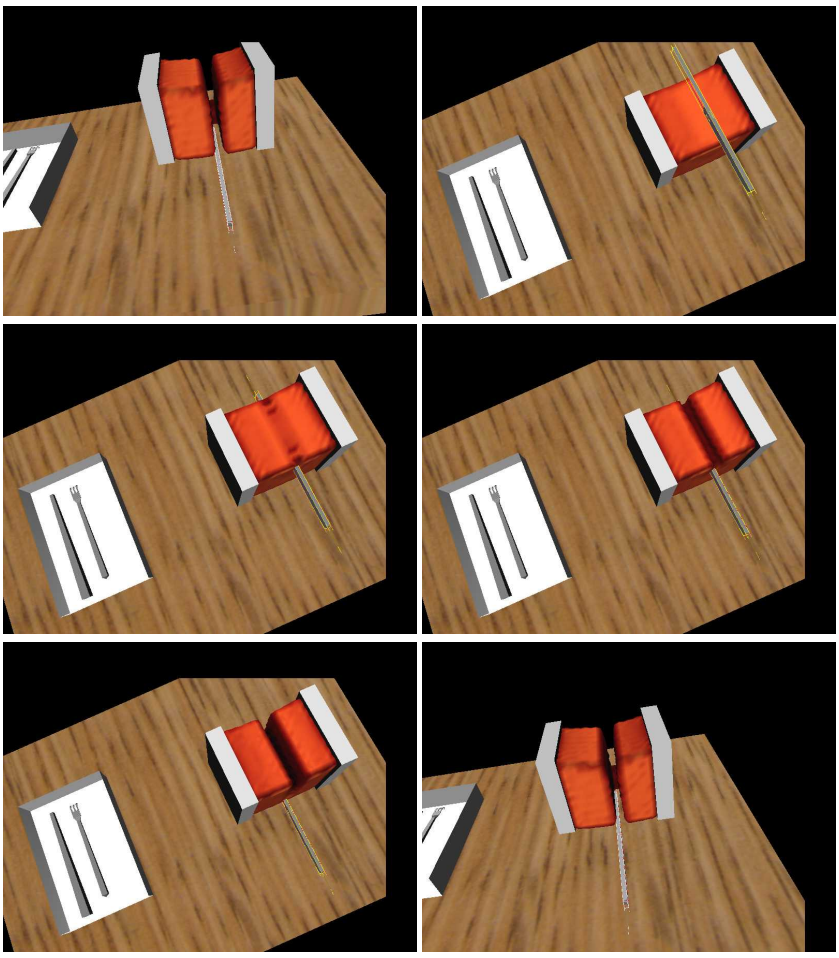

Figure 5-2: Cutting by splitting particles—vertical cut

with optimization in force (`-O1`[1]).
To statistically even out the overhead used by the initialization, the simulation was run for $2'000$ timesteps. No cutting was performed. Table 5-1 shows the first few lines of output of the GNU profiling program `gprof`. Red lines pertain the displayal of the iso–surface, green lines the actual simuluation.

In the profile of table 5-1, most of the time is spent in `getDensity`. This is due to the fact that the iso–surface is recalculated completely in every time step and involves 8000 evaluations of Eq. (2.8), as opposed to Eqs. (2.16)–(2.18), which are only evaluated 729 times. The marching cubes routines also take a fair share of the running time. Thus, a possible improvement is to update the organ's surface only every 2 or 3 timesteps, which can be seen in table 5-2 for every third timestep.

[1]Higher values did not work with the profiler. Still, -O2 showed a considerable improvement over -O1: 17 FPS instead of 10 FPS were possible for the example given in table 5-1.

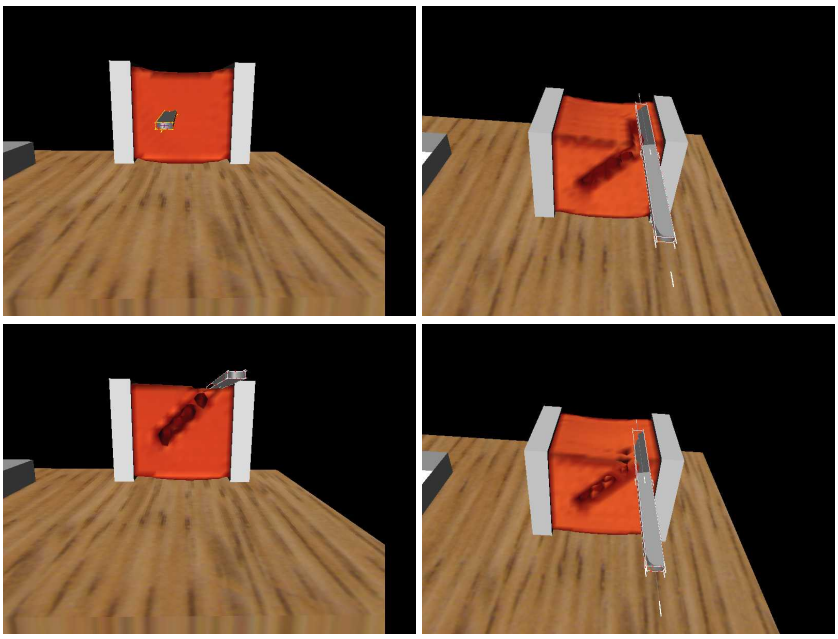

Figure 5-3: Cutting by splitting particles—diagonal cut

% time	Self sec	Calls	Self μs / call	Total μs / call	Name
70.89	105.50	16008000	6.59	6.59	ParticleList::getDensity
22.58	33.60	206188000	0.16	0.17	ParticleForce::operator()
2.15	3.20	2002	1598.40	1598.40	MarchingCubes::calcPoints
1.26	1.87	1458000	1.28	25.75	ParticleList::fPNE[2]
0.75	1.11	37908000	0.03	0.03	Particle::getDisplacement
0.65	0.96	37908000	0.03	0.03	Particle::getVelocity
0.61	0.91	2000	455.00	53293.59	MyUserData::updateOrgan
0.38	0.56	2002	279.72	279.72	MarchingCubes::calcGradients
0.16	0.24	2002	119.88	119.88	MarchingCubes::calcIndexes
0.13	0.20	1456542	0.14	0.14	Particle::integrate
0.11	0.17	2001	84.96	89.96	Organ::loadData
0.09	0.13				idleCB
0.08	0.12	1458000	0.08	0.08	ParticleForce::apply
0.08	0.12	2000	60.00	18910.00	ParticleList::forEachDoForce
0.03	0.05	4538580	0.01	0.01	ParticleList::operator[]
0.02	0.03	1458000	0.02	0.03	ParticleForce::reset

Table 5-1: Profile (10.6 FPS)

Another method would be to use the performance improvement suggested in section 4.2. In the test shown in table 5-3, the iso–surface is updated in every timestep, but only particles whose position is subject to a change larger than 0.001 are recalculated.

% time	Self sec	Calls	Self μs / call	Total μs / call	Name
46.19	35.45	5344000	6.63	6.63	ParticleList::getDensity
45.33	34.79	206188000	0.17	0.18	ParticleForce::operator()
2.68	2.06	1458000	1.41	26.64	ParticleList::fPNE
1.34	1.03	669	1539.61	1539.61	MarchingCubes::calcPoints
1.30	1.00	37908000	0.03	0.03	Particle::getDisplacement
1.29	0.99	37908000	0.03	0.03	Particle::getVelocity
0.44	0.34	667	509.75	53683.42	MyUserData::updateOrgan
0.43	0.33	669	493.27	493.27	MarchingCubes::calcGradients
0.25	0.19	1456542	0.13	0.13	Particle::integrate
0.22	0.17	669	254.11	254.11	MarchingCubes::calcIndexes
0.18	0.14				idleCB
0.08	0.06	668	89.82	89.82	Organ::loadData
0.07	0.05	1458000	0.03	0.03	ParticleForce::apply
0.07	0.05	2000	25.00	19500.00	ParticleList::forEachDoForce
0.04	0.03	1458000	0.02	0.02	ParticleForce::setRef
0.03	0.02	2916002	0.01	0.01	Vec3D::set
0.01	0.01	1575891	0.01	0.01	ParticleList::operator[]
0.01	0.01	1458000	0.01	0.02	ParticleForce::reset

Table 5-2: Profile with skipping of marching cubes (19.3 FPS)

% time	Self sec	Calls	Self μs / call	Total μs / call	Name
49.55	42.63	15992000	2.67	2.67	ParticleList::getChangedDensity
39.15	33.68	206188000	0.16	0.17	ParticleForce::operator()
3.86	3.32	2002	1658.34	1658.34	MarchingCubes::calcPoints
2.09	1.80	1458000	1.23	25.53	ParticleList::fPNE
1.24	1.07	37908000	0.03	0.03	Particle::getDisplacement
0.92	0.79	2000	395.00	21859.95	MyUserData::updateOrgan
0.79	0.68	37908000	0.02	0.02	Particle::getVelocity
0.64	0.55	2002	274.73	274.73	MarchingCubes::calcGradients
0.53	0.46	2002	229.77	229.77	MarchingCubes::calcIndexes
0.23	0.20	1456542	0.14	0.14	Particle::integrate
0.22	0.19	2001	94.95	94.95	Organ::loadData
0.19	0.16				idleCB
0.15	0.13	1458000	0.09	0.09	ParticleForce::apply
0.14	0.12	2000	60.00	18770.00	ParticleList::forEachDoForce
0.12	0.10	16000	6.25	6.25	ParticleList::getDensity
0.07	0.06	4538580	0.01	0.01	ParticleList::operator[]
0.03	0.03	2916002	0.01	0.01	Vec3D::set
0.03	0.03	1458000	0.02	0.04	ParticleForce::reset

Table 5-3: Profile with performance improvement (12.8 FPS)

Note that the kernel used for the visualization, Eq. (2.8), is dependent on the dis-

tance between particles, and thus involves the computationally expensive calculation of a square–root function. A performance increase could be achieved by using another display kernel that only depends on the squared distance. As a test, the square–root was removed from the code for Eq. (2.8), and the test of table 5-3 was repeated. While the resulting image is unusable, less time is spent in the function `getChangedDensity`—38.71 seconds instead of 42.63, and the framerate increases from 10.6 to about 14–15 frames per second.

Yet another—and better—possibility to increase the performance of the visualization is to only recalculate the iso–surface at grid points where the iso–value was within certain bounds. This saves a lot of computation especially in the inside of the organ. Still, as bigger changes can occur, the iso–surface has to be recalculated completely after a certain number of time–steps.
This was tested for the iso–value of 0.8. Grid points that yield a value within $0.8 - \epsilon$ and $0.8 + \epsilon$ are recalculated in the next time–step, and every 10 time–steps, the entire surface is recalculated. Here, ϵ is chosen to be 0.19. This yieds a considerable performance improvement: compared to the profile in table 5-1, only 54.29 instead of 105.5 seconds are spent in `getDensity`. The framerate reaches 16 frames per second, and the surface animates smoother than in the improvements suggested before.
The only disadvantage seems to be a slightly larger lag in the animation when cutting the surface.

The simulation itself also takes a considerable amount of the execution time. As `-O1` introduces inlining of code fragments, some functions are not listed in the above profiles, especially the time–consuming call to `Vec3D`'s `length()` function, needed for the calculation of the kernel. A solution here might be to tabulate the square–root function used in `length()`, which would work especially well when considering only initial positions in the evaluation of Eqs. (2.16)–(2.18), as was done in this semester thesis.

Lastly, a profile with a horizontal cut is presented in table 5-4. Other than the cut, the settings are equivalent to the one used for the profile from table 5-1.

As expected, the framerate doesn't drop considerably, since no modifications to the topology have to be applied, as is the case with cuts in tetrahedral meshes. However, significantly more time is spent in the routines `getDisplacement` and `getVelocity`, as each hybrid particle first has to look for the appropriate neighbor in a list. This could be improved considerably by the use of a hash–table, for instance.

% time	Self sec	Calls	Self μs / call	Total μs / call	Name
68.65	105.44	16008000	6.59	6.59	ParticleList::getDensity
23.16	35.57	206188000	0.17	0.20	ParticleForce::operator()
2.06	3.16	37908000	0.08	0.08	Particle::getDisplacement
1.78	2.73	2002	1363.64	1363.64	MarchingCubes::calcPoints
1.33	2.05	1459467	1.40	29.24	ParticleList::fPNE
1.22	1.87	37908000	0.05	0.05	Particle::getVelocity
0.53	0.81	2000	405.00	53193.61	MyUserData::updateOrgan
0.44	0.67	2002	334.67	334.67	MarchingCubes::calcGradients
0.29	0.44	2002	219.78	219.78	MarchingCubes::calcIndexes
0.16	0.24	1455084	0.16	0.16	Particle::integrate
0.11	0.17	2001	84.96	84.96	Organ::loadData
0.10	0.15				idleCB
0.07	0.10	2000	50.00	21418.55	ParticleList::forEachDoForce
0.06	0.09	1458000	0.06	0.06	ParticleForce::apply
0.01	0.02	4542054	0.00	0.00	ParticleList::operator[]
0.01	0.02	73720	0.27	0.27	ParticleLink::operator()
0.01	0.01	2916002	0.00	0.00	Vec3D::set
0.01	0.01	1458000	0.01	0.01	ParticleForce::reset

Table 5-4: Profile with diagonal cut (10.3 FPS)

Chapter 6

Conclusion

In this semester thesis, a framework for virtual surgery was extended to support the physically–based simulation of a solid material, later to become a biological material. The material was modelled using the generalized Hooke's Law, extended by the Kelvin–Voigt damping model. In contrast to other publications, Smoothed Particle Hydrodynamics (SPH) were chosen to discretize the partial differential equations. The concept of ghost particles was used as an alternative to one–sided differentiation to account for boundary conditions. Both fixed and stress–boundaries were considered. With the particles arranged in a cube, we achieved a stable simulation of the material, even with very low relaxation times.

Next, the concept of ghost particles for stress–free boundaries was extended to support more than one neighbor. This allowed for ghost particles to reside between multiple material particles and behave like a stress–free boundary for each of them.
This technique was used in a first approach for cutting the material by converting material particles inside the scalpel into ghost particles. The algorithm developed for this is only suited for small kernel radii, and has the major disadvantage that volume is lost by converting particles.
In a second step, hybrid particles were introduced as particles that can behave both like ghost and material particles, depending on the situation. Cut information was now inferred from a cut surface, defined by the movement of the scalpel.

In both cases, sufficient damping is needed to obtain a stable simulation.

Both cutting algorithms work well and demonstrate the possibility to cut material apart within the SPH framework. This has the major advantage that no changes to the geometry of the computational elements has to be made, as it is the case with other grid–based methods, like FEM.

Chapter 7

Future Work

This semester thesis barely scratches the surface of the possibilities of a virtual surgery simulator. While the algorithms developed within this thesis are promising, much more attention to detail is needed for a realistic simulation.

7.1 Performance

The realism of the surgery simulation is heavily dependent on the number of particles used, among others. This figure is limited by the currently available processing technology. For a realistic simulation, many more particles than the 729 used above need to be taken into account.

Performance is not only hindered by the actual evaluation of Eqs. (2.16)–(2.18), but also by the marching cubes algorithm. For the former, a method like adaptive subdivision could already yield a noticeable increase in performance, since during the surgery, only the few particles near to the cut move heavily. For the latter, a small performance improvement is already presented in section 4.2.

Ultimately, the goal would be to use point–based rendering, as presented in [11]. This method could directly rely on the particles, without the need of first creating an iso–surface needed for the rendering of triangles. Still, normals of the point–based image elements, called surfels, have to be calculated, and this type of rendering is not yet supported by today's graphics cards. Furthermore, many more particles ($\approx 100'000$) would be needed as well to guarantee a smooth surface.

7.2 Physically–based simulation

For simplicity's sake, the evaluation of Eqs. (2.16)–(2.18) is based on the particles' initial positions. To base it on the current positions, remeshing has to be introduced, as presented in [7]. Even though the link information for the cuts as presented here is also based on the initial positions, this should translate well to remeshing. Possibly, a re–evaluation of the algorithm according to the cut surfaces has to be done for this.

The cutting algorithms presented here have been developed with a kernel support radius smaller than $0.75h$. If cutting by converting is still considered, changes have to be applied to get it working for larger support radii. The algorithm for cutting by splitting was already designed with a larger support radius in mind. Still, for inner edges as seen in case (c) of figure 2-2, more diagonal particles have to be considered for a particle inside the edge. It is not quite clear on how to combine the displacements of the ghost's neighbors in these cases. Most likely, a weighted sum depending on the distance will yield the best results.

For all the tests performed, the particles were arranged in a cube. When a general surface needs to represented using the methods presented here, good care has to be taken with respect to boundary conditions. Ghost particles have to be placed all around the organ, with linking done accordingly to figure 2-2. Again, larger support radii will complicate things here.
The class `ParticleList` already offers a function `linkGhost` that searches for the nearest material particles of a ghost particle.

As for fixed boundaries, only the $y - z$ plane is supported. This needs to be generalized. Furthermore, cuts at fixed boundaries are not taken care of so far. This should be an easy extension of the cutting algorithm, involving a fixed ghost to behave like a stress–free boundary ghost depending on the particle its invoked by.

Lastly, the model used for the tissue is not completely correct. One one hand, ρ was chosen to 1 as a simplification. On the other hand, a linear model is used. For a more realistic simulation, this should be extended to a non–linear model. This would only involve changes to the `ParticleForce` class, and should have no impact whatsoever on the cutting algorithm itself.

7.3 User interaction

The cutting process as presented here is rather rough. No matter how the scalpel is moved through the organ, it will cut the tissue. In reality, different movements entail different actions, compare [5]. Before cutting, there is the process of penetrating the tissue. Movement orthogonal to the blade's cut surface should not cut particles apart, but rather push them in a direction. To achieve this, some kind of force feedback is needed, since the user shouldn't be able to push the scalpel through particles.

Last, the framework provided by [4] offers more instruments than just the scalpel. These also need to be accounted for in future works.

Chapter 8

Contents of CD

This chapter gives a quick overview of the directory structure of the accompanying CD.

- **Coin-2.2.1**. Include files for Coin3D.
- **CutBetween**. The most current version of the surgery simulator, supporting the algorithm for cutting by splitting presented here.
  - **CollisionControl**. Files needed for the collision detection routine.
  - **data**. Data for the Coin3D scene.
  - **draggers**. Data for the draggers.
  - **GUI**. Files for the graphical user interface developed in [6].
  - **images**. Icons used by the graphical user interface.
  - **Libs**. Shared libraries for Coin3D and SoQt.
  - **Movies**. Directory movies will be stored in.
  - **Screenshots**. Directory screenshots will be stored in.
  - **Simulation**. Files for the actual simulation. This encompasses all files and classes presented in chapter 4.
- **Improvement**. Updated C files for the last improvement of the marching cubes routines discussed in 5.
- **Presentation**. The LaTeXfiles for the presentation, including a few movies.
- **Report**. The LaTeXfiles for this report.
- **simage-1.6.0**. SImage library needed for loading textures.
- **VirtualSurgeryKDev**. The KDevelop project used first. This support cutting by converting, however, the algorithm used for cutting by splitting is not fully developed at this stages and produces instabilities. As the approach of using Eq. (3.3) was abandoned for cutting by splitting, this project is not up to date.

Bibliography

[1] D. Beeman. Some multistep methods for use in molecular dynamics calculations. *Journal of Computational Physics*, 20:130–139, 1976.

[2] D. Bielser. A framework for open surgery simulation. http://e-collection.ethbib.ethz.ch/show?type=diss&nr=14900, 2003.

[3] T.J. Chung. Applied continuum mechanics, 1996.

[4] S. da Silva. A virtual surgery environment, 2003.

[5] M. Gernss. Real–time interaction with tetrahedral meshes, 2004.

[6] I. Guajana. A graphical user interface for virtual surgery, 2004.

[7] S. Hieber, J.H. Walther, and P. Koumoutsakos. Remeshed smoothed particle hydrodynamics simulation of the mechanical behavior of human organs, 2004.

[8] G.A. Holzapfel. Nonlinear solid mechanics: a continuum approach for engineering, 2001.

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

[10] J.P. Morris, P.J. Fox, and Y. Zhu. Modeling low reynolds number incompressible flows using sph. *Journal of Computational Physics*, 136:214–226.

[11] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. *Proc. SIGGRAPH 2000*, pages 335–342, July 2000.

22nd July 2004