

Real-Time Cloth Simulation

Robert Bargmann

SSC

Diploma Thesis

Summer 2003

Supervisors

Prof. Markus Gross
ETHZ / CGL

Prof. Sabine Süsstrunk
EPFL / LCAV

Dr. Matthias Teschner
ETHZ / CGL



Abstract

The field of cloth-modeling has been widely investigated during the last decade. Many solutions have been proposed for the different aspects related to that field, ranging from fast cloth integration methods to complex collision handling. However, it is still challenging to perform simulations in real-time without compromise.

In this thesis, we base our work on fast collision detection methods which use *Layered Depth Images* (LDIs). We propose a new approach for handling cloth self-collisions and collisions between cloth and deformable volumes.

This work mainly focuses on real-time performance and we achieve to run simulations at a frame-rate of up to 30 fps, while performing cloth integration and manipulating self-collisions and collisions on deformable bodies.

Zusammenfassung

Das Gebiet der Kleider-Modellierung ist während der letzten zehn Jahre weit erforscht worden. Viele Lösungen sind für die verschiedenen Aspekte dieses Themas vorgeschlagen worden, von schnellen Integrationsmethoden bis hin zur Behandlung komplexer Kollisionen. Es ist jedoch immer noch schwierig, Simulationen in Echtzeit durchzuführen.

Diese Diplomarbeit begründet sich auf schnelle Kollisionsdetektionsverfahren, welche sogenannte "Layered Depth Images" (LDIs) verwenden. Wir schlagen ein neues Verfahren für die Behandlung von Kollisionen zwischen einem Tuch und verformbaren Körpern vor. Weiterhin werden Selbstkollisionen behandelt.

Diese Arbeit konzentriert sich hauptsächlich auf Echtzeitanforderungen. Wir erzielen Simulationen mit einer Geschwindigkeit von bis zu 30 fps.

Contents

Contents	1
1 Introduction	5
1.1 Assignment	5
1.2 Main Contributions	6
1.3 Organization	7
2 State of the Art	9
2.1 Volumetric Collision Detection for Deformable Objects [HTG03] (2003)	9
2.2 Selected References	10
2.3 Summary	16
3 Collision Detection	19
3.1 Introduction to the Collision Control Library	19
3.2 Growing the LDI	20
3.2.1 Dilation	21
3.2.2 Stretching	22
3.2.3 Merging	22
3.3 Three-directional LDIs	22
3.4 Optimizing the 3-directional LDI	25
3.5 Discussion	26
3.5.1 Three LDIs versus 3-directional LDI	26
3.5.2 Choosing the Direction for Rasterizing	27
4 Collision Response	29
4.1 Triangle-to-LDI Association	30
4.2 When is a Cloth-point Inside the Volume?	30
4.3 Projecting the Cloth-point	31
4.4 Affecting the Velocity	33
4.5 Two-side Detection	34
4.6 The Algorithm	37
4.7 Discussion	38
4.7.1 Aura	38

4.7.2	LDI-resolution	38
5	Self-collision	41
5.1	Considerations	41
5.2	Self-collision Detection	43
5.2.1	The Approach	43
5.2.2	Improvement	44
5.3	Self-collision Avoidance	45
5.3.1	The Approach	45
5.4	Self-collision Response	47
5.4.1	Changing Positions	47
5.4.2	Changing Velocities	49
5.5	Putting All Together	51
5.5.1	Skipping Neighbors	52
5.5.2	Future After Past	52
5.6	The <i>Self-Collision</i> Algorithm	53
5.6.1	The Procedure	53
5.6.2	First Collision Occurrence	54
5.6.3	ϵ -distance	55
5.6.4	Velocities	55
5.6.5	Attribution	57
5.6.6	Marking Triangles and Vertices	58
5.6.7	The Algorithm	58
5.7	Discussion	58
5.7.1	When to Change Velocities	58
5.7.2	Friction	60
5.7.3	Side-collisions	60
6	Experiments	63
6.1	The Environment	63
6.1.1	Most Important Simulation Parameters	63
6.1.2	Working with Deformable Volumes	64
6.2	Discussion	65
6.2.1	The Measurements	66
6.2.2	Analysis	68
6.3	Application: the Walking Avatar	72
6.3.1	The Poncho	73
6.3.2	Discussion	73
7	Future Work	77
7.1	Extensions to Self-collision	77
7.2	Multiple Cloth Pieces	79
7.3	Colliding Volumes	80
7.4	Multiple-LDI Structure	82

8 Conclusions	83
Acknowledgment	84
Bibliography	85
A The Libraries	87
B Measurements	91
C Code Description	95

Chapter 1

Introduction

We introduce this thesis by giving the assignment we had to fulfill, we then present our contribution to the field of cloth-modeling and finally we describe the organization of the chapters.

1.1 Assignment

The simulation of cloth is based on character animation, deformable modeling, and collision handling. While off-line cloth simulation is a well investigated field, it is still an unsolved problem to combine all components in a real-time system.

The CGL has investigated methods for real-time deformation of very complex mass-spring models and methods for real-time volumetric collision detection of geometrically complex deformable objects. Based on these existing techniques and on the public domain 3D character animation library “Cal3D” it is intended to build a system for real-time cloth simulation.

The following problems should be addressed during the project:

- Familiarizing with the basic components: OpenInventor™, the deformable modeling environment, the volumetric collision detection library, and the character animation library “Cal3D”.
- Integration of “Cal3D” and volumetric collision detection into the existing deformable modeling framework.
- Realization of collision detection for animated characters and cloth using existing library.
- Implementation of collision response for cloth.
- Generation of appropriate cloth models which can be used with “Cal3D” characters.

- Performance optimization of the system.
- consideration of cloth self-collision.

1.2 Main Contributions

In the last decade, the field of cloth modeling has been widely investigated. Many aspects are related to it, from cloth-modeling integration methods to flexible bodies deformations. Explicit and implicit cloth integration methods have been developed. Efficiency and speed of simulations could be improved by using larger time steps without losing stability. Moreover, a lot of different approaches to handle collisions and self-collisions have been proposed, some based on hierarchical bounding volumes structures, or space discretization methods.

It is challenging, to perform simulations in real-time without compromise. Collisions and especially self-collisions handling have proven to be the bottleneck in the simulation processes and should be resolved to obtain simulations with high frame-rates. Also, deformable models are rarely considered in such simulations, as their geometry is not known *a priori*, which requires a precise tool for detecting, at any time step, where collisions can occur.

This thesis presents an approach to cloth simulation which will consider all of the following aspects:

Cloth modeling . The cloth modeling method is based on a mass-spring system with numerical integration which was already developed and implemented by Matthias Teschner, from CGL, ETH, Zurich.

Deformable models . We will work with models which undergo a geometrical deformation: the models will follow a deterministic deformation function but their geometry will not be affected by collisions. However, we will not pre-compute the mesh deformations and we will get the knowledge of their updated geometry after every time step.

Two side collision detection . We will take advantage of a new collision detection approach. This method uses *Layered Depth Images* (LDIs) which was developed by Bruno Heidelberger from CGL, ETH, Zurich. This tool will allow us to make fast and efficient collision detections on any deformable body, and we will also be able to detect the side of the meshes on which collisions occurred.

Collision response on open meshes . Along with the detection phase, we developed an efficient response method based on the relative velocities of two colliding entities. This method will allow us to work with open meshes.

Self-collisions . We will make further extensions to the LDI-tool and will propose a self-collision algorithm which performs in real-time.

Applications for our cloth-simulation system are of course games, where it would be possible to integrate clothes on the different characters involved, but also for the movie industry. Preliminary fast simulations will allow to have a more precise idea of what the final rendering of scenes might look like, without having to wait for minutes-long off-line computations. It will even be possible to visualize simple scenarios in real-time.

1.3 Organization

We tried to keep the organization of this thesis in the order as our development evolved. We begin by presenting the state of the art in the field of cloth-modeling (Chapter 2) by giving a short resumé of some publications to illustrate the different aspects that are considered in setting up a cloth simulation system, and also to show how far research has gone today. Our approach is mainly based on using a new tool for detecting collisions in real-time. This tool, available in the *Collision Control* library, had to be adapted in order to be able to properly detect collisions in our system. The modifications we brought to that library are described in Chapter 3.

Once we had a reliable tool for detecting collision between a cloth and a volume, we could implement the *collision response* phase (Chapter 4), which is the second aspect to consider in the *collision process*.

The further step was to implement a *self-collision* algorithm for the cloth. We made again use of the adapted tool which required further modifications (Chapter 5).

In Chapter 6 we then present our experiments and study the efficiency of our system, by analyzing simulations of two different scenarios. We will end this thesis by proposing further investigations that shall improve our actual results (Chapter 7) and finally, in Chapter 8, draw the conclusions of our work.

Chapter 2

State of the Art

Today, the field of cloth modeling has been investigated for about ten years. Many papers have been published covering many different aspects, going from cloth integration techniques to volume deformations or cloth self-collision.

This work concentrates on solving *collision detection* and *response* by an approach based on a new tool, the *Layered Depth Images* (LDIs). This tool was developed by Bruno Heidelberger from the *Computer Graphics Laboratory* at the ETH in Zurich, Switzerland.

In this chapter we give a short resumé of the technical report presenting the LDI tool and also of a selection of papers to illustrate the current state of the art in the field of cloth simulation.

2.1 Volumetric Collision Detection for Deformable Objects [HTG03] (2003)

This report presents the collision detection algorithm we will use in this work. The algorithm is based on a “Layered Depth Image” (LDI) representation of deformable volumes for fast collision detection. It is particularly efficient, as it uses the hardware graphical acceleration through OpenGL for the LDI generation, which is an important factor for real-time processing.

Volumes are required to be closed for the algorithm to generate their LDI representation and a rendering method is needed to compute these representations. As an output we obtain an explicit representation of the volume allowing fast detection of vertices inside the volume.

Also, the library detects collisions between two volumes. For detecting the collision intersection, the algorithm first determines the bounding-box on the volumes and check whether they intersect. In that case, it will only consider that intersection for the LDI discretization. In a last step, the algorithm queries for *entry* and *leaving points* (Fig. 2.1) of the intersection and can then create the LDI representation.

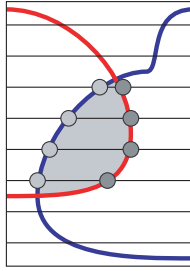


Figure 2.1: Collision queries for two intersecting volumes

2.2 Selected References

From the many papers and reports written over the last decade on cloth-simulation, we present here, in this second part of the chapter, a selection that illustrates the different aspects considered in that field and the approaches that have been developed.

Dynamic Simulation of Non-Penetrating Flexible Bodies [BW92] (1992)

The authors present an implementation for first- and second-order polynomially deformable bodies. This will allow simulating flexible bodies, but we will not go into further details here as we only consider non-flexible deformable bodies. However, we will focus on paragraphs of this article related to *non-penetration* and *colliding contact*.

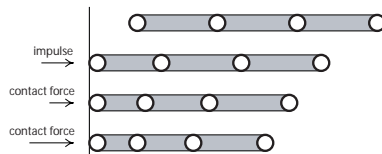


Figure 2.2: Response for flexible bodies

The *non-penetration* constraint forces to keep every foreign point outside a body. When we detect that a point has penetrated, we must pull it back to the surface. In our implementation, this aspect is considered. We first pull the point out and then perform the *collision response*.

In this paper, the collisions are not considered as instantaneous. When two bodies collide, a finite number of vertices are in contact with the other body and are to be considered. To simulate a collision between two flexible bodies, the approach is the following:

1. When a point penetrates a foreign body it is pulled outside

2. Its relative velocity to the foreign body is canceled.
3. Defined physical constraints of the bodies including *stiffness* will affect the neighbors of the point that has penetrated as shown in Fig. 2.2.
4. Finally the body will be compressed locally and then will return to its rest state. It is only when the volume has recovered its rest state that the colliding point will separate from the body.

This paper describes in a very rigorous fashion deformable models and collision response for flexible bodies. Although, these aspects are not compatible with our needs, because our deforming bodies do not react to collision with the cloth, it could be considered in an extension of our current work.

Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects [VCT95] (1995)

This paper mainly treats *cloth integration*, *collision detection* and *self-collision detection*.

The integration method is similar to the one used for our work. Two kinds of strains have to be pointed out:

The elastic and shearing strains : this concerns the strains that are working onto the edges of the triangles. It defines the *elasticity* of the cloth.

The bending strains : It is based on the curvature evaluated from the angle between the triangles. It defines the *rigidity* of the cloth.

The second part of the paper focuses on *collision* and *self-collision detection*. The self-collision algorithm is presented in another work from the same authors [VT]. Although we have not yet written on self-collision, it is an aspect that we keep in mind. Indeed, we might want in a later stage of this diploma thesis, resolve this aspect of cloth simulation.

Regarding collision detection, the paper describes different parameters to look at relatively to the kind of collision we deal with:

Proximities : triangle-to-vertex, edge-to-edge, edge-to-vertex or vertex-to-vertex

Interferences : occurs whenever an edge crosses a triangle

Also, different approaches are used for detecting collision:

Remnant proximities : keep trace of the proximities between triangles so that when they occur again we can detect them

Kinematical tracking : when a proximity is detected, one considers the current velocities of the triangles and deduces if a collision occurred previously.

Consistency checking and correction : previous approaches are not deterministic, so adding a consistency checking is necessary. It is based on *collision regions*.

Except for the integration, the approaches presented here are slow and roughly described. In our work, the collision detection uses the LDI which allows us to perform it in real-time.

Collision Response [Hec97] (1997)

The *collision response* presented in this article is for instantaneous collisions and does not consider forces involved in the collision process. These forces are important for collisions of flexible bodies, where the elasticity of the bodies aims at bringing the bodies back to their rest state and would produce an acceleration of the bodies. This is therefore only usable for rigid bodies.

The approach here is however similar to ours even if we deal with deformable objects. We must clearly make the distinction between *flexible* and *deformable* volumes in that the flexible ones can change their geometry through collision or any external force but they will return to a rest state. Deformable models change their geometry according to a function of time and collisions can be treated as those for the rigid bodies.

A difference with our approach is that the collision detection in this article is supposed to detect whenever two bodies are *in contact*, i.e. touching but have not penetrated each other. The way we pull a point outside of a volume is not described.

The article ends by considering the angular velocity of the bodies. Admitting that a body always rotates around its center of mass, and knowing the geometry of the body, one can always tell the velocity of any point of the body. But in our case, the relative position of the points with a rotating axe changes over time. We compute the velocity of the faces of our volumes from the distance they traversed since their last know position. Thus, the component of the velocity due to rotation is included.

Collision and Self-Collision Detection: Efficient and Robust Solutions for Highly Deformable Surfaces [VM95] (1998)

Self-collision is time costly and few algorithms offer at the same time accuracy and efficiency. It is probably the bottleneck of all cloth simulation regarding computation time and no bounding volume approach can be useful.

This article proposes an algorithm based on surface discretization allowing to handle complex situations, including wrinkling. Throughout the list of our selected articles, it is actually the most often used algorithm for self-collision.

The approach is based on optimizing common collision detection techniques by lowering the test number in delimiting regions that can not be self-colliding.

Such regions have the property that the normals all their faces have a positive dot product with some common vector. If the projection of the border of the region onto an orthogonal plane to the common vector does not self intersect, then the region is not self-colliding. Extensions allow also avoiding collision tests between adjacent regions.

Finally, these properties coupled with a fast hierarchical algorithm, produces performances where the time spent on self-collision detection is proportional to the number of colliding elements.

The second half of the article goes further beyond the detection of collision but presents a way to react to it. This is called the *orientation consistency* which will pull the surfaces back, once a proximity or collision is detected, to the correct sides of the surfaces. The approach taken for this part, is based on *consistency tracking* when we store the history of the surfaces location so to remember where their last position was before the collision.

Large Steps in Cloth Simulation [BW98] (1998)

This paper presents an approach for implementing cloth simulation. The advantages with this method are that they allow great time steps between two consecutive integration steps. We will not discuss this aspect here as it is not a part of the cloth modeling we are investigating. However, there is a paragraph on collision which is of interest for us.

For the detection phase of the collision, the authors refer to another article of theirs [BW92] that is adequate for flexible bodies as clothes are. In the response phase, when two faces are detected to be colliding, a strongly damped spring is inserted between them to push them apart. This response might be of great interest for our work as we are currently setting up a new approach for detecting self-collisions but have not thought yet of an appropriate response.

Interactive Animation of Cloth-like Objects for Virtual Reality [DMB] (2000)

In this more recent paper, a modification to the largely used mass-spring model for simulation cloth is presented. It introduces the *implicit integration* which replaces the forces at time t by the ones at time $t+dt$. This alternative

allows greater stability in the system and thus larger time steps. The first three parts of the paper discuss the *implicit integration*.

The fourth part is devoted to external forces. Beginning by considering forces induced by wind or fluid flows, collision is then considered.

Nevertheless, they use a voxel-discretization of the bounding-boxes approach which does not consider self-collision of the cloth. The collisions are related to the voxels who have each of them been associated to a closest vertex of the body. This approach is similar to ours as we use the LDI-sticks instead of the voxels. It differs in that we make a triangle association rather than a vertex association. This difference gives us a more accurate projection direction to pull the points out of the volume and a better computation of the response. Furthermore, we are able to modify the length of the sticks, whereas the voxels are rigid cubes, allowing us to have a more flexible proximity detection and response treatment.

Improved Collision Detection and Response Techniques for Cloth Animation [MKE02] (2002)

This paper focuses on *collision detection*, *self-collision detection* and *collision response* for deformable bodies. It is based on different previous works mainly on *hierarchical bounding volumes* [VT94], *collision prediction* (proximities) through *k-DOPs bounding volumes* [KHM⁺98] and *heuristics* [VT94]. It presents a system by merging and improving these techniques.

A *k-DOP* (discrete oriented polytype) is a convex polyhedron defined by *k* half spaces. It allows a more precise enclosure of volumes than the traditional rectangular bounding-boxes. To perform proximity detection or collision prediction, the *k-DOP* gets inflated in the direction of its movement. A hierarchy is then generated within the bounding volumes by recursively splitting it into *k-DOPs*, thus generating a tree data-structure until one single polygon remains per leaf.

Collisions can then be detected by recursively traversing the inflated hierarchies of two colliding bodies. When two or more candidate faces are returned, they are passed to the collision response. Self-collision detection proceeds in a similar way but uses also adjacency and curvature properties of the body. The self-collision turns out to be very efficient.

The idea is to preserve large time steps and stability by detecting collisions before they occur. Two different types of collisions are considered: firstly, between two deformable faces, and secondly, between a deformable face and a rigid environment-face. In the second case, we must consider a transfer of momentum. The further computations for the response are comparable with what we have presented until now and we will not discuss them here any longer.

The results presented at the end of this paper show great efficiency. However, we cannot consider the different approaches related here as they

are not easily compatible with the architecture we are working with.

Real-time Animation of Dressed Virtual Humans [CMT02] (2002)

This document is the first claiming effective *real-time* cloth-simulation. The work presented describes rather an overall optimization than bringing new more effective integration methods or collision algorithms.

The idea here is to decrease the computations necessary for a fully dressed human, by classifying clothes with regard to their usual behavior when worn. The considered cloth-types are the following:

Stretch clothes : tight clothes that remain at a constant distance to the body and for which no or few simulation is needed. The mesh of the cloth is attached to the body.

Loose clothes : like sleeves or trousers. The tissue cannot be attached to the body but one defines *latitudes* along the arm or the leg, cloth-points can be assigned to remain on a same plane defined by these latitude. This means it is not possible any more to roll up trousers or sleeves.

Floating clothes : the common considered cloth for simulation. The cloth can move freely (dress, cape, etc.) still with control points attached to the body so to fix the tissue as a belt would do.

Interactive Animation of Cloth including Self Collision Detection [FGL03] (2003)

This paper is very recent and presents an interactive system for animating cloth. It first describes a integration method allowing large time steps while keeping high stiffness of the cloth. The integration method is based on discretizing the cloth into particles and assigning constraints to them, mainly a *minimum distance* constraint preventing particles to come too close to each other.

The minimum distance constraint will be reused in the self-collision detection. Instead of correcting collisions once they occur, the constraint will prevent them. However, a proximity test is still necessary between all particles and a hierarchical structure is implemented.

Although this approach allows very rapid computations, it still is not robust enough and is not able to treat a collision whenever it might occur. Also, even if computations are drastically reduced, it is indicated that over 1600 particles, the simulation slows down and the self-collision process must be bypassed.

For the experimental part, the cloth is tested onto a static rigid volume and there are no considerations regarding collision with deformable bodies.

Computational Aspects of Dynamic Surfaces [Bri03] (2003)

This last document is the Dissertation for the Degree of Philosophy from Robert E. Bridson. It is the most recent document we have, dealing with cloth simulation. Whole chapters are dedicated to integration methods, collision detection and response, but the presented approaches aim to get the most realistic possible simulations using often different techniques to achieve the necessary precision. Thus, no real-time aspect is considered even if efficiency is clearly one of the most important parameters.

2.3 Summary

As we have seen, investigation on cloth modeling began in 1992, with a paper from Baraff and Witkin [BW92] where they deal with deformable flexible bodies and present approaches to detect collisions and perform efficient response.

In 1995, Volino and Thalmann [VCT95] present an approach for cloth integration based a mass-spring system. They consider collisions and already distinguish proximities from interferences.

Hecker then describes, in a game programming journal, an efficient way to handle collision response [Hec97]. It is on his approach that the collision response for our system is based.

One of the first, and nowadays one of the most widely used, self-collision algorithm is proposed by Volino and Thalmann in 1998 [VM95]. The approach studies the curvature of the cloth and defines regions where no self-collision can occur; collisions can then be detected by searching intersecting regions.

The same year, Baraff and Witkin [BW98] show how simulations can be accelerated by presenting methods allowing larger time steps in the cloth integration without losing on stability. Moreover, they consider self-collisions and propose to separate colliding regions of a cloth by inserting small springs between them.

In the year 2000, Meyer, Bebune, Desbrun and Barr [DMB] propose an *implicit* cloth integration method, which allows to use even larger time steps. For handling collisions, they make a voxel-discretization of the space for fast proximities detections.

Recently, in 2002, Mezger, Kimmerle and Etmuss [MKE02] implemented many of the different existing techniques in a single system. Although they did not achieve real-time simulations, they clearly defined collision handling as being the bottleneck for interactive simulations.

The same year, Cordier and Thalmann [CMT02] showed how it is possible to optimize systems simulating a large number of clothes at the same time, by making a distinction among the different clothes. Clothes are separated into three categories, the stretched ones, that require few computa-

tions (socks, hat, etc.), the loose ones, which can only slide along the body (trousers, T-shirt, etc.), and the floating ones, which can move freely (dress).

Finally, this year, Fuhrmannm, Gross and Luckas [FGL03] achieved one of the first real-time simulations. Their integration method allows large time steps, but they pretend having used a rough collision handling algorithm that avoids collisions from happening with a relative efficiency. If collisions occur, the system cannot correct them.

Chapter 3

Collision Detection

The *collision detection* is the first phase of the collision process, the second being the *collision response*. In the first section of this chapter, we present the library on which our collision detection approach is based and in the following sections the necessary modifications we made to that library so to adapt it to our needs. We will not directly describe the final implementation we have made but rather we will describe the evolution of our development so to make clear where the different ideas came into consideration.

The *Collision Control* library offers a tool that allows efficient queries on whether a point penetrates a given volume by using an instance of an LDI (Section 3.1). To use this library in our cloth-simulation system, we will have to adapt it (Section 3.2) and furthermore use three instances of this LDI to get a greater stability (Section 3.3). A last modification will allow to optimize the generation of these three instances (Section 3.4). We will end by discussing the efficiency of the different modifications (Section 3.5).

3.1 Introduction to the Collision Control Library

The *Collision-Control* library was recently developed by Bruno Heidelberger at the CGL. It is a collision detection algorithm working with *LDIs* (Layered Depth Images) which, in real-time, discretizes closed deformable bodies into parallel sticks having all the same section.

The LDI generation uses an OpenGL feature that allows extracting the pixels-data from the graphic card once a scene has been rasterized on it. This data is stored in the *frame buffer* which holds more information than just the colors of the pixels to be drawn onto the screen: this memory is multi-layered and the whole information of every object to be rendered is kept, mainly the *depth information*.

The idea is to render any volume off-screen at a low resolution and to

retrieve the information afterwards. Every LDI stick is thus created from the information of a retrieved pixel. The depth information indicates when an LDI stick enters or leaves the body: counting from zero, even depths are entry-points, odd depths are leaving-points.

OpenGL provides three basic commands for manipulating image data:

- `glReadPixels()`
- `glWritePixels()`
- `glCopyPixels()`

We will focus on the first one as it is the only one used by the *Collision-Control* library. This command reads a rectangular array of pixels from the frame buffer and stores the data in the processor memory. The command for reading pixels from the frame buffer is the following:

```
- void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);
```

x, *y*, *width* and *height* define the portion to read. *format* indicates the kind of pixel-data elements that are to be read, *type* indicates the data type for each element and *pixels* is a pointer to the array that will hold the values.

In our case, the *format* will be `GL_DEPTH_STENCIL_NV` and the *type* `GL_UNSIGNED_INT_24_8_NV`. These constants are provided by recent versions of the OpenGL related library *glxt*. The data for each pixel will be stored over 32 bits. The 24 first bits give the depth of the *entry-* or *leaving-points* of the LDIs. The 8 last bits give the number of depth layers for that pixel.

3.2 Growing the LDI

The primary idea is to hold a cloth-point whenever it penetrates the LDI. With a fine resolution the LDI irregularities should not be noticeable.

In its original state, the generation algorithm gives an LDI which does not include entirely the volume as illustrated in Fig. 3.1: the sticks begin and end when the center of their square section intersects the volume. This has the drawback that even if some cloth-points remain outside the LDI, they might already have penetrated the volume and this one is visible through the cloth on several regions. The first modification consists in inflating the LDI so that we can keep the cloth at a certain distance from the body to avoid this “see-through” artifact.

Due to the geometrical properties of the unit of the sticks, the inflation should be performed in two steps. First we would dilate the LDI by adding

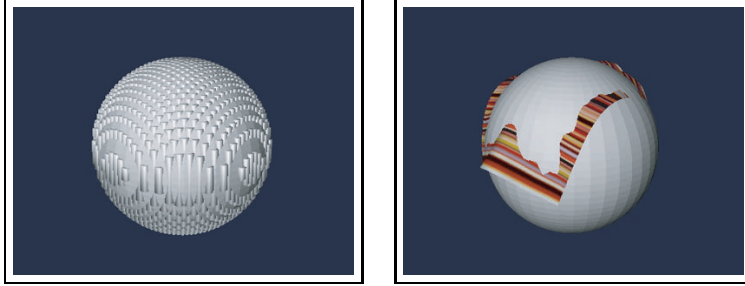


Figure 3.1: The original LDI discretization on a sphere

a layer of sticks around the original LDI (Section 3.2.1) then stretch all sticks in their length so that the LDI would completely enclose the volume (Sections 3.2.2 and 3.2.3).

3.2.1 Dilation

Fig. 3.2-b illustrates the *dilation* process on the original LDI. We begin by increasing the size of the LDI array by two in both directions so that we will be able to add a layer of sticks all around the existing LDI. We then go through the entire LDI array and for each *pixel* we consider its four neighbors, on its left, on its right the upper as the lower one. On that pixel, the stick is modified or generated so that along its whole depth, if there exists a portion of stick in its neighbors or in that pixel, then the updated stick will hold that portion, i.e., every stick information is reported to its neighbors and the sticks are updated according to the \cup operation. Algorithm 3.2.1 shows how to update all sticks.

Algorithm 3.2.1: DILATESTICK()

```

nbSticks  $\leftarrow$  0
currentDepth  $\leftarrow$  0
neighbor  $\leftarrow$  GETNEIGHBORWITHCLOSESTBOUNDARY(currentDepth)
currentDepth  $\leftarrow$  GETNEXTBOUNDARY(neighbor)
if (ISENTRYPPOINT(neighbor, currentDepth))
  then
    { nbSticks  $\leftarrow$  nbSticks + 1
      if (nbSticks = 1)
        then ADDENTRYPPOINTTOCURRENTSTICK(currentDepth)
      else
        { nbSticks  $\leftarrow$  nbSticks - 1
          if (nbSticks = 0)
            then ADDENTRYPPOINTTOCURRENTSTICK(currentDepth)
        }
    }

```

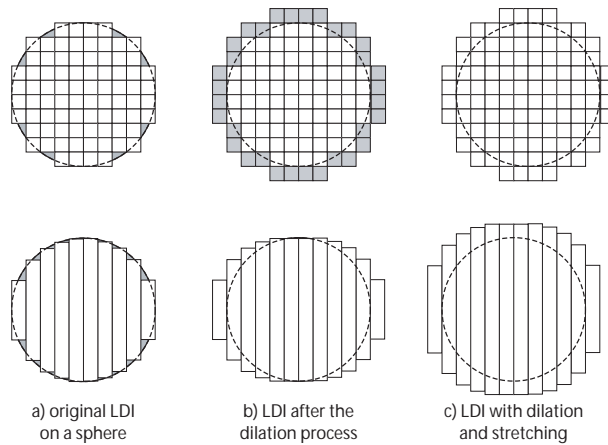


Figure 3.2: The early inflation stages on a sphere. The first row represents a section-view of the LDI sticks and the second a side-view.

3.2.2 Stretching

The step following dilation is the *stretching*. We want to extend all entry- and leaving-points of every stick with the length of an LDI pixel size. This is illustrated in Fig. 3.2-c. Again we process all LDI sticks and when we encounter an entry- or a leaving-point, we respectively pull or push it. The procedure is quite trivial and we do not expose here a pseudo-code to illustrate it.

Nevertheless, while stretching the sticks, entry-points may overlap preceding leaving-points. This is handled in a sub-step of the stretching: the *merging*.

3.2.3 Merging

As explained before, *merging* occurs when an entry-point is pulled over a leaving-point. These two points can thus be canceled and we get a continuous LDI stick. For example, when we lower the resolution of the LDI, the stretching pushes the entry and leaving points further and thus merging is more likely to occur. This is illustrated in Fig. 3.3.

3.3 Three-directional LDIs

Up to that point, our work went along to the *response phase* of the collision process. Regarding detection, we have a fast and deterministic tool, but detecting LDI penetration of cloth-points is not sufficient. A simple method would be to stop cloth-points as soon as they penetrate the LDI. The cloth would thus lie on the LDI rather than on the volume and we have no way

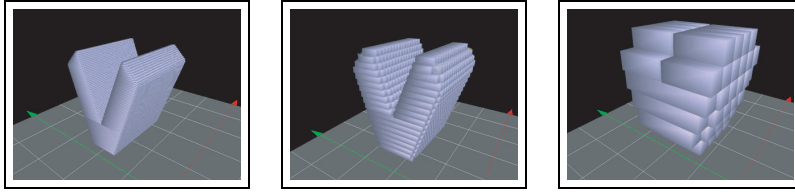


Figure 3.3: At low LDI resolution merging occurs more often

to deal with deformable volumes. So we implemented a *triangle-to-LDI association* (see chapter on Collision Response). Every entry- and leaving-point of an LDI stick gets associated with a triangle onto which the cloth-point will be projected as soon as it penetrates the LDI. This is necessary for two main reasons; we now know where to project the point and how to influence its velocity as we store the movement of all triangles.

However, experiments show that the cloth got very unstable unless the LDI is oriented in the direction of the gravity, in which case, the cloth trembles when it touches “steep” faces of the volume. This behavior is due to the approximation of the triangle-to-LDI association which is very awkward when the LDI sticks are getting “too parallel” to the faces of the volume. Fig. 3.4 illustrates such a case. When a cloth-point penetrates the LDI in the middle of a stick, it can be projected to a triangle that is far from the one it actually came through. Not only the projection will generate an unordinary displacement of the point, which will finally fall back to where it penetrated, but also it generates a wave within the cloth which leads to a chaotical behavior and the cloth integration will fail.

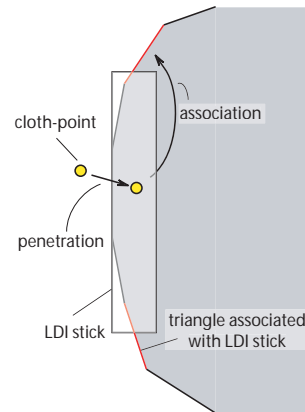


Figure 3.4: Problematic *triangle-to-LDI association*

To avoid this artifact, we generate a three-directional LDI. This is, we generate 3 LDIs along the X, the Y and the Z axis. A cloth-point can now penetrate at most 3 LDI sticks at the same time and we must associate it with the appropriate triangle: the algorithm takes the three possible triangles and makes the association with the closest one.

This approach leads to a much better stability of the cloth but gets greedy on time. That drawback could be somehow eliminated in that we removed the dilation phase in the inflation. Indeed, the stretching now covers all three directions and the dilation becomes useless.

Our collision response algorithm pulls the cloth-points outside the volume up to a certain distance (*aura*) above the triangles. This is necessary

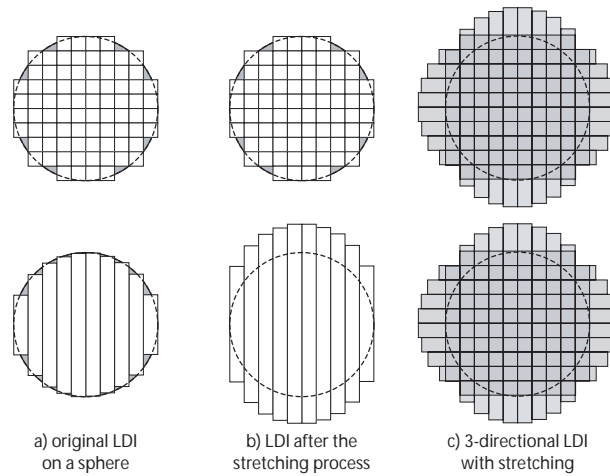


Figure 3.5: The actual inflation stages on a sphere. The first row represents a section-view of the LDI sticks and the second a side-view.

to anticipate the deformation of the volume. The distance that keeps the cloth separated from the volume is a parameter in our implementation and can be adjusted during run-time. We must be careful that the aura remains inside the inflated LDI in spite of what a colliding point would be projected outside the LDI and fall back all the time generating another tremor of the tissue. We adapted the stretching to that constraint so that the lengthening of the sticks depends on the *aura* rather than on the section of the sticks (which is also not unique unless the bounding-box is not a cube).

The 3-directional LDI inflation process is modeled in Fig. 3.5, and the final aspect of the 3-directional LDI, along with the cloth projected onto the volume, is shown in Fig. 3.6.

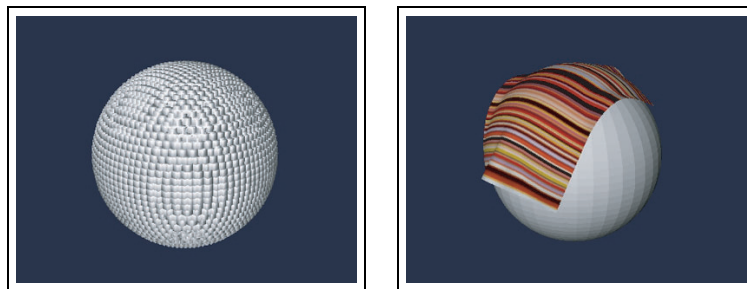


Figure 3.6: The 3-directional LDI discretization on a sphere along with the cloth projected onto it

3.4 Optimizing the 3-directional LDI

Up to that point, we developed a reliable and stable collision detection algorithm. However, generating the 3-directional LDI gives too much information and we look for a way to reduce its generation time.

We know that the projection of a cloth-point will be more reliable when the triangle gets orthogonal to the LDI-stick. Also, the closer the triangle to the penetration location, the more stable the simulation. Now, we suppose the cloth-point penetrates the volume and falls into three LDI-sticks, each of different direction. One of these is actually more orthogonal to the penetrated triangle and the closest triangle association is likely to be detected in that stick.

We recall that during the LDI generation, we insert the depth of every rasterized triangle in the pixel depths sequence. With three LDIs, triangles can be rasterized up to three times whereas one time would be sufficient. So we must choose in what direction a triangle is rasterized, and this will be in the direction with which the normal of the triangle has the greatest component.

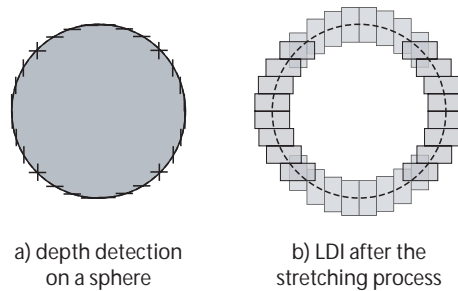


Figure 3.7: The inflation stages on a sphere after the 3-directional LDI optimization. On the left, depths are detected where the normals of the triangles have their greatest component in the direction on the LDI. On the right, short LDI sticks are produced.

With such an implementation, sticks can not be considered any more. Indeed, in a given sequence, it is no longer ensured that two consecutive depth correspond to an entry- and a leaving-point as the triangles will not have the same slope.

However, we still want to use the collision response as it was implemented and thus, we still need to work with sticks. The idea is to generate small sticks from every depth retrieved from a sequence. So we pull back every depth retrieved from a sequence and add a new depth after each of them. This step could be easily implemented through our stretching process. When generating the LDI, every time a depth gets added to a sequence we add it a second time. Although the depth will be the same, their index in the

sequence will differ. Original depths will have even indices and added depths odds. The stretching will pull back every even depth and push the evens as it did for entry- and leaving-points. This will produce small LDI-sticks with a length of two stretch distance centered on the triangle.

As a result, the volume will not be filled with the LDI anymore (see Fig. 3.7-b), but if the sticks are long enough, no cloth-point will be able to enter the volume deeply enough .

3.5 Discussion

3.5.1 Three LDIs versus 3-directional LDI

Regarding the 3-directional LDI, we have the choice to implement it in two different ways: either we generate three LDI instances, each in a different direction, or we can modify the Collision Control library so that it handles the three directions transparently, returning only the index of the triangle we have to deal with.

In our implementation, we went used the first approach. By the time we made that choice, we saw no great difference between the two approaches, although the second was much more fastidious to implement, as it would have required wide modifications of the original library.

Also, we could choose between rendering the volume using the graphic card or a software implementation. Referring to the technical report on the Collision Control library, the hardware begins to take advantage over the software for meshes with a lot of triangles; much more than the volumes we worked with. Not only the software alternative was thus more efficient for our simulation, but the adaptations were easier to implement.

By using the software implementation, we could thus store the indices of the triangles while performing the rendering, but we also had a direct access to where the rendering process adds the depths of the triangles in the pixel sequences, where it was made possible to add the depth twice for the optimization of the 3-directional LDI. Also, working with 3 instances allowed to deal with a lot of information outside of the Collision Control library.

However, all steps from the *inflation process* were implemented for both options.

Choosing the software approach prevents from reducing considerably the generation time with the 3-directional LDI. Indeed, the three instances work totally independent, and when the LDI is generated, the process will go three times through all the triangles, regardless of whether some triangles should be rendered or not, the orientation of the triangles being verified right before their depth is added to the sequence. Beside of this, aside from the generation time, the accuracy and the efficiency of the detection and the simulation stability would be exactly the same with both implementations.

3.5.2 Choosing the Direction for Rasterizing

When we presented the optimization of the 3-directional LDI, the triangles of the mesh can only be rendered in a single direction. For example, when a sphere is rendered, we could define six regions of neighboring triangles rendered in a same direction (see Fig. 3.8-a).

As described earlier, our implementation computes the orientations of the triangles in order to determine which one will be rendered. At first thought, we should render triangles whose normal n forms an angle smaller than 45 degrees with the LDI direction (i.e. $|n \cdot dir_{stick}| > 0.5$). In this case, not all the triangles would be rendered but only those in the regions delimited in Fig. 3.8-b. For the regions to cover all the sphere, all triangles with a normal forming an angle below about 55 degrees should be considered. Nevertheless, as we can see in Fig. 3.8-c, some triangles would be rendered more than once. To obtain a good distribution (Fig. 3.8-c), we really must consider the three components of the normals, and render a triangle only in the direction of its greatest component.

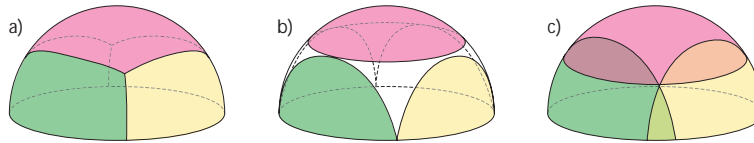


Figure 3.8: The colors identify the regions in which the triangles are detected to be rasterized in the same direction. a - illustrates the correct repartition, b - the regions that would be rasterized when considering a 45 degree criterion, c - with a higher degree criterion, the entire surface is rendered, but some regions are rasterized twice.

Chapter 4

Collision Response

In the previous chapter, we presented the *collision detection* which allows to detect a collision as soon as a cloth-point penetrates the volume. This chapter presents an efficient way to handle these collisions by correcting the position and the velocity of these colliding points.

The *collision detection* was developed in parallel with the *collision response*. To allow the reader to better understand how we implemented this second phase of the *collision process*, we will show how the implementation evolved according to the adaptation of the LDIs. That is, we will first show how it worked along with the *stretched 3-directional LDI* and secondly how we could adapt it to a *two-side detection*.

Collision detection and *collision response* can be treated separately, but we need to agree on a coherent transfer of information from the first one to the second. In this chapter we focus on *collision response* and describe our approach for handling cloth-points that have been detected as being inside a volume, i.e. that have already collided with it.

The approach here is slightly different from others as it considers deformable models, that is, we cannot simply consider the center of mass of the volume and its rotation, but we need the information on the velocity, the position and the normal for every triangle of the volume. However, we suppose the volume to always be much heavier than the cloth and therefore we will make abstraction of the masses in our computations.

In this chapter, we will present the *triangle-to-LDI association* (Section 4.1) which allows to detect through which triangle a cloth-point penetrates the volume. Then we explain, how we detect which cloth-points are candidate to collision response (Section 4.2) and how we perform the computations, regarding the projection (Section 4.3) and the velocity change (Section 4.4). In a further step, we also show, how the optimization of the 3-directional LDI presented in the previous chapter allows to handle collisions on both sides of any open mesh (Section 4.5). Finally, we give

the algorithm used for collision response (Section 4.6) and will discuss our implementation (Section 4.7).

4.1 Triangle-to-LDI Association

Our collision detection phase uses LDIs. The volume is thus discretized into parallel sticks having all the same square section. We thus have a fast implementation for detecting collision but we still lack some information. We need to know through which triangle the cloth-point has penetrated so that we can pull it out of the volume correctly and precisely compute its new velocity.

We cannot rely on any geometrical computation relative to the center of mass of the volume or any of its characteristics as we consider deformable volumes and it would be too difficult to describe a displacement function for the triangles.

So, our approach will make an estimation of the triangle the cloth-point has gone through by associating a unique triangle with every entry- or leaving-point of the LDI sticks. Thus with a correct LDI resolution and a volume with a consequent number of faces, we can admit that the different triangles that can be associated to an entry-/leaving-point would lead to equivalent computations (i.e. their velocity and normal are similar). Fig. 4.1 illustrates the “triangle-to-LDI” association.

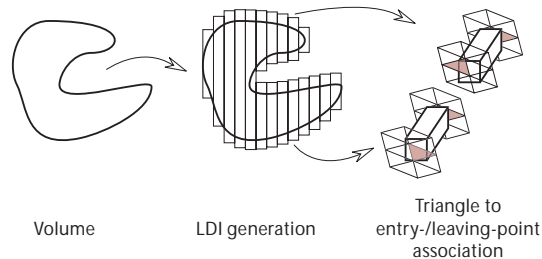


Figure 4.1: Triangles-to-LDI association

4.2 When is a Cloth-point Inside the Volume?

Now that we can retrieve the triangles associated with the collision of the cloth with the volume, we can begin with the *collision response* process. The way we modified the LDI generation ensures that the volume is entirely included in it. Therefore, there are portions of LDI sticks that are outside the volume and querying the LDI whether a cloth-point lies inside the volume could return *true* even if the cloth-point has not actually penetrated the

volume. Thus, this query is not sufficient and we need to know whether the cloth-point lies *above* or *below* the triangle the query returned. However, although the LDI is not a reliable structure for queries on point-to-volume belonging, it is still necessary because it indicates when a point has come close enough to the volume to be taken into consideration and particularly, it associates the point with a triangle.

As we already mentioned when presenting the 3-directional-LDI (Section 3.3), when a point penetrates the volume, we want to project it to a certain distance (aura) over the triangle. In fact, we consider a point to have penetrated the volume as soon as it gets below this aura-distance. Referring to Fig. 4.2, there are four possible cases to consider for a cloth-point. P_1 is outside the LDI, the query will return *false* and we will skip to the next cloth-point. P_2 , even being inside the LDI, remains outside the volume and over the aura-distance and no modification apply to its behavior¹. Finally P_3 is below the aura-distance and P_4 is inside the volume; both points are treated the same way: they must be pulled outside and their velocity must be modified so as to behave as if it rebounded on the surface of the volume.

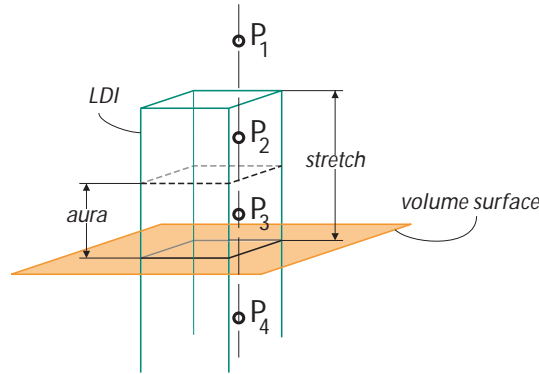


Figure 4.2: The different positions of a point-cloth

4.3 Projecting the Cloth-point

Once we know a cloth-point is inside the volume (or below the aura-distance), we must do two things: first we must displace the point to the surface of the volume and secondly compute its new velocity. Note that in the following, we will not consider the aura in our computations, so to keep our

¹Note that the stretching could not be shorten to the aura-distance. For stability reasons, we keep it longer: the plane spread by the triangle is not necessarily parallel to the extremity of the sticks, and therefore, the plane on which we project the points neither. We want to be sure that the points, once projected, are still inside the LDI so that they are maintained at the aura-distance at every time step.

explanations clean.

We begin by describing here the first of these steps. The approach is to project the point onto the triangle. We want to compute the displacement vector for this projection. Fig. 4.3 will help us understand how we compute the projection vector d_{proj} . Knowing the vertices of the triangle and the location of the cloth-point, we can deduce the vectors a , b and c . Then we project c onto a and the difference between this projected vector and c onto n ,

$$\begin{aligned} \text{proj}_{\rightarrow a} c &= \frac{c \cdot a}{\|a\|} \frac{a}{\|a\|} \\ d_1 &= \text{proj}_{\rightarrow a} c - c \\ d_{proj} &= (d_1 \cdot n) n, (\|n\| = 1) \end{aligned}$$

This will give us d_{proj} , the displacement vector for P to be projected onto the plan spread by the triangle.

If we consider the aura-distance as ϵ value, we want the point to be slightly pulled outside the volume so to avoid the artifact of seeing the volume through the cloth. To pull the point outside, we add a portion on n with a length of ϵ to that projecting vector. Thus,

$$d'_{proj} = d_{proj} + \epsilon n$$

We can now determine whether the point lies above or under the triangle by computing (Fig. 4.3).

$$c \cdot n \begin{cases} > 0 \Rightarrow \text{the point is above the triangle} \\ = 0 \Rightarrow \text{the point is in contact with the triangle} \\ < 0 \Rightarrow \text{the point is below the triangle} \end{cases}$$

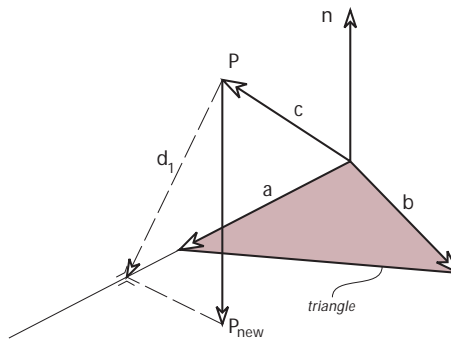


Figure 4.3: Projecting a cloth-point onto a triangle

4.4 Affecting the Velocity

Hitherto, we have described how to handle the position of a cloth-point but we still have to write on how its velocity changes after it has been in contact with the volume. For this, we must consider the relative velocity v_{rel} between the cloth and the volume. This will allow to know whether the velocity of the point is directed *towards* or *off* the triangle it is associated with. Knowing the velocity of the cloth-point v_p and the velocity of the triangle $v_{triangle}$ we have

$$v_{rel} = v_p - v_{triangle}$$

v_p is already known from the *integration steps* and is associated with every point. However, from the triangle we only know its current position. To make an approximation of its velocity, we have to store the preceding position of all triangles aside from its current ones, and compute the velocity by using the displacement of its three vertices. Suppose collision response is performed n_{update} times more often than the shape of the volume is updated and that we know the time elapsed $t_{elapsed}$ between two collision response computations. We can compute, based on Fig. 4.4, the average displacement d_{avg} of the triangle².

$$d_{avg} = \frac{\sum_{i=1}^3 dp_i}{3}$$

and we get the velocity

$$v_{triangle} = \frac{d_{avg}}{n_{update} \cdot t_{elapsed}}$$

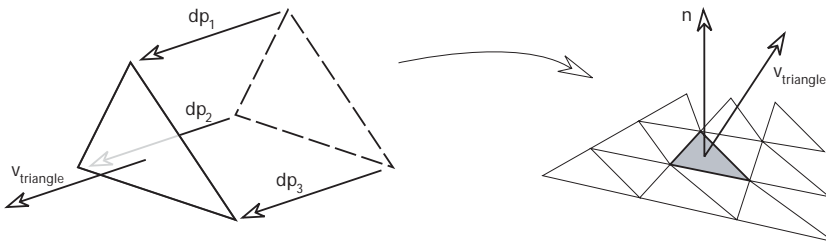


Figure 4.4: Approximating the velocity of a triangle

²Note that with our approximation of the velocity of the triangle, we loose the angular velocity information. However, considering a body that undergoes a rotation that is not detected on the considered triangle, the neighbors of that triangle will rotate around it. The rotation will thus influence the cloth on the surrounding of the cloth-point that collided of that triangle. The cloth-point will thus get the rotating component when it will be trailed by its neighbors during the integration step.

Now that we know the relative velocity, we can take the following considerations:

$$v_{rel} \cdot n \begin{cases} > 0 \Rightarrow \text{the point leaves the triangle} \\ = 0 \Rightarrow \text{the point slides along the triangle} \\ < 0 \Rightarrow \text{the point hits the triangle} \end{cases}$$

It is only in the third case that we will affect the velocity of the point. For this, we will compute the new relative velocity v'_{rel} by reflecting it onto the triangle. Based on Fig. 4.5 we make the following computations:

$$\begin{aligned} v_{proj} &= (v_{rel} \cdot n)n \\ v_{refl} &= v_{rel} + 2 \cdot v_{proj} \end{aligned}$$

where v_{refl} is the geometrically reflected velocity, but we still need to consider a damping factor ρ_{damp} and a friction factor $\rho_{friction}$ where

$$\rho_{damp}, \rho_{friction} \in [0, 1]$$

The damping factor will crush v_{refl} along n and the friction factor orthogonally to n . The components of v_{refl} along and orthogonal to n are given by:

$$\begin{aligned} v_{refl\parallel} &= (v_{refl} \cdot n)n \\ v_{refl\perp} &= v_{refl} - v_{refl\parallel} \end{aligned}$$

to which we multiply the damping and friction factors to obtain the final reflected velocity of the cloth-point,

$$\begin{aligned} v'_{refl\parallel} &= \rho_{damp} \cdot v_{refl\parallel} \\ v'_{refl\perp} &= \rho_{friction} \cdot v_{refl\perp} \end{aligned}$$

and

$$v'_{refl} = v'_{refl\parallel} + v'_{refl\perp}$$

Finally we attribute to the cloth point its new velocity v'_p :

$$v'_p = v'_{refl} + v_{triangle}$$

4.5 Two-side Detection

The two previous sections explained, how we modified the position and the velocity of any cloth-point colliding with the volume. These two principles

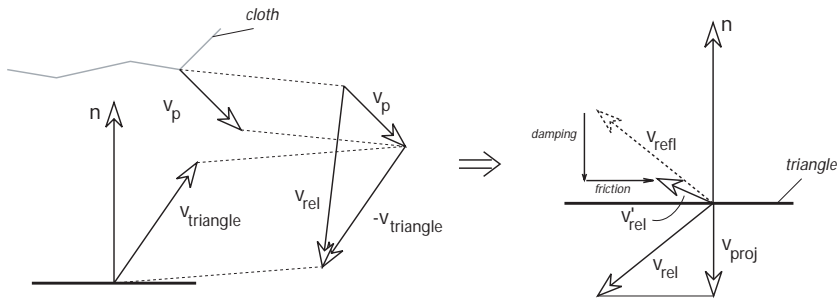


Figure 4.5: Computing the new relative velocity

constitute the *collision response process* and require only to know the direction of the projection and the normal of the triangle to compute the reflected velocity.

From these considerations we understand that the collision process to be performed on the two sides of a mesh only depends on the implementation of the collision detection phase. Therefore, when we began working with the LDIs, we focused on single-side collisions as the LDI required meshes to be closed, so as to always have pairs of entry- and leaving-points.

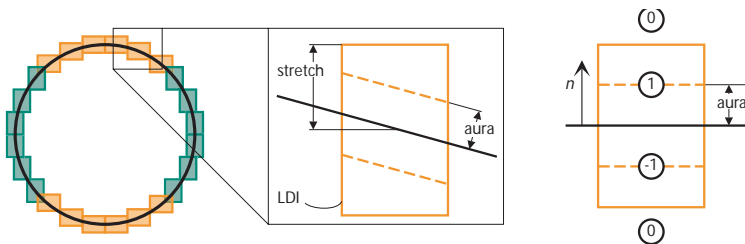


Figure 4.6: The symmetry in the new LDI-sticks, leads to a way to detect collisions on both sides of the mesh of the volume.

This consideration is no longer valid once we use the optimization of the 3-directional LDI (Section 3.4). We have seen that the LDI-sticks are formed around every rasterized triangle and are centered in their length to the triangle. This symmetry in the new LDI-sticks leads to a way of detecting collisions on both sides of the mesh.

We begin with an approach illustrated in Fig. 4.6. We assign a *state* to every cloth-point. As long as the cloth-point is outside any LDI-stick, the state remains 0. As soon as it is detected to be inside the LDI we check whether the point is above or below the triangle. We assign the values 1 and -1 , respectively. The efficiency of this approach resides in the fact that when the state of a point is non-zero, we need no further computations to

know, to what side the cloth-point belongs. Moreover, we can multiply the normal of the considered triangle by the state of the point and the projection and velocities are computed correctly automatically.

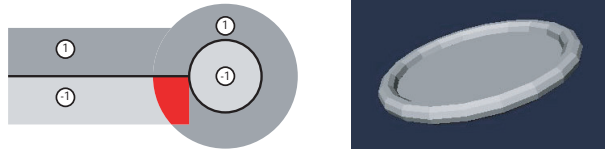


Figure 4.7: The regions where the state of the cloth-points are positive or negative. The red region describes, where the cloth-points could get two different states and thus must be treated differently.

We tested our method with open meshes like an open hemi-sphere (Fig. 4.8) and a plate (Fig. 4.7). It worked fine, until we reversed the plate. The cloth fell perfectly in the center region but became very unstable at the borders. The problem lies in the geometry of the plate. Fig. 4.7 shows the profile of the plate which is composed of a plane and a torus (represented as a circle). Also, the regions where the state of the cloth-points are positive or negative are shown. That is, for the plane, 1 is above and -1 is below. For the circle, 1 is outside and -1 inside. This defines a region where the cloth-points get randomly two different states and must thus be treated differently is they get associated with a triangle from the plane or with a triangle from the torus.

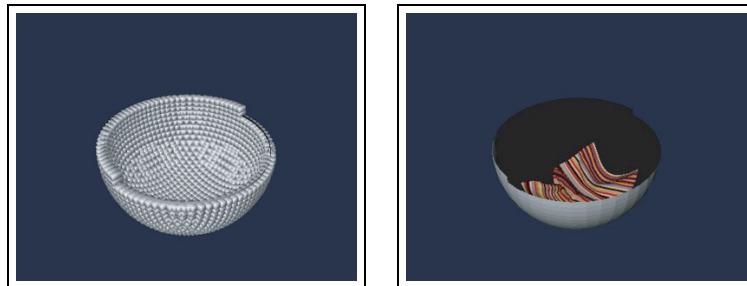


Figure 4.8: The two-side detection LDI discretization on a sphere

In our algorithm, we must thus reconsider the cloth-points when their state is non-zero. So when we get a cloth-point and the associated triangle, we check whether the point was above or below that triangle in the last time step and determine the side it belongs to. The states are no longer necessary and we simply twist the normal of the triangle if the point was below.

With that implementation, it is now possible to detect collisions on both sides of any open meshes. The LDI representation after the 3-directional

optimization and the detection of the inner side of the mesh are illustrated in Fig. 4.8.

4.6 The Algorithm

Now that we have gone through all the stages of computing the collision response, we present in pseudo-code the algorithm we follow (see Algorithm 4.6.1).

Algorithm 4.6.1: COMPUTEREFLECTION(*cloth*, *ldi*, ϵ , $\rho_{damping}$, $\rho_{friction}$)

```

pointset ← cloth.GETPOINTS()
for each P ∈ pointset
  if ldi.ISINSIDE(P)
    then
      { Triangle ← ldi.GETASSOCIATEDTRIANGLE(P)
        n ← GETNORMAL(Triangle)
        projP ← GETPROJECTIONVECTOR(Triangle, P)
        projO ← GETOLDPROJECTIONVECTOR(Triangle, P)
        if DOTPRODUCT(n, projOldP) > 0
          then
            { comment: the point was previously below the triangle
              n ← -1 · n
            }
        if DOTPRODUCT(n, projP) > -ε
          then
            { comment: the point is below the triangle (under aura)
              P ← P + projP + ε · n
              vP ← P.GETVELOCITY()
              vTriangle ← GETVELOCITY(Triangle)
              vrel = vP - vTriangle
              if DOTPRODUCT(n, vrel) < 0
                then
                  { comment: the point hits the triangle
                    vrefl ← COMPUTEREFLECTEDVECTOR(n, vrel)
                    vrefl|| ← GETPARALLELCOMPONENT(vrefl, n)
                    vrefl⊥ ← vrefl - vrefl||
                    vrefl|| ← ρdamping · vrefl||
                    vrefl⊥ ← ρfriction · vrefl⊥
                    v'refl ← vrefl|| + vrefl⊥
                    P.SETVELOCITY(vTriangle + v'refl)
                  }
                }
            }
          }
        }
      }
    }
  }

```

We consider all cloth-points one after another. We query whether it belongs to the LDI and to which triangle it is associated. Then, using the information from the previous time step, we determine from which side of the triangle the cloth-point comes from; if it comes from below, the normal of the triangle is inverted.

The further step checks whether the point has come under the aura-

distance. If not, we proceed to the next cloth-point. Otherwise, the point is projected.

We then verify the velocity. Only if the point comes towards the triangle, its velocity will be modified.

4.7 Discussion

We have implemented the second phase of the collision process. The algorithm involves different parameters and some of them have great influence on the stability. Mainly, the aura-distance and the LDI resolution.

4.7.1 Aura

The experiments show great stability, even with relatively large time steps. The stability is very satisfying both when the cloth lies on and when it is colliding with deformable volumes. However, when the volume gets vigorous deformations, the aura-distance has to be increased to avoid seeing the volume through the cloth. When the aura-distance is increased, so is the stretching; we must ensure that when a cloth-point is projected above the triangle it remains within the LDI. When the aura is getting high, the point-to-triangle association gets more approximate as the surface delimited by the aura becomes less similar to the surface of the volume. Also, from one time point to the other, in such a case, a cloth-point gets more chance to be associated with different triangles that are relatively distant in the geometry of the mesh, and the cloth begins to loose stability.

4.7.2 LDI-resolution

Another important factor regarding stability, is the resolution of the LDI. If the section of the sticks is relatively bigger than the area of the triangles of the volume's mesh, only a few triangles will be rendered in the LDI generation process. While the volume is deforming, the triangles associated with a stick might be considerably different from one time point to the other. This will be sensed when we observe the behavior of the cloth on a volume deforming slowly. Also, the cloth will fit less smooth onto the volume, as large regions of it will be associated to the same triangle (i.e. the same plane).

On the other hand, raising the resolution will drastically decrease the frame rate of the simulation. The user must thus carefully choose an adapted resolution to for any given simulation. Note that when the section of the LDI-sticks gets relatively small regarding the triangles of the mesh, higher resolution brings no further improvement to the stability of the simulations.

Fig. 4.9 illustrates, how better precision is obtained using a higher resolution for the LDI. Especially for concave surfaces, if the section of the

LDI-sticks is much larger than the triangles, two neighboring cloth-points might get torn apart generating great unregularities in the cloth's surface. Also, great forces are created between the two points, and the simulation system is more likely to break.

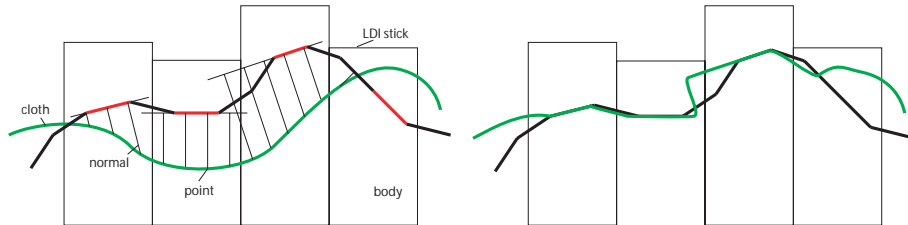


Figure 4.9: Projection of cloth points onto the surface of the volume

Chapter 5

Self-collision

In the previous chapters we discussed the *collision process* involving a cloth piece and deformable volumes. Another important aspect in cloth simulation, is *self-collision*. This problem cannot be solved with the approaches described above, for different reasons:

- with self-collision, positions and velocities of *every* colliding part of the mesh have to be affected;
- a cloth has no inner nor outer side;
- LDI-sticks are not well defined anymore.

However, we still want to use LDIs to solve this problem. Roughly, we want to find a way to use the depth information we retrieve from the LDI to detect self-collisions and deal with them. The “stick” consideration can no longer be taken into account, but rather, we will use the information from an LDI pixel as a sequence of depths and use their association with the triangles of the mesh of the cloth.

For handling self-collisions, we considered a lot of different ideas. We will show what information is at our disposal (Section 5.1) and then present independently the different elements we want to include in our algorithm: collision detection (Section 5.2), collision avoidance (Section 5.3) and the response (Section 5.4). Once these ideas are clarified we will show how they can work together (Section 5.5) and how we could integrate them in a pipeline (Section 5.6) and present a global algorithm. Finally, we will discuss the advantages and the drawbacks of our method (Section 5.7).

5.1 Considerations

From an LDI stick, we retrieve an ordered sequence of triangle indices:

$$sequence = [3, 7, 4, 1, 9, 2, 3, 1]$$

From these indices, we find the triangles associated and deduce their approximate z -component¹.

We must consider two cases:

- (A) triangles might have collided during the last integration step,
- (B) triangles might collide during the next integration step.

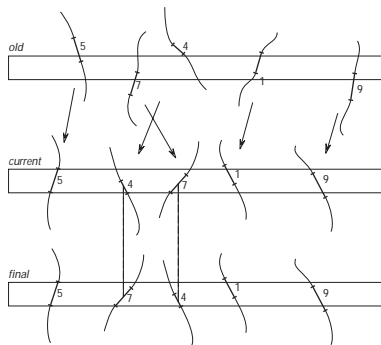


Figure 5.1: After a collision has been detected the depths of the triangles are swapped.

When considering (A), two triangles that have collided must be separated by modifying their actual position. Also, as a first response, their velocities should be affected. In case (B), triangles have not yet collided and we will thus not alter their position. However, we will change their velocity to avoid future collisions.

At first thought, these two cases could be two different approaches to implement separately as one would prevent the other one. Still, we will need both of them to have a more reliable *self-collision process*: our system for cloth simulation is based on the LDI. Currently, we have associated a unique triangle for every entry- or leaving-point of an LDI stick. Our *self-collision process* will only affect these triangles but not their neighbors which might lead to instability. However, by implementing *collision detection* and *collision avoidance* together, we will consider up to two times more triangles. Indeed, triangles associated with the sticks change after every integration step and therefore, a collision that cannot be *detected* in the current point in time, might have been *avoided* in the past.

Apart from these considerations, we will ensure in a further step, that the triangles are at least separated by a distance ϵ .

¹The approximate z -component is obtained by computing the average over the z -components of the three vertices of the triangles.

5.2 Self-collision Detection

We know the previous depth of all triangles and can therefore deduce their previous sequence. Permutations of the old sequence in the current one indicate which triangles have collided.

To recover from a collision, we reorder the old depths sequence while performing the same permutations over the current sequence to obtain the old indices sequence.

That means, the triangles will not return to their previous positions, but they take the current z -coordinate of the triangle it has been permuted with.

5.2.1 The Approach

We have the current sequence of possibly colliding triangles:

$$sequence_{curr} = [5, 4, 7, 1, 9, 8, 2, 3]$$

We compute the current depths related to this sequence and the elder one:

$$\begin{aligned} Zseq_{curr} &= [0.1, 0.2, 0.5, 1.2, 1.8, 2.1, 2.2, 2.8] \\ Zseq_{old} &= [0.1, 0.4, 0.3, 1.2, 1.8, 2.1, 2.2, 1.9] \end{aligned}$$

$Zseq_{old}$ is an unordered sequence of depths and we compute $sequence_{old}$ while reordering $Zseq_{old}$:

$$\begin{aligned} sequence_{old} &= sequence_{curr} \\ Zseq_{old} &= [0.1, \mathbf{0.4}, \mathbf{0.3}, 1.2, 1.8, 2.1, 2.2, 1.9] \\ sequence_{old} &= [5, \mathbf{4}, \mathbf{7}, 1, 9, 8, 2, 3] \\ Zseq_{old} &= [0.1, 0.3, 0.4, 1.2, 1.8, 2.1, \mathbf{2.2}, \mathbf{1.9}] \\ sequence_{old} &= [5, 7, 4, 1, 9, 8, \mathbf{2}, \mathbf{3}] \\ Zseq_{old} &= [0.1, 0.3, 0.4, 1.2, 1.8, \mathbf{2.1}, \mathbf{1.9}, 2.2] \\ sequence_{old} &= [5, 7, 4, 1, 9, \mathbf{8}, \mathbf{3}, 2] \\ Zseq_{old} &= [0.1, 0.3, 0.4, 1.2, 1.8, 1.9, 2.1, 2.2] \\ sequence_{old} &= [5, 7, 4, 1, 9, 3, 8, 2] \end{aligned}$$

Now we can make the comparison:

$$\begin{aligned} sequence_{old} &= [5, 7, 4, 1, 9, 3, 8, 2] \\ sequence_{curr} &= [5, 4, 7, 1, 9, 8, 2, 3] \end{aligned}$$

where the groups of permutations are numbered increasingly:

$$groups = [0, 1, 1, 2, 3, 4, 4, 4]$$

The triangles that have been permuted are $\{4 \leftrightarrow 7\}$, $\{3 \leftrightarrow 8 \leftrightarrow 2\}$. For each of them, we will take the current z -component of the triangle that had the same sequence position in the elder sequence. This last step is in fact the beginning of the *self-collision response* process which will be discussed in a next section.

Fig. 5.1 illustrates the behavior for the five first triangles of our sequence example.

5.2.2 Improvement

The permutation approach has presented successful collision corrections but the simulation becomes unstable once these collisions become too complex. Such difficult collisions come up when the cloth is falling with great velocity and a tissue sample collides deeply into another and even more when it crosses multiple layers of cloth. Also when two neighbor triangles are involved in the permutation process, great distortions occur leading the system to collapse.

Fig. 5.2 illustrates such a case. The far right triangle collides with the two neighbor triangles during the last integration step. The permutation process will hold the current depths sequence and redistribute the triangles according to the indices sequence from the *old* state. In this case, the neighbor triangles will get great intervals separating them and their common vertices should be split into two. As this is not possible, the triangles will get distorted and great strengths will be generated on their arrests. The overall energy will thus increase and a chaotical behavior will follow breaking the next integration step.

The improvement aims at conserving reasonable intervals between triangles and recover the depths of the triangles, of a same permutation group, at the time of the first collision.

We will start by considering a group of colliding triangles (by considering permutation groups) and find the time of the first collision between them. This step is not greedy in computation time as we only look for the time of the collisions between two adjacent triangles in the *old* indices sequence².

Once the time of the first collision has been detected, we can find the depth for all triangles at that time. These depths will be attributed to each

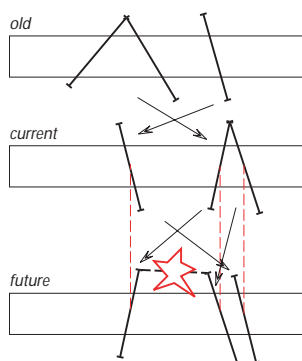


Figure 5.2: The permutation step can lead to simulation instability when the permutation involves neighbor triangles.

²Suppose a collision occurs between two non-adjacent triangles in the *old* indices sequence. Then one of them must already have collided with a triangles that was in between them in the sequence.

triangle for the *future* state. Fig. 5.3 illustrates this improvement.

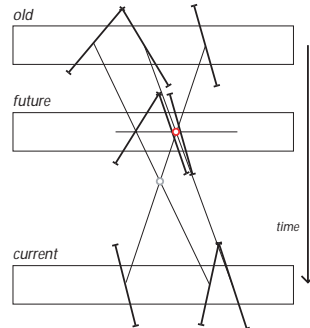


Figure 5.3: The improvement consists in retrieving the depth of all triangles within a permutation group at the time of the first collision occurrence.

5.3 Self-collision Avoidance

The *self-collision avoidance* consists in detecting whether two triangles will collide during the next integration step. Obtaining the current velocities of the triangles we can estimate their future positions and look for collisions. Positions are not affected through this step, only velocities. We will thus prevent collisions from happening.

5.3.1 The Approach

Our approach is similar to the previous one. We will also compare two sequences, but this time we compare the current sequence with the future one. Also, although we will find future colliding pairs of triangles, we will not need to tear them apart but only alter their velocity so that the collision will not happen.

An important difference with *self-collision detection* is that we do not know a priori the future sequence and we must thus approximate it³.

Knowing the current velocities of the triangles, we compute the future depths of the triangles from our sequence and rebuild a new plausible sequence. Like before, we will look for permutations in the two sequences and deduce the pairs of triangles that will collide.

³Note that we could render the simulation with a delay of one frame so that we would perform the next integration step, retrieve the future sequence and perform the *self-collision avoidance* afterwards.

We have the current sequence of triangles and the related depths:

$$\begin{aligned} sequence_{curr} &= [5, 4, 7, 1, 9, 8, 2, 3] \\ Zseq_{curr} &= [0.1, 0.2, 0.5, 1.2, 1.8, 2.1, 2.2, 2.8] \end{aligned}$$

We can compute the z-component of the velocities of the triangles:

$$Vseq_{curr} = [0.5, -0.3, 0.7, -0.1, -0.2, 0.4, 0.2, 0.0]$$

We know how long a time step lasts (eg. $\Delta t = 1$), and we compute:

$$\begin{aligned} \Delta Zseq &= Vseq_{curr} \cdot \Delta t \\ &= [0.5, -0.3, 0.7, -0.1, -0.2, 0.4, 0.2, 0.0] \cdot 1 \\ &= [0.5, -0.3, 0.7, -0.1, -0.2, 0.4, 0.2, 0.0] \\ Zseq_{future} &= Zseq_{curr} + \Delta Zseq \\ &= [0.1, 0.2, 0.5, 1.2, 1.8, 2.1, 2.2, 2.8] \\ &+ [0.5, -0.3, 0.7, -0.1, -0.2, 0.4, 0.2, 0.0] \\ &= [0.6, -0.1, 1.2, 1.1, 1.6, 2.5, 2.4, 2.8] \end{aligned}$$

Again we can reorder this future depths sequence with the current indices sequence to find the future sequence of indices:

$$\begin{aligned} sequence_{future} &= sequence_{curr} \\ Zseq_{future} &= [\mathbf{0.6}, \mathbf{-0.1}, 1.2, 1.1, 1.6, 2.5, 2.4, 2.8] \\ sequence_{future} &= [\mathbf{5}, \mathbf{4}, 7, 1, 9, 8, 2, 3] \\ Zseq_{future} &= [-0.1, 0.6, \mathbf{1.2}, \mathbf{1.1}, 1.6, 2.5, 2.4, 2.8] \\ sequence_{future} &= [4, 5, \mathbf{7}, \mathbf{1}, 9, 8, 2, 3] \\ Zseq_{future} &= [-0.1, 0.6, 1.1, 1.2, 1.6, \mathbf{2.5}, \mathbf{2.4}, 2.8] \\ sequence_{future} &= [4, 5, 1, 7, 9, \mathbf{8}, \mathbf{2}, 3] \\ Zseq_{future} &= [-0.1, 0.6, 1.1, 1.2, 1.6, 2.4, 2.5, 2.8] \\ sequence_{future} &= [4, 5, 1, 7, 9, 2, 8, 3] \end{aligned}$$

The comparison is then:

$$\begin{aligned} sequence_{curr} &= [5, 4, 7, 1, 9, 8, 2, 3] \\ sequence_{future} &= [4, 5, 1, 7, 9, 2, 8, 3] \end{aligned}$$

and the groups of permutations become:

$$groups = [0, 0, 1, 1, 2, 3, 3, 4]$$

5.4 Self-collision Response

5.4.1 Changing Positions

As we have mentioned it before, there are two cases in which we modify the position of a triangle. The first one occurs *only* when two triangles have already collided and we compute the z -position they had by the time the collision occurred. The second case applies when the triangles get too close to one another, right after the swapping or during the *self-collision avoidance*. This second fashion is the ϵ -distance method.

Permutations

This first method retrieves the depth of the triangles at the time of the first collision occurrence within a permutation group. It has been described in the subsection on *self-collision detection* and we will not comment it any further here. However, one must notice that when computing these depths, the two first colliding pair of triangles get the same depth and thus the triangles might intersect. The ϵ -distance procedure we describe next will ensure that the triangles get properly separated as soon as the indices sequence is correct.

The ϵ -distance

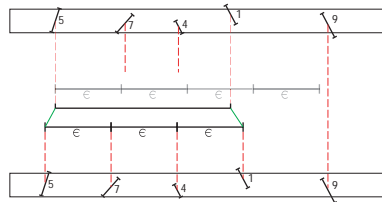


Figure 5.4: Spacing triangles that come too close to one another

Once collision has been detected or avoided, we want the triangles to be separated with a small distance ϵ . Basically, the idea is to equilibrate the spacing around the middle-point of successive close triangles, as illustrates Fig. 5.4. The algorithm for this is not that trivial because it has to be considered that when a group of triangles has been widened, it is likely that the extremities of that group now come too close to some neighbor triangle which had not been considered in the first place.

We present an efficient algorithm to spread long sequences of triangles. Through a pre-processing step, we furnish a sequence of depth corresponding

to the distance separating the successive triangles, eg:

$$depth = [3, 1, 3, 1, 1, 4, 2]$$

For $\epsilon = 2$, there are three gaps that are too short, mainly those on position 1, 3 and 4 (counting from 0). These gaps will be widened and the space they gain will be cut from the two first large gaps to the left and to the right of these successive small gaps. The added values for each gap are stored in a separate list:

$$depth = [3, 1, 3, 1, 1, 4, 2]$$

$$add1 = [-0.5, +1, -1.5, +1, +1, -1, 0]$$

If we sum up these two lists, we see that the gap in position 2 has been cut off too much, and another 0.5 must be gotten back from the updated now closest left and right large gaps. Thus we get:

$$depth = [3, 1, 3, 1, 1, 4, 2]$$

$$add1 = [-0.5, +1, -1.5, +1, +1, -1, 0]$$

$$add2 = [-0.25, 0, +0.5, 0, 0, -0.25, 0]$$

Finally the sequence is:

$$depth + add1 + add2 = [2.25, 2, 2, 2, 2, 2.75, 2]$$

Algorithm 5.4.1 presents in pseudocode the algorithm for the ϵ -distance.

Now consider, like in Fig. 5.5, that one of the triangles has already been pulled out from a deformable body considered in our subsections on *collision detection and response*. We can still use our algorithm, except that we now will fix the position of the first triangles and shift all successive triangles to the right so to fulfill the new sequence of gaps.

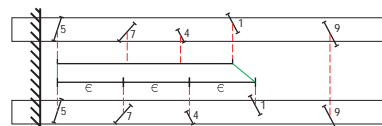


Figure 5.5: The spacing is shifted when a triangle is in contact with the deformable body.

A last consideration on our algorithm is to consider the case when the gaps on the two ends of the sequence do not furnish enough space to spread all triangles (i.e., with n the number of triangles in the sequence and d the distance spread by all triangles, $(n - 1) \cdot \epsilon > d$). In such a case, the algorithm will loop to infinity because it will never get sufficient space for all the gaps. So we modify the algorithm by adding on both sides of the depth sequence, wide enough gaps which will behave as space-sources for the algorithm. At the end, we will withdraw what has been taken from these gaps to the first and the last “real” gaps and if the depth is negative, the whole depth sequence will be shifted to the left.

Algorithm 5.4.1: ϵ -SPACING($depth$)

```
done  $\leftarrow$  false
rest  $\leftarrow$  0
sum  $\leftarrow$  0
size  $\leftarrow$  GETSIZE( $depth$ )
add  $\leftarrow$  INITIALIZE_ARRAY(0, size)
while ( $\neg$ done)
{
  done  $\leftarrow$  true
  i  $\leftarrow$  0
  while (i < size)
  {
    if ( $depth[i] + add[i] > \epsilon$ )
    then
    {
      add[i]  $\leftarrow$  add[i] - rest
      j  $\leftarrow$  i + 1
      while ( $(depth[j] + add[j] \leq \epsilon)$  and j < size)
      {
        sum  $\leftarrow$  sum +  $\epsilon - (depth[j] + add[j])$ 
        if ( $depth[j] + add[j] < \epsilon$ )
        then add[j] =  $\epsilon - depth[j]$ 
        j  $\leftarrow$  j + 1
      }
      sum  $\leftarrow$  sum/2
      if (sum  $\neq$  0)
      then ok  $\leftarrow$  false
      add[i]  $\leftarrow$  add[i] - sum
      rest  $\leftarrow$  sum
      sum  $\leftarrow$  0
    }
    i  $\leftarrow$  j
  }
  for (i  $\leftarrow$  0; i < size; i  $\leftarrow$  i + 1)
  depth[i]  $\leftarrow$  depth[i] + add[i]
}
```

5.4.2 Changing Velocities

For both occurrences (A) and (B) described in the first section of this chapter, we will modify the velocities of the triangles. Our computations will consider the middle-point between two triangles, mainly its positions and velocity.

If the collision is damped, the two colliding triangles velocities will be equal to the one of their middle-point. For an elastic collision, we must consider the relative velocities of the triangles against the middle-points and compute their reflection on a plane orthogonal to the straight line cutting the two triangles in their middle.

The approach

Define the triangles' velocities to be v_1 and v_2 . The velocity of the

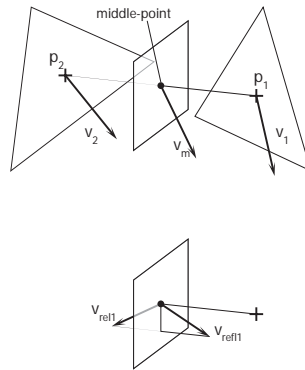


Figure 5.6: The “middle-point” between two triangles and the reflected velocities.

middle-point is thus:

$$v_m = \frac{v_1 + v_2}{2}$$

The relative velocity of the first triangle with the middle-point becomes:

$$\begin{aligned} v_{rel1} &= v_1 - v_m \\ &= v_1 - \frac{v_1 + v_2}{2} \\ &= \frac{v_1 - v_2}{2} \end{aligned}$$

Let p_1 and p_2 be the centers of the triangles and n the normal to the plane orthogonal to the direction of the straight going through p_1 and p_2 . We have:

$$n = \frac{p_1 - p_2}{\|p_1 - p_2\|}$$

The projection of v_{rel1} on n ,

$$proj_{\rightarrow n} v_{rel1} = (v_{rel1} \cdot n)n$$

and the reflected relative velocity,

$$v'_{rel1} = v_{rel1} - (1 + \tau) \cdot proj_{\rightarrow n} v_{rel1}$$

where $\tau \in [0, 1]$ is the *damping* factor.

We can now compute the new velocity:

$$\begin{aligned}
v'_1 &= v_m + v'_{rel1} \\
&= \frac{v_1 + v_2}{2} + v_{rel1} - (1 + \tau) \cdot proj_{\rightarrow n} v_{rel1} \\
&= \frac{v_1 + v_2}{2} + \frac{v_1 - v_2}{2} - (1 + \tau) \cdot proj_{\rightarrow n} v_{rel1} \\
&= v_1 - (1 + \tau) \cdot proj_{\rightarrow n} v_{rel1} \\
&= v_1 - (1 + \tau) \cdot (v_{rel1} \cdot n)n \\
&= v_1 - \frac{1}{2}(1 + \tau) \cdot ((v_1 - v_2) \cdot n)n
\end{aligned}$$

So finally we have:

$$\boxed{
\begin{aligned}
v'_1 &= v_1 - \frac{1}{2}(1 + \tau) \cdot ((v_1 - v_2) \cdot n)n \\
v'_2 &= v_2 - \frac{1}{2}(1 + \tau) \cdot ((v_2 - v_1) \cdot n)n
\end{aligned}
}$$

We verify that for a strong damping coefficient ($\tau = 0$) we obtain:

$$\begin{aligned}
v'_1 &= v_1 - \frac{1}{2}((v_1 - v_2) \cdot n)n \\
&= [v_1 - (v_1 \cdot n)n] + (v_1 \cdot n)n - \frac{1}{2}((v_1 - v_2) \cdot n)n \\
&= v_1^\perp + [(v_1 \cdot n) - \frac{1}{2}((v_1 - v_2) \cdot n)]n \\
&= v_1^\perp + [(v_1 - \frac{1}{2}(v_1 - v_2)) \cdot n]n \\
&= v_1^\perp + (\frac{1}{2}(v_1 + v_2) \cdot n)n \\
&= v_1^\perp + (v_m \cdot n)n
\end{aligned}$$

which means that the velocity keeps its orthogonal to n components (v_1^\perp) and its parallel component becomes equal to the one of the middle-point⁴. Because we obtain a similar equality for v'_2 , we have proven that, with $\tau = 0$, when two parts of the cloth get into collision, they receive an equal velocity component parallel to the direction of the straight line cutting the center of the triangles.

Fig. 5.6 illustrates our computations.

5.5 Putting All Together

Now that we have exposed the different ideas and information we have for implementing a global *self-collision* procedure for our system, we still have to make clear how we put everything together. Important points to be implemented in a reliable way are the following:

⁴Note that we will later also implement friction in the self-collision process by altering the orthogonal v_1^\perp component.

- Neighbors: find a way to avoid considering neighboring triangles in a mesh within a same sequence but still consider each of them with further triangles.
- Past and future: when considering permutations with the *past* information how can we merge this with considering also the *future* information?
- Epsilon: the ϵ -distance procedure regulates the spacing between the triangles. Not only the information of the modified depths should be considered, but an exact description of which triangle and in what way it has been affected must be transmitted.
- Velocities: the two preys of the *self-collision* are the positions and the velocities. We must express clearly when and in what way both of them are modified.

We will next precisely explain how we implemented the ideas we presented earlier while considering these mentioned points.

5.5.1 Skipping Neighbors

Considering the sequence of triangles we obtain from an LDI stick we want by the mean of the ϵ -distance method, ensure that triangle do not get too close from one another and if necessary tear them apart. This method will make the system collapse when two triangles are actually neighbors in the mesh of the cloth (Fig. 5.7-A).

A simple way would be to remove from the sequence any triangle which has a neighbor already identified (Fig. 5.7-B). The ϵ -distance would thus be inserted between two triangles who actually have another triangle in between and the ϵ -distance requirement will no longer be fulfilled.

So we must modify the ϵ -distance algorithm presented previously so that intervals between neighbor triangles are not altered (Fig. 5.7-C).

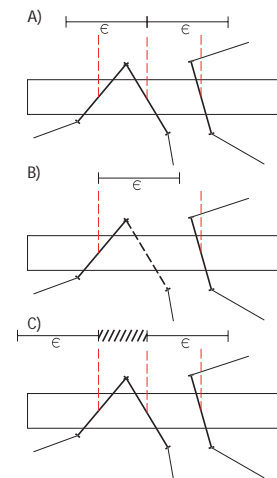


Figure 5.7: Skipping neighbor triangles.

5.5.2 Future After Past

As we have seen, it is important that the *self-collision detection* and the *self-collision avoidance* are implemented together. Both methods work similarly, one working with the information from the past, the other by estimating the future. However, it is difficult to implement them together. It might often occur that a triangle that has already collided with another (or more) also

might collide with a further one during the next integration step. Although only the first method would alter the position of the triangle, both deal with its velocity. Computing the resulting velocities in both cases and then taking the average would not be a good idea because it would not agree with any of the methods any more.

The implementation will thus cheat a little bit in the sense that the second method will work with the velocities obtained from the first one. We justify this by the fact that the velocities also get altered during the *detection* and the ϵ -*distance* steps and therefore relative velocities of close triangles are likely to be low and should reduce the number of collisions during the next integration step. Thus, only a few collisions will be avoided and two triangles prone to collide will also see their velocities only slightly altered.

5.6 The *Self-Collision* Algorithm

Now that we have expressed the main ideas behind our approach and the different aspects to consider to implement them together, we illustrate the procedure to go through by a simple example. Finally we will give the whole algorithm in pseudo-code.

5.6.1 The Procedure

We give here the procedure to follow that will allow to perform all the necessary aspects for the *self-collision algorithm*. The algorithm works with one-directional tables holding sequences of indices, depths or velocities with respective i , d and v prefixes. These tables are also denoted by a suffix which has the following meanings:

- *current*: for the information we have on the present configuration of the triangles when we begin with the self-collision procedure;
- *old*: for the information we have of the configuration of the triangles in the preceding time step;
- *future*: for the information we approximate the configuration of the triangles in the next time step;
- *final*: for the information we get after we have been through the whole procedure;
- *tmp*: for a temporary information.

Also, for a better comprehension, the example is illustrated in Fig. 5.8. The indices of the triangles are arbitrarily denoted by capital letters in the order they have been retrieved when we queried for the triangles sequence in an LDI stick.

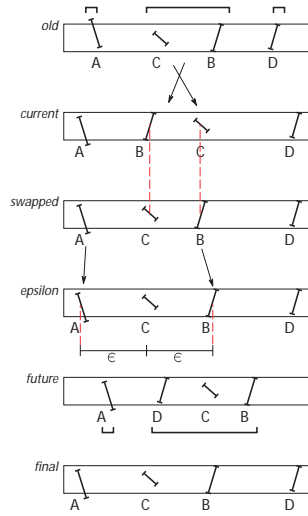


Figure 5.8: The successive steps during the self-collision procedure

5.6.2 First Collision Occurrence

Now, the first information we get is the actual indices sequence of the triangles in the LDI stick from which we know the depths (an ordered sequence):

$$\begin{aligned}
 i_{current} &= [A, B, C, D] \\
 d_{current} &= [Dc_A, Dc_B, Dc_C, Dc_D]
 \end{aligned}$$

To get the *old* indices sequence, we first compute the previous depths for the triangles and obtain the non-ordered sequence of depths:

$$d_{old} = [Do_A, Do_B, Do_C, Do_D]$$

We order this sequence and reproduce the permutations in the current sequence of indices to get the old sequence of indices:

$$\begin{aligned}
 d_{old} &= [Do_A, Do_C, Do_B, Do_D] \\
 i_{old} &= [A, C, B, D]
 \end{aligned}$$

While doing this last step, we keep trace of *groups of permutations*. By *groups of permutations* we mean the successive indices of triangles who have been permuted.

We now have the index sequence to which we must return. We take all pairs of successive triangles in the old indices sequence and compute the times at which they have collided. The depth sequence is corrected so that all triangles recover the depth they had at the time of the earliest collision occurrence within a group.

Triangles in a common group have all collided together and we need to keep trace of them to adjust their velocity. Their new velocity will become all equal to the average velocity of the group⁵.

$$\begin{aligned} v_current &= [Vc_A, Vc_B, Vc_C, Vc_D] \\ group &= [0, 1, 1, 2] \\ v_tmp &= [Vc_A, Vc_{BC}, Vc_{BC}, Vc_D] \end{aligned}$$

The swapping step consists of inverting the current depths of the triangles. This means, we keep the current depths sequence but need an updated sequence of indices. This sequence is i_old .

$$\boxed{i_final = i_old}$$

5.6.3 ϵ -distance

We now have i_final but still need d_final and v_final . After the ϵ -distance process we will have d_final .

We have proposed an algorithm for ϵ -distance which does not consider neighbor triangles. We slightly modify it by considering a one-dimensional binary array indicating whether a gap is allowed to be modified during the process. The pseudo-code is given in Algorithm 5.6.1

Once the algorithm has been performed, we get the final sequence of depths.

$$\epsilon(d_current) = [Dc'_A, Dc'_B, Dc'_C, Dc_D]$$

$$\boxed{d_final = \epsilon(d_current)}$$

Remark: Velocities get altered also during the ϵ -distance step. To do so, we keep trace of groups of successive triangles that have been separated by the ϵ -distance from one another. The triangles within a common group will have their velocity adjusted to the average velocity of the group.

5.6.4 Velocities

The last information missing is the one related to the velocities of the triangles. We have already modified them during the swapping and the ϵ -distance steps when their corresponding triangles have collided or were too close. In the present step, which is the *collision avoidance*, we make again use of the current velocities (we will use the velocities stored in v_tmp2) to estimate

⁵In fact we compute the average velocity as we described it earlier. We have seen that with a strong damping ($\tau = 0$) the triangles get the velocity component of the middle-point.

Algorithm 5.6.1: ϵ -SPACING($d_current, d_epsilon, neighbors, group2$)

```

size ← GETSIZE( $d\_current$ )
gaps ← INITIALIZE_ARRAY(0, size + 1)
neig ← INITIALIZE_ARRAY(0, size + 1)
group ← INITIALIZE_ARRAY(0, size + 1)
gaps[0] ← 100
gaps[size] ← 100
neig[0] ← 0
neig[size] ← 0
cursor ← 0
flag ← 0
done ← false
sum ← 0
for ( $i \leftarrow 0; i < size; i \leftarrow i + 1$ )
{
  gaps[ $i$ ] ←  $d\_current[i] - d\_current[i - 1]$ 
  neig[ $i$ ] ← neighbors[ $i - 1$ ]
}
while ( $\neg done$ )
{
  done ← true
  for ( $i \leftarrow 0; i < size; i \leftarrow i + 1$ )
  {
    if (neig[ $i$ ] = 1 or gaps[ $i$ ] ≤ epsilon)
    {
      then group[ $i$ ] ← 1
      else group[ $i$ ] ← 0
    }
  }
  while (cursor < size + 1)
  {
    while (group[cursor] ≠ 1 and cursor < size + 1)
    {
      cursor ← cursor + 1
    }
    if (cursor ≥ size + 1)
    {
      then break
    }
    flag ← cursor - 1
    sum ← 0
    while (group[cursor] = 1)
    {
      if (neig[cursor] ≠ 1)
      {
        then
        {
          sum ← sum +  $\epsilon - gaps[cursor]$ 
          gaps[cursor] ←  $\epsilon$ 
          cursor ← cursor + 1
        }
      }
    }
    if (sum ≠ 0)
    {
      then done ← false
      gaps[cursor] ← gaps[cursor] - sum/2
      gaps[flag] ← gaps[flag] - sum/2
    }
    cursor ← 0
  }
  gaps[0] ← gaps[0] - 100
  gaps[size] ← gaps[size] - 100
  d_epsilon[0] ←  $d\_current[0] + gaps[0]$ ;
  for ( $i \leftarrow 1; i < size; i \leftarrow i + 1$ )
  {
    group2[ $i - 1$ ] ← group[ $i$ ]
    d_epsilon[ $i$ ] ←  $d\_epsilon[i - 1] + gaps[i]$ 
  }
}

```

the sequence in the next time step. Having i_final , d_final and v_tmp2 , we can compute d_future .

$$d_future = [Df_A, Df_C, Df_B, Df_D]$$

Again we order this sequence while reporting the permutations to the final sequence of indices and obtain:

$$\begin{aligned} d_future &= [Df_A, Df_D, Df_C, Df_B] \\ i_future &= [A, D, C, B] \end{aligned}$$

In a similar way to the one we have used before, we form groups of permutations and a common velocity is attributed to all triangles from a same group.

$$\begin{aligned} group &= [0, 1, 1, 1] \\ v_tmp &= [V_{CA}, V_{CB}, V_{CD}, V_{CB}] \end{aligned}$$

Finally we get the velocities for each triangle:

$$\boxed{v_final = v_tmp}$$

5.6.5 Attribution

At last we have all the necessary information for our *self-collision* approach. A last point we must not omit is that all computations are done with positions and velocities that were averaged from the three vertices of each triangle. The algorithm also returns values that are valid for the triangles and we must have an correct way to distribute them to the vertices in a way that the values for the vertices do not become equal but that the average of them still matches the output of the algorithm. Therefore it is important that we keep the *current* information of the vertices and only attribute their new values at the end. This is actually verified in our algorithm, as we only work on arrays aside from the vertices.

If we define z and v the average current position and velocity of a triangle whose three vertices have values z_i and v_i ($i \in [1, 3]$) we computed the average values as

$$z = \frac{z_1 + z_2 + z_3}{3}$$

Now the *self-collision* algorithm outputs z' and v' . We compute:

$$\Delta z = z' - z$$

and we alter the vertices

$$z'_i = z_i + \Delta z$$

We verify that:

$$\begin{aligned} \frac{z'_1 + z'_2 + z'_3}{3} &= \frac{z_1 + \Delta z + z_2 + \Delta z + z_3 + \Delta z}{3} \\ &= z + \Delta z \\ &= z' \end{aligned}$$

The same computations are valid for the velocities.

5.6.6 Marking Triangles and Vertices

As we have seen, after the self-collision step we modify the triangles and in particular their vertices. It can occur that a triangle or a neighboring one is associated with another LDI stick, in which case the triangle or some of its vertices are modified more than once. Also, when we get the current information of a triangle, it might have been modified during the process of another LDI stick.

In our current implementation, we mark triangles and vertices which have been modified and prevent any further modification (computations are done but the values do not get assigned). A more proper manner would be to attribute the final values once we have gone through all the sticks of the LDI instead of doing it after each stick.

5.6.7 The Algorithm

We give in pseudo-code the algorithm we follow for the *self-collision* procedure. The different steps are executed through method calls. (See Algorithm 5.6.2)

5.7 Discussion

5.7.1 When to Change Velocities

As we presented the self-collision algorithm, the two things on which we can have an influence are the position and the velocity of the triangles. It is clear that when a collision has been detected, position and velocity must be altered, whereas for collision avoidance position remains the same. We can question whether velocity has also to be modified when we perform the ϵ -distance step.

The goal of the ϵ -distance step is to lower the number of possible collisions during the next time step. The space inserted between two successive

Algorithm 5.6.2: CLOTHSELF-COLLISION()

```
INITIALIZE_ARRAYS()
selfCollider.INITRETRIEVAL()
while selfCollider.GETNEXTSEQUENCE(i_current)
{
  if GETSIZE(i_current) < 2
  then continue
  COMPUTE_CURRENT(i_current, d_current, v_current)
  COMPUTE_OLD(i_current, d_old)
  CORRECT_CURRENT(d_current, d_corrected, d_old, d_current, v_current, time_step)
  COMPUTE_FUTURE(d_future, d_current, v_current, time_step)
  for i ← 0; i < size; i ← i + 1 swap2[i] = i
  ORDER_DEPTHS(d_future, swap2)
  GET_GROUPS(group, swap2)
  ASSIGN_VELOCITIES(v_tmp1, v_current, group)
  for i ← 0; i < size - 1; i ← i + 1
  {
    if ARENEIGHBORS(i_current[i], i_current[i + 1])
    then neighbors[i] ← 1
    else neighbors[i] ← 0
  }
  EPSILON_SPACING(d_corrected, d_epsilon, neighbors, group)
  ASSIGN_VELOCITIES(v_final, v_tmp1, group)
  for i ← 0; i < size; i ← i + 1
  {
    dz ← d_epsilon[i] - d_current[i]
    dv ← v_final[i] - v_current[i]
    UPDATE_TRIANGLE(i_current[i], dz, dv)
  }
}
```

triangles in the sequence ensures that the cloth pieces get separated and that the displaced triangles will pull their neighbors with them. Also, when the cloth is folding, this distance prevents from seeing the lower regions through the higher ones. Finally, it also aims at keeping a certain stability when the cloth rests folded on the ground for example.

The idea behind affecting the velocity is to prevent that triangles that have been separated come too close again after the next time step. However, the displaced triangles cannot be considered as having collided together and an accurate computation of reflected velocities is hard to perform, even more so, when large sequences of triangles are separated. As we implemented it, for the velocity attributed to the group of displaced triangles, we take the average velocity over them (i.e. strong damping). The velocities get modified quite often that way. Not only a triangle can be considered during the ϵ -distance step, but also during the collision-detection and the collision-avoidance steps.

We experimented on the ϵ -distance step, both with velocity changes and without. The simulations behaved quite differently. In the first case, the system was not stable and the cloth trembled when two regions overlapped. In the second case, we got more stability but the cloth behaved more unusually; regions that are treated by the algorithm begin to move more slowly

while the surrounding collision-free regions still fall at normal speed.

5.7.2 Friction

Another factor that affects stability is that we did not implement friction in the self-collision algorithm. As we have already shown, damping applies when velocities are attributed to colliding groups, but friction cannot be implemented along with the damping. Indeed, we work with a unique vertically directed LDI and consider positions and velocities by their z-component; friction has an effect on the orthogonal components with which we currently do not deal.

To implement friction, we should thus store and update three times more information. However, friction would have a great influence on stability. When we simulate a cloth falling on a flat plane and that the cloth folds, we can observe how the forces generated from the folds pull the upper layer of the cloth. Although these strengths are small, because no friction exists, the cloth will unfold (see Fig. 5.9). While the two cloth regions rub, the upper region vibrates. This vibration is due to the changing of geometry of the cloth piece, affecting the bonding-box of the cloth which has a direct effect on the size of the LDI-sticks. So, not only the LDI-sticks change constantly, but the sequences we get from the sticks are always different. Therefore, the system gets unstable. Implementing friction would prevent the cloth from rubbing and the system would keep calm.

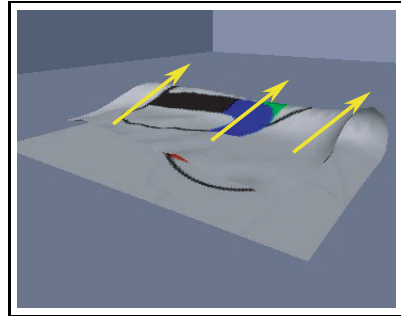


Figure 5.9: Friction is not implemented in the self-collision algorithm. Therefore, the folds exert forces on the top layer of the cloth which is pulled until the cloth lies flat.

5.7.3 Side-collisions

The presented self-collision algorithm uses a single LDI whose sticks are directed vertically. Therefore, the depth sequences represent the z-coordinate of the triangles and the investigation of the permutations is only focused on z-coordinate permutations. With a fine enough LDI resolution, we can assume that when permutations are detected, triangles really have collided as they are both detected to be in the same stick and thus one is above the other. On the other hand, cases such as one illustrated in Fig. 5.10 cannot be detected. We can refer to such cases as *side-collisions*. The figure shows two regions of a same cloth that have triangles whose z-coordinate remain the same over a time step but their x,y-components have changed such that the two parts collide. As there is no permutation detected, the collision will

not be corrected. In our simulations we had good results when considering vertical collisions such as when we let the cloth fall onto the floor.

This side-collision problem could be resolved if we used a 3-directional LDI. However the ideas we have exposed throughout this chapter shall all be deeply reconsidered to work with such a structure. By the same time, it would perhaps also be possible to find a way to implement friction in another way than the one exposed in the preceding section.

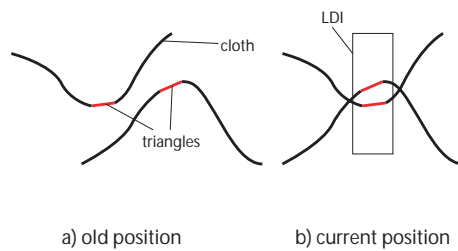


Figure 5.10: Side-collision cannot be detected with our algorithm as we look for depth permutation in a single direction.

We could not, during this work, go as far as we would have wanted in the self-collision investigations. However we believe that there is a great potential in this approach, because the simulations we achieved were acceptable and ran in real-time.

Chapter 6

Experiments

Now that we have presented the theoretical part of our work, we expose with what we dealt in the simulations during the development (Section 6.1) and give a short study (Section 6.2) of some common scenarios as we will measure the distribution of the processing times of the different tasks the simulation loop has to perform. Through this, we will know precisely which are the expensive tasks so that future work can focus on these to optimize the overall performance of the system. In the last section (Section 6.3), we present as an example, the application where we integrated a walking avatar in our system and observed its interaction with a poncho-like cloth.

6.1 The Environment

Before we go to an analysis of the simulations which we performed with our cloth-simulation applications, we show which parameters can be tuned. We also show, what kind of scenarios we could experimentally carry out by describing the different volumes we used for testing our system during its development.

6.1.1 Most Important Simulation Parameters

The parameters for our cloth simulation are grouped in a file which the application will read before it sets the simulation up. In that file, we give the location of the cloth file description and the one of the volume. In the cloth file we find the vertices of the cloth-points and the description of the mass-spring systems. Some other parameters are also found, like the bending, the stiffness and the mass of the cloth-points. In the volume file, only the geometry is given by a list of vertices and face indices.

Along with the location of these two files, all parameters concerning our collision algorithms are listed. Aside from the camera specifications, movie recording options and scene setup, two groups of parameters are given: the

one related to the cloth integration process and the one for the collisions. We describe next the most important of them. See Appendix C for an example of that file.

Cloth-integration Parameters

First, the integration time step is given (TIMESTEP). This parameter was usually set to values between 0.005 and 0.01 sec.

In this group, we can also indicate how many times the integration and collision steps should be performed, before the deformation of the volume and the LDIs are updated. The parameters are INTSTEPS and COLSTEPS which were tuned from 1 to 10 cycles per simulation step.

Collision Handling Parameters

Regarding collision handling, some parameters influence the weight and the rapidity of the deformation of the volume (WEIGHT and OFFINCR respectively) which are set to values between 0 to 10, 0 being low influence. Then, the volume's behavior can be selected from a list of 5 deformations (OFF_DEFORM) and from a list of movements, too (OFF_LIFT, OFF_ROT_X, OFF_ROT_Y and OFF_ROT_Z).

Finally, the damping and the friction intensity can be adjusted from 0 to 1, and the distance-spacing for the volume collision response (aura) and the self-collision (ϵ) are defined.

Note that it is possible to modify some of the parameters during runtime.

6.1.2 Working with Deformable Volumes

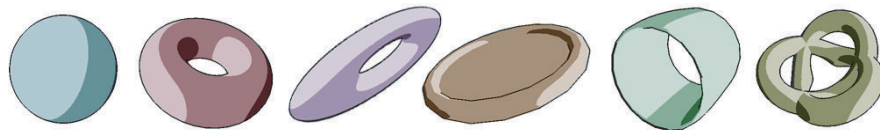


Figure 6.1: The volumes that were used for our simulations: a sphere, a torus, a deformed torus, a plate, a “dryer” and a “twisted knot”.

For testing our implementations, we launched our simulation with different volumes like the ones illustrated in Fig. 6.1. The difference of geometry among them allowed us to analyze the collision detection and the response to the particular volumetric characteristics. The sphere, with the most trivial geometry, allowed to test early implementations on our collision detection

algorithm. With the different tori, we could either let the cloth fall into their hole or let it lie on the highly curved extremities of the deformed torus, and analyze the reactions. On the plate we tested the two-side collision detection. The dryer was used rotating around its axis to perform the “washing-machine” demonstration. The twisted knot, finally, which has a more complex geometry, with thin segments, corners and different curvatures, was used for simulations to show how the cloth would react on complex geometries.

Once we select one of these volumes, we can decide on a displacement and a deformation function. The displacement functions describe mainly a rigid, sinusoidal lifting movement that can be mixed together with rotations along the three axes. While the volume moves, we can choose a deformation function to be added from the following five:

- **clock**: the volume gets stretched in a direction that rotates along the z-axis;
- **wings**: the center of the mesh gets pulled downwards while the exterior is pushed upwards and vice-versa, alternatively;
- **grow**: the overall scale of the volume grows and shrinks;
- **bruno**: a particular twisting deformation, called after Bruno (from CGL) as he has implemented it;
- **drum**: implemented especially for the plate volume, so that it behaves like a vibrating drum; this was to illustrate the two-side collision detection.

The above list of volumes and the displacement and deformation functions allowed a large number of combinations and a lot of scenarios to be analyzed. The cloth remains stable in all of the different situations we brought it to.

6.2 Discussion

The simulation main loop goes through a sequence of processes. In the first part, it processes several times *cloth integration* and *collision detection* and *collision response*. The number of times these processes are effectuated is a parameter the user can tune. Usually, a simulation runs 5 to 10 times this first part before it would proceed to the second phase. The *integration* makes the cloth move. The *collision processes* detect and correct collisions with the cuboid, the volume and the cloth itself.

In the second phase, the scene and the different LDIs are updated. Mainly, the LDI for the cloth is computed, the volume is deformed and its LDI adapted. Finally the scene is rendered on screen. Note that when

LDIs are generated, the vertices of the cloth and the volume must all be transferred from their respective data-structure to the LDI generator.

The average processing times for a common simulation (cloth falling on a torus and then on the ground) involving both volume and cloth’s self-collision are given in table 6.1. These times are evaluated over one second of simulation, where around 30 steps of simulation were executed each performing the integration and the collision handling five times before the scene is updated. The mesh in that example consists of 1600 vertices and 3200 triangles and the cloth has 441 points.

task	time [ms]	percentage [%]
cloth integration time	1.027	6.5
cuboid collision process	0.053	0.3
self-collision process	0.206	1.3
volume-collision process	0.607	3.8
cloth coordinates transfer	0.027	0.2
self LDI generation	1.891	11.9
model deformation	0.067	0.4
volume coordinates transfer	0.064	0.4
volume LDI generation	10.60	66.6
visualization	1.366	8.6

Table 6.1: Average processing times and distribution for a common simulation

From these measures, we see clearly that the most expensive tasks are the LDI generations with almost 80% together. The generation time of the volume’s LDI is about 6 times longer that for the self-collision. We will see later that it actually is about 4 time longer. Clearly, this is because we only use one LDI for the self-collision and three for the collisions with volume. The coefficient is however higher than 3 because the 3-directional LDI performs the *stretching* which the self-collision process does not.

Beside the integration and visualization times, we see that the collisions processes are relevant although much smaller than the LDI generations. Also, the self-collision process is here 3 times lower that the process for collisions with the volume. This is however greatly dependent on the LDI resolution we use: recall that the self-collision algorithm traverses all LDI-sticks, whereas the cloth-to-volume one traverses all cloth-points.

6.2.1 The Measurements

Our measurements are based on two similar scenarios. We let the cloth fall onto a skew volume so that it slides onto it and falls down on the floor. We ensure that when the cloth begins to touch the floor, its higher region still

collides with the volume before it finally flattens on the ground.

While the simulation run, we time the different tasks we enumerated previously. With such scenarios we will have successive stages:

- the cloth falls freely;
- the cloth collides with the volume;
- the volume collisions end while self-collision begins;
- only self-collision is performed;
- finally the cloth lies flat on the ground.

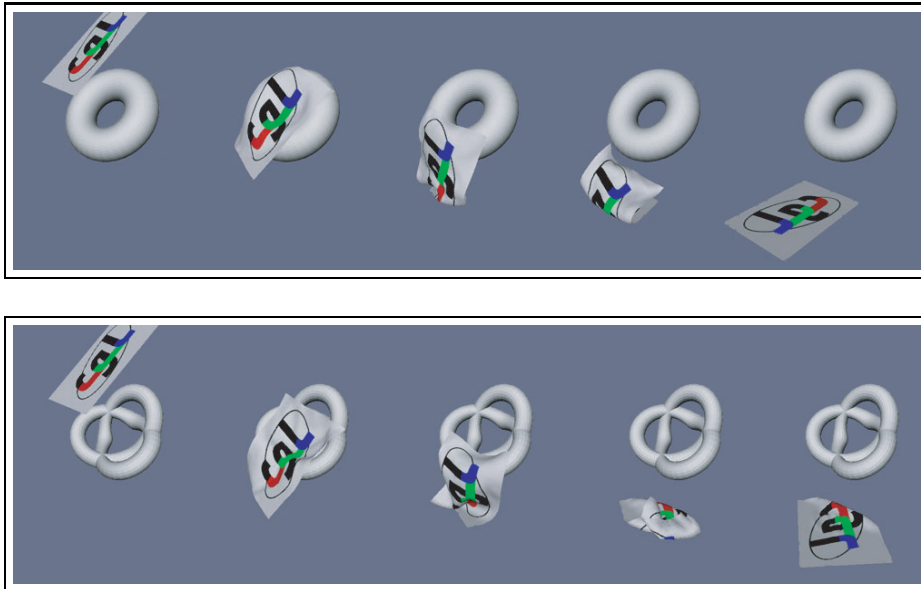


Figure 6.2: The five stages of the scenarios: cloth falling freely, volume collision only, volume collision and self-collision, self-collision only and lying flat.

The scenarios differ in that we changed the volume with which the cloth interacts. In the first scenario, we use a *torus* whereas in the second, we switched to a *twisted knot*. Fig. 6.2 illustrates the five stages for the two scenarios. Note that in the second scenario, the cloth ended folded.

From the measurements (see Appendix B for all numerical results), we draw the evolution of the processing times for the different tasks the simulation pipeline has to go through (see Figs 6.3 and 6.4). Each scenario was launched three times while we changed the LDI resolution (32×32 , 64×64 and 128×128 sticks). For each of these simulations, we get two graphs. The first one shows the processing times for collisions with the volume and with

the cloth itself. Also, the visualization and cloth integration time evolutions are shown; their evolution is constant, but we wanted to illustrate that the collision *detection* and *response* time are of the same order of magnitude. The Y-coordinate is in millisecond while the X-coordinate corresponds to the measurements steps. Indeed, we let the simulations run for about one second until we make an average of the different times over this interval.

The second graph shows the generation times for the different LDIs. Also, we drew on the same graph, the frame-per-second rate (FPS). Note that as before, the X-coordinate stands for the measurements steps whereas the Y-coordinate gives the time in milliseconds and the rate of the FPS; both scales are confused.

6.2.2 Analysis

As we have written before, the evolution of the visualization and cloth integration times are only presented to show the order of magnitude of the collision handling processes. The integration time is here lower than the visualization time with about 1 ms versus 1.4 ms. These values do not change over all the simulations and correspond to the average values found in the table 6.1. However, the integration is directly dependent of the number of cloth-point, 441 ($21 \cdot 21$) in our case. The complexity of the integration algorithm is of $\mathcal{O}(n^2)$, with n the number of cloth-points.

LDI Generation Time

At a first glance, we clearly see that the different scenario stages have a direct effect on the collision handling processes. We see how the collision with the volume interlaces with the self-collision.

Also, we note that the LDI generation times are constant over the time, especially for the volume collision; we see however that in the case of the self-collision, it take a little more time when it comes to the self-collision stage. This is particularly noticeable for the two graphs related to the twisted knot with higher LDI resolution. If we take a look at Fig. 6.2 we see that the self-collision is much more complex in that scenario than the one with the torus: there are many folds and thus more depths are added in the LDI-sticks. When the cloth is folding, its bounding-box gets smaller but the number of sticks remain the same, so triangles might be rendered in more sticks at the same time. On the other hand, the 3-directional LDI generation of the volume is independent of the scenario stage. The volumes keep its geometry. Although the volume does not deform it would not affect the generation time, there are always as many sticks.

The LDI generation time grows exponentially with the resolution; this is, it grows linearly with the number of sticks. It take around 10 ms at low resolution, grows to 20 ms and ends with 40 ms with the highest resolution.

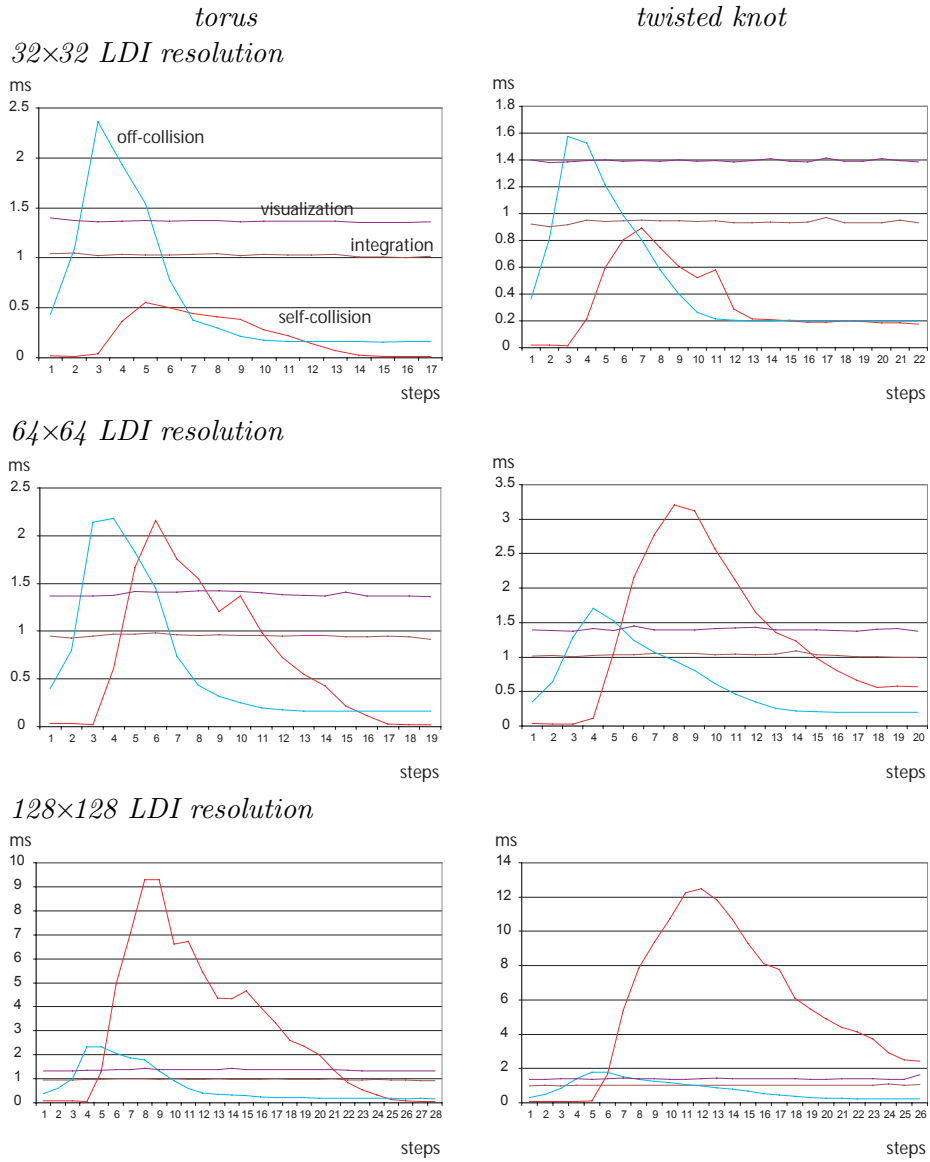


Figure 6.3: Time usage of the collision handling, cloth-integration and visualization rendering tasks. The graphs on the left are for the scenario with the *torus* and the ones on the right for the scenario with the *twisted knot*. Measures are effectuated for both scenarios at three different LDI resolutions. The values are the averaged computations time over a second of simulations.

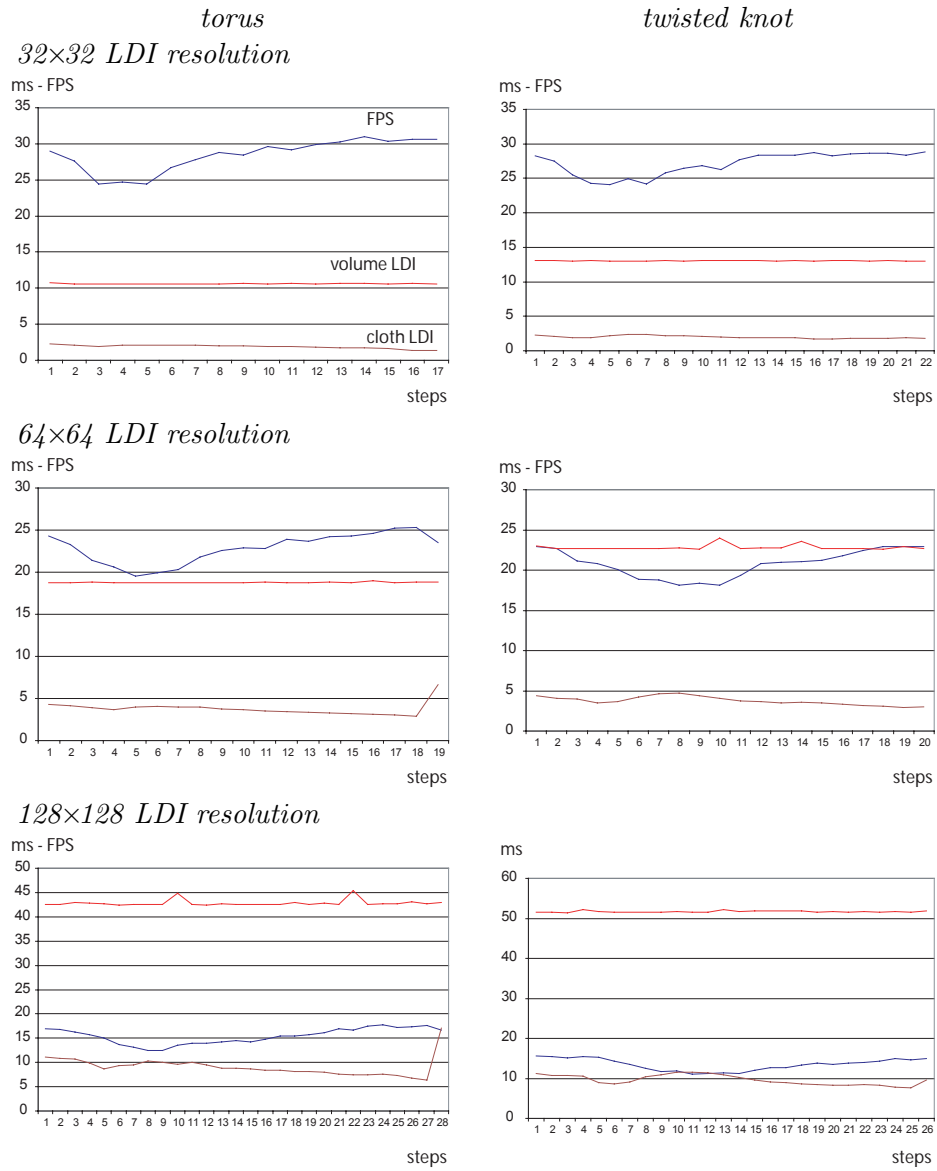


Figure 6.4: Time usage of the LDI generation for the volume collision and the self-collision. The graphs on the left are for the scenario with the *torus* and the ones on the right for the scenario with the *twisted knot*. Measures are effectuated for both scenarios at three different LDI resolutions. The values are the averaged computations time over a second of simulations.

Therefore, it is preferable to work with low resolution LDIs. Even though we would get better stability and precision with high resolution, we have shown earlier in this work that when the section of the sticks become relatively smaller than the triangles, there is no more profit regarding these criteria. Experiments show that with low resolution we get sufficiently stable results and gain a lot in time consuming.

As we have mentioned it before, the 3-directional LDI takes between 4 and 5 times longer to be generated than the single LDI for the cloth. Again, not only three LDIs are generated in the first case, but also the stretching phase is not performed in the second one.

A last remark regarding the LDI generation time: we can notice a sudden increase of time at the last measurement steps for the LDI of the cloth. This increase is due to the abortion of the simulation which terminates some processes and frees memory before it finally killed the application.

Collision Handling Time

Although the two scenarios are very similar, we can see important differences. Regarding the geometry of the two considered volumes, the twisted knot is more complex than the torus. This would require more depths stored and stretching along a single stick during the LDI process. This can be noted as the necessary time is 20% larger (13 ms against 11 ms for the low resolution case) in the second scenario, the LDI of the cloth generation time remaining unchanged.

On the other hand, the geometry difference has another impact on our measurements: the volume collision handling with the twisted knot takes about 25% less time (1.6 against 2.2 ms). Indeed, the twisted knot has thinner segments and the cloth lies on less surface than on the generous torus. However the gain of time is withdrawn by the LDI generation time loss. The overall frame rate of the simulation is nevertheless not sensitive to these geometry related time variances (smaller than 3%).

Self-collision handling is naturally independent of the shape of the volume. The increase of processing time in the second scenario is due to the fact that the cloth falls differently on the floor and more self-collisions occurs; this is observable at any LDI resolution. However it is interesting how the self-collision handling takes over the volume collision with the resolution increasing. In reality, the volume collision handling remains constant independently, whereas the self-collision peaks from 0.5 ms to 2.2 ms and then 9.3 ms for the first scenario and 0.9 ms, 3.2 ms and 12.5 ms for the second. Again, this exponential growth is related with the quantity of LDI-sticks.

The volume collision process does not depend on the LDI resolution as the algorithm queries for penetration for every cloth-point, which are always of same number. At the opposite, the self-collision algorithm considers all LDI sticks, one after the other, and is therefore directly influenced by

the resolution. During the simulation, we noticed no great increase in the stability of the cloth at higher resolution.

A further point to remark, is that the self-collision consumes no time as long as no collision occur. The volume collision, on the other hand, keeps using 0.2 ms even when the cloth is not touching it. Note that for the twisted knot scenario, the self-collision handling does not fall back to zero as the cloth rested folded onto the floor. The reason is that in the self-collision algorithm, sticks holding only one layer of depth are not considered and skipped; in the volume collision however, we still consider all the cloth points at any stage of the scenario, and the queries to the LDI are always performed. An improvement could be to make queries only for the cloth-points that would be located inside the intersection of the bounding-boxes of the cloth and the volume.

Summary

In our analysis, we showed the influences on the simulations of the LDI resolution and of the complexity of the volume’s geometry. Also, the most time consuming tasks are the generation of the two LDIs.

However, we could clearly stress that using a high resolution for the LDI is not necessary to produce reliable simulations and that the complexity of self-collision does not drastically affect their overall efficiency.

However, there is still a decay in the frame rate when the collisions occur. With a resolution of 32×32 for the LDI we can ensure a frame rate of 25 fps¹.

6.3 Application: the Walking Avatar

The most obvious application for our cloth-simulation system is to simulate cloth on a moving avatar. We included in our system the open source 3D character animation library *Cal3d*. This library was developed by Bruno Heidelberger (CGL/ETHZ).

The library allows to load different avatar models, from which we chose “Cally”, and to blend between different animation sequences. We retained the walking behavior as it is the most commonly used in character animation demonstrations.

The idea of including the Cal3d library in our system was present since the start of our development. As we began the collision detection implementation, we had to put that idea aside when we encountered that the mesh of the character was not properly closed and that the triangles of the

¹The measurements were obtained on a Pentium IV with a frequency of 2.8 GHz and 1GB of memory.

mesh were not all oriented in the same direction. We could reintegrate Cally when we developed the 3-directional LDI, which allowed to use open meshes, and the two-side collision detection, which lead to side independent collision response.

We integrated Cally in our system and had to adapt somehow our *model* architecture to allow it to work both with the volumes and with Cally. The LDI gets generated over the whole mesh of the character and the deformations steps in the volume are switched with the animation steps. We soon could launch simulations by dropping the cloth on to Cally and see how it interacted with the new model.

The system worked fine but on some difficult regions of the mesh. Mainly the wrist and the fingers of the character are too thin. If parts of the mesh are much smaller than the actual section of the LDI-sticks, we get great instability as LDI-sticks will not always be created in such regions. This was noted as the cloth, from one point time to the next, was first caught by the hand, and then no more. The instability became great enough to make our system crash.

An optimization to that behavior would be to generate a hierarchical LDI structure over the mesh of the character. We present such an extension to our work in Chapter 7.

6.3.1 The Poncho

Although we can have problems like the one exposed in the previous section, we still wanted to illustrate our cloth-simulation system with a walking avatar. Moreover, we wanted to design a simple cloth that would fit Cally. However, we lack of adapted tools for designing complex clothes such as trousers or T-shirts, and looked for a simple cloth desing like a poncho². The poncho consists in a rectangle in which we have cut out a square in its center so to let the head pass through. Also, we refined the extremities so that the cloth would not get in contact with the hands.

The scenario for that simulation is illustrated in Fig. 6.5. We can see in the left most image how the poncho looks like in its original state. The cloth will begin to fall free onto Cally's shoulders, then its extremities will go further down until they hit the back and the belly of the model. After that point, the simulation loops.

6.3.2 Discussion

We made the same measurements for the Cally scenario as we did for the torus and the twisted knot in Section 6.2. We give the average time processing over the cyclic part of the simulation (Table 6.2). The numerical values

²Note that if we wanted to simulate trousers or a dress, we would have needed to fix certain points of the cloth to the mesh of the character, so as to act like a belt.

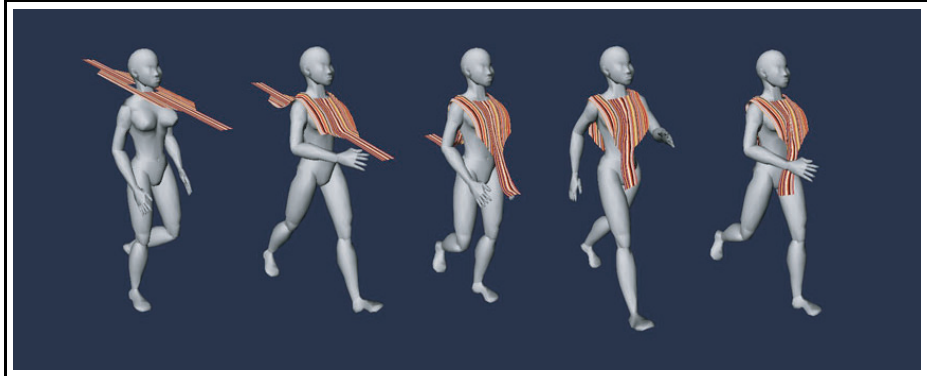


Figure 6.5: Cally walking with a poncho falling onto her. The simulation starts with the cloth falling free and continues in cycles.

of the measurements can be consulted in Appendix B and the processing time evolutions are drawn in Fig. 6.6.

task	time [ms]	percentage [%]
cloth integration time	2.014	2.5
self-collision process	0.820	1.0
volume-collision process	3.140	3.9
cloth coordinates transfer	0.071	0.1
self LDI generation	4.612	5.8
model deformation	0.237	0.3
volume coordinates transfer	0.707	0.9
volume LDI generation	66.382	82.9
visualization	2.022	2.5

Table 6.2: Average processing times and distribution for a common simulation

As a first observation, we see on the graphs that the visualization and integration time are again constant over the time. Both take however more time than in the simulation we presented before. The model takes more time to be rendered as it has more faces than the volumes we have used before. The integration time has grown because of the cloth size: the poncho holds 1242 points, whereas the rectangular cloth held only 441. Dependent on the number of cloth-points is also the handling of collisions with the volume. In this simulation, the task does not seem to need more time to be processed than with the torus or the twisted knot. In fact, if we look at the simulation, the cloth collides mostly with the shoulders and the breast of the model. In the simulations with the volumes, these colliding regions were relatively

larger and might have involved a same number of cloth-points.

For that simulation, we also used an LDI resolution of 64×64 sticks. As we can see, the self-collision handling starts to act when the cloth is beginning to hit the model. Although no self-collision is noticed in the simulation, the few necessary computations are more time-consuming due to the resolution of the LDI. However, it remains under 1 ms whereas it took more than 3 ms when self-collisions occurred during the simulation with the twisted knot (at same LDI resolution).

Regarding the LDI generation, the 3-directional LDI is clearly the most time-greedy task again. In this case, it uses more time by a factor 15 than the time necessary for the self-collision LDI to be generated. This is understandable, regarding the complexity of the mesh of Cally. Compared with the torus and the twisted knot, which needed 20 ms and 23 ms, respectively, Cally needs 66 ms to accomplish that task.

On the overall performance, we see that the simulation with Cally is much lower than in the earlier examples. The frame-rate falls to 15 fps instead of 25 fps. Note also that the complexity for generating the LDI over the mesh of Cally is dependent of her position. This is can be seen from the repeated peeks over the volume LDI generation time evolution curve.

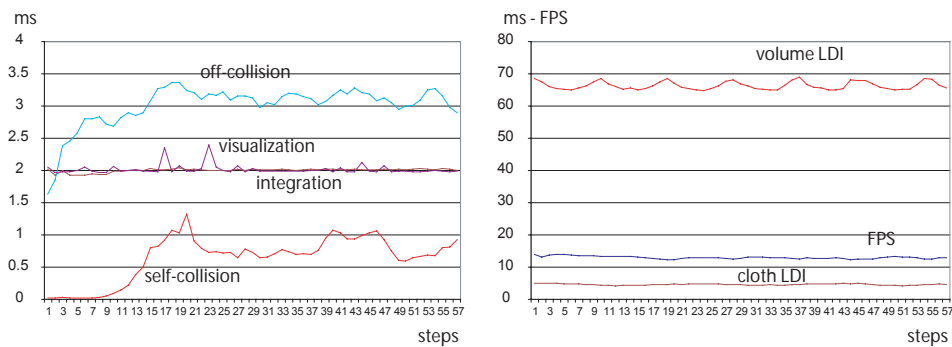


Figure 6.6: Evolution of the processing times of the different tasks involved in the simulation with the walking avatar. We see the part when the cloth is falling free until it collides with the model followed by the regular steps the model makes.

Chapter 7

Future Work

One of the most important further implementation would be to replace the time-greedy *three LDI instances* by the *3-directional LDI*. We have already discussed that modification in Chapter 6 and will not discuss the related improvements that this would bring.

In this chapter we present future approaches to our work. We will show how we could deal with more stable self-collision by considering a new approach based on the *collision detection* and *response* we used for the collision handling of the cloth with the deformable volumes (Section 7.1). We shall then show a way to use multiple instances of clothes and volumes in the simulations and how to deal with the different collisions that would be involved (Section 7.2).

Two further problems are then proposed. Firstly, we describe a way to handle cloth that gets gripped between two colliding volumes (Section 7.3), and then show the problems exposed by using a multiple-LDI structure to handle instabilities like the one produced when the cloth fell on small meshes of the walking avatar (Section 7.4).

7.1 Extensions to Self-collision

In Chapter 5, we have presented an approach to self-collision based on a modified version of the LDI. In contrary to the cloth-to-volume collision process, we used a single LDI and did not consider the short LDI sticks we had developed for the optimization of the 3-directional LDI and the two-side collision detection. Rather, we used the original sticks as a sequence of depths associated with the corresponding triangle indices. Although we got some good results for specific scenarios, the system cannot handle complex self-collisions, nor was its stability satisfying. Complex collisions need mainly to have a tool that detects collision in different directions and the instability is not only related to the complex pipeline the algorithm has to go through, but also, it is hard to detect which triangles are neighbors in

the mesh of the cloth, so as to avoid tearing them apart during the collision handling.

We thought of an approach that might lead to better, or at least different results. Our system for handling collisions with volumes has shown to be highly stable and reliable. The self-collision approach we present here uses most of the aspects we have considered for that collision handling part. The idea is to generate the optimized 3-directional LDI over the mesh of the cloth. The only step we will skip is the *merging* from the *stretching* phase.

We want to perform a similar collision detection and response to the one presented for the volume. The optimized 3-directional LDI generated short LDI-sticks in the direction of the greatest component of the normal of the triangles. In this way, it becomes very unlikely, that a cloth-point finds itself in the stick generated from one of its neighboring triangles. Fig. 7.1 shows how sticks are generated around a fold of a cloth; the folds are precisely where we should be careful when performing self-collision detection to avoid performing the self-collision algorithm. Further more, the sticks are generated along the three X, Y and Z axes, and *side-collisions* will thus be handled.

Suppose the cloth gets folded and two regions are getting close, as illustrated in Fig. 7.2. In these regions, the sticks begin to overlap, but as we skipped the merging phase during the LDI generation, they remain intact. We are now able to detect if a cloth point has penetrated an LDI-stick in which the associated triangle is not one the point belongs to. If this is the case, we perform the usual collision response, considering the cloth-point to strike onto that triangle. The cloth-point will be projected up to the aural distance above the triangle and will remain inside the LDI-stick. This has shown to bring good stability. Reciprocally, the cloth-points belonging to the struck triangle will also fall in the LDI-stick associated with the triangle to which the first considered cloth-point belongs. The response will thus take place for the two colliding regions.

Regarding the implementation, the cloth points can be updated at once as this would not alter the LDI structure and thus, the detection is ensured for both regions. However, both the orientation and the location of the triangles should be stored as they will be affected by the displacement of the cloth-points (their vertices).

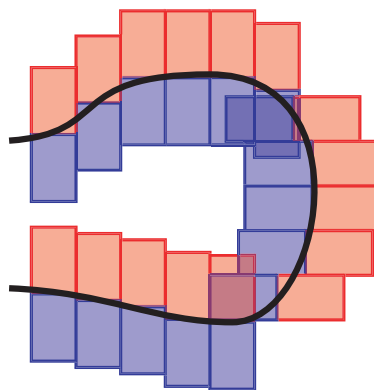


Figure 7.1: Generating small LDI-sticks with the *optimized 3-directional LDI* method ensures that a cloth-point will never find itself in the stick generated from one of its neighboring triangles.

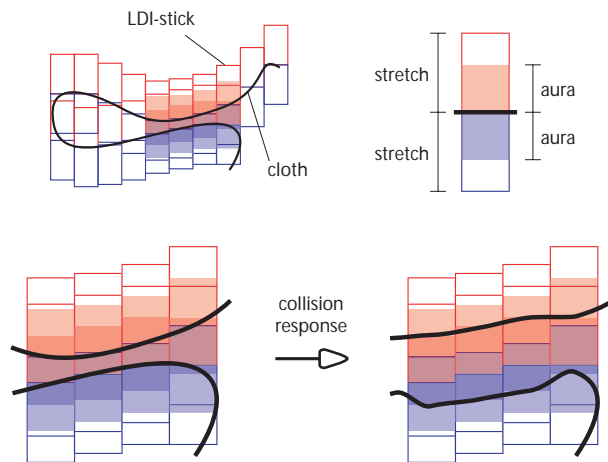


Figure 7.2: We can use the same approach for the collision detection and response as the one we used for collisions with volumes. Colliding triangles fall into the LDI-stick of their colliding opponents and collision handling can be performed for both candidates.

7.2 Multiple Cloth Pieces

In our simulation we have always worked with single instances of cloth and volume. Collision handling is first executed for the cloth with the volume, and secondly for the cloth with itself. If we want to use multiple instances of volumes or clothes it would require some modifications. We suppose we do not want to handle collisions between two volumes, but still have to manage that the collisions of the cloth with different volumes as well as the collisions among the different cloth pieces, are possible.

A demonstration that is often illustrated in papers on cloth simulation like the ones by Mezger, Kimmerle and Etmuss [MKE03, MKE02, EKK⁺01], is the one where multiple instances of cloth fall one over another onto the floor. We show how we should deal with the different collisions and how an optimization could be made.

Handling collisions with the volumes would not require great modifications. We would just consider each cloth mesh one after the other and the

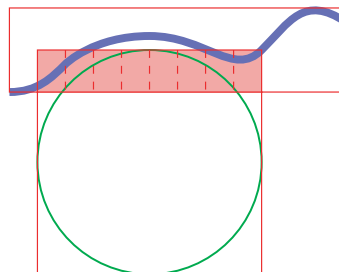


Figure 7.3: An improvement for the cloth-to-volume collision process would be to consider only the cloth-points located within the intersection of the bounding-box of the cloth with the one of the volume.

collision process successively over the different volumes. An improvement could be made in so far as we would not consider all cloth-points of every cloth, but rather only those located within the intersection of the bounding-box of the cloth with the one of the volume, such as in Fig. 7.3. Furthermore, we could also think of generating the 3-directional LDI for the volume only in that intersection which would allow to lower the LDI-resolution even more.

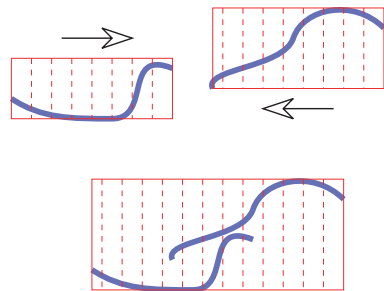


Figure 7.4: With the current implementation of self-collision, if we want to handle collisions between two or more cloth pieces, we must merge their bounding-boxes as soon as they collide and perform the self-collision process as for a single cloth.

Dealing with self-collisions needs more consideration. With the current implementation of the self-collision process, we consider the depths sequences retrieved from the LDI-sticks. The collisions between two different clothes should be treated the same way as they are for a single cloth. The depths sequences should thus include the depths of all colliding cloth pieces. Therefore, it would be necessary to treat different clothes as a single one by generating the LDI over the union of all colliding cloth as soon as their bounding-boxes collide (see Fig. 7.4). The overall bounding-box would thus get much bigger and we might increase the LDI-resolution. However, we have seen that the processing time for self-collision is directly dependent on the LDI-resolution and the simulation could significantly slow down.

If we consider the alternate self-collision procedure we presented in Section 7.1, collisions between different cloth pieces should be handled differently. This new procedure handles self-collisions in the same way we did for collisions with volumes. It would thus not be necessary to handle different cloth pieces as one single cloth. We should consider all cloth-points from a given mesh and check for collisions with all the other instances of volumes, clothes and even with the cloth itself. Considering generating LDIs only inside the intersection of two colliding bounding-boxes of two cloth pieces would, however not be an improvement, as the LDI needs to be generated entirely over each cloth to perform the usual self-collision.

7.3 Colliding Volumes

A common avatar has a mesh divided in several parts. For example, Cally is built out of a collection of small meshes like the chest, the pelvis, the lower and the upper arm and so on, where the animation consists in moving

these meshes after the gestures of an invisible skeleton. These different meshes, however, never get deformed. It happens, thus, that when Cally is kneeling or when we let her arm fall along the chest, these different meshes intersect. This is usually the case with standard avatars, as it would be much more difficult and time consuming to detect these collisions and adapt the geometry of the meshes.

When we simulate trousers or a T-shirt, for example, it becomes important to consider this aspect. In regions such as the armpits or behind the knee, the tissue would get gripped and when the collisions occur, the system must be able to manage the cloth regions concerned to be handled correctly while they came inside the volumes.

Based on an example presented in a recent publication from Baraff, Witkin and Kass [BWK03], we wondered if with our system we could find out, how to handle a cloth between two colliding volumes. Fig. 7.5 illustrates a simple case: a cloth piece is lying onto a sphere and regions are hanging on the sides. A second sphere collides and the cloth must remain stable while being inside the two volumes.

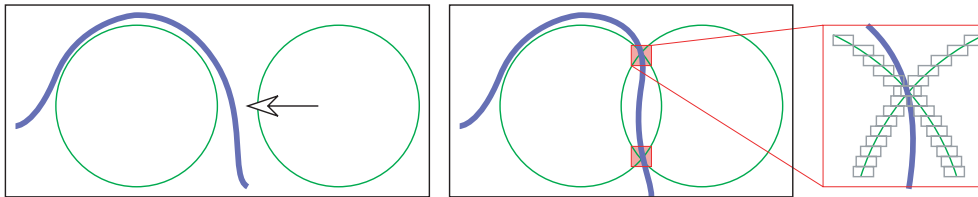


Figure 7.5: Two spheres collide while one has a cloth lying on it. With our two-side collision detection and skipping cloth-points that belong to LDI-sticks of different volumes, it should be possible to allow such collision and still ensure that the system remains stable.

We have seen that our two-side collision detection implied that the volumes are no longer filled with the LDI. Moreover, as we can work with open meshes, there is no notion of inside or outside anymore. Therefore, with the current implementation of our system, it would be absolutely possible for the cloth to be inside the two spheres. The problem resides in how we deal with the tissue in the boundary defined by the intersections of the meshes. This boundary is stressed in Fig. 7.5.

At this boundary, we have a superposition of the sticks of two different volumes. If we launched the simulation with our actual implementation, the triangles in that location would be thrown from one LDI-stick to to other if not inside one of the spheres and outside the other.

An idea would be to skip the collision process on cloth-points that reside in LDI-sticks from two different volumes. The outer part of the cloth would still lie on the first sphere, whereas all the region that belongs to the

intersection would be ignored. Actually, the inside part would always stay in the intersection as the collision process is active on the inner parts of the spheres. As long as the cloth remains inside the two volumes, the system should remain stable.

7.4 Multiple-LDI Structure

For the simulation scenario with the walking avatar, we designed a poncho to fit onto Cally. We refined the extremities as the simulation fails when a cloth piece gets in contact with the hands. As we explained before, this is mainly because the fingers are much smaller than the actual section of the sticks from the LDI generated over the whole mesh.

To handle this problem, we could think of generating a multi-LDI structure as illustrated in Fig. 7.6. This would require to generate more LDIs, which is costly, but the LDI-resolution for each instance could be significantly reduced and good results could be obtained. Recall, however, that even if the hands got an appropriate LDI, the triangles of the cloth might still remain too large for being handled by such small LDI-sticks, as the one generated for these small meshes.

Another approach would be to generate a virtual sphere around the critical meshes and generate the LDI over these, but still render the hands on the screen.

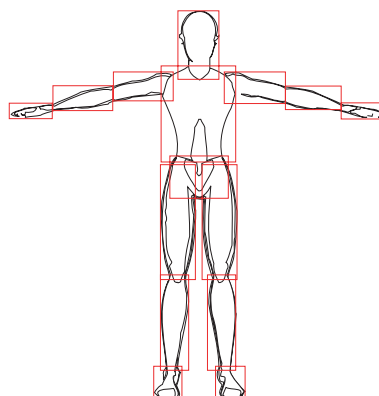


Figure 7.6: Generating a multiple-LDI structure over the mesh of the avatar could resolve some instability in the case where the cloth collides with small meshes like the fingers.

Chapter 8

Conclusions

In this diploma thesis, we have developed a new approach to *collision detection* and *collision response* of cloth onto deformable volumes, along with a *self-collision* algorithm to handle collisions of the cloth with itself. These techniques are based on the new *LDI tool* which can take advantage of the hardware acceleration of the graphic card when meshes become large.

The LDI tool is part of the *Collision Control* library developed at CGL, ETH, Zurich, by Bruno Heidelberger, and was intended to detect intersections of closed colliding bodies in real-time. By modifying this library, we were able to use it for real-time cloth simulations.

We started by implementing a procedure for handling collisions of cloth pieces with deformable volumes and went on to self-collision. These two aspects use the LDIs in different ways but we could generalize our modifications of the library so that it remains compatible for both procedures.

For the collisions involving both cloth pieces and volumes, we obtained great stability, even with large integration time steps. Regarding self-collisions, the procedure still allows real-time simulations but the behavior of the cloth remains unstable as soon as the collisions become too complex. As we have shown throughout this work, the self-collision approach we made can be considerably improved, even though we obtained a high degree of self-collision avoidance for specific simulation scenarios.

The great advantage of using this new tool in our approach is that the collision detection part is fast and reliable even if the LDI has to be generated at each time step. It was thus possible to work with highly deformable volumes without influencing the overall efficiency.

In today's literature, cloth-modeling is widely investigated. Although there exist different approaches that bring much higher stability especially for self-collision handling, few papers present methods that can run in real-time, and self-collision is often recognized as the bottleneck for fast simulations. Moreover, deformable models are not considered when it comes to deal with interactive simulations.

Nevertheless, although the collision detection and response part is resolved, we can push our research much further regarding self-collision. We have also proposed alternatives and optimizations that should bring greater stability and increase the frame-rate of the simulations which remains between 15 to 35 fps. We have also shown that the LDI approach for resolving real-time cloth modeling with deformable volumes has a great potential to become very efficient.

Acknowledgment

I am indebted to Dr. Matthias Teschner, CGL Computer Graphics Laboratory at ETH Zurich, for having proposed the topic and for his constant interest and advice during the preparation of the present work. I am grateful to Professor Dr. Markus Gross, Director of CGL, for the kind reception at his laboratory, thus enabling me to carry out my diploma work at ETH Zurich. Last but not least, I would like to thank Professor Dr. Sabine Süsstrunk, LCAV Audiovisual Communications Laboratory, EPF Lausanne, for kindly having accepted the supervision of this work.

Bibliography

- [BFA] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation.
- [Bri03] Robert Edward Bridson. Computational Aspects of Dynamic Surfaces. PhD thesis, Stanford University, May 2003.
- [BW92] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. Computer Graphics, 26(2):303–308, 1992.
- [BW98] David Baraff and Andrew Witkin. Large steps in cloth simulation. Computer Graphics, 32(Annual Conference Series):43–54, 1998.
- [BWK03] David Baraff, Andrew Witkin, and Michael Kass. Untangling cloth. In Proceedings of ACM SIGGRAPH 2003, editor, ACM Transactions on Graphics, volume 22, pages 862–870, 2003.
- [CK] Kwang-Jin Choi and Hyeong-Seok Ko. Stable but responsive cloth.
- [CMT02] Frederic Cordier and Nadia Magnenat-Thalmann. Real-time Animation of Dressed Virtual Humans. EUROGRAPHICS 2003, 21(3), 2002.
- [DMB] Mathieu Desbrun, Mark Meyer, and Alan H. Barr. Interactive animation of cloth-like objects for virtual reality.
- [EKK⁺01] Olaf Etzmuß, Michael Keckeisen, Stefan Kimmerle, Johannes Mezger, Michael Hauth, and Markus Wacker. A Cloth Modelling System for Animated Characters. In Proceedings Graphiktag, 2001.
- [FGL03] Arnulph Fuhrmann, Clemens Gross, and Volker Luckas. Interactive Animation of Cloth including Self Collision Detection. WSCG, 11(1), February 2003.

- [Hec97] Chris Hecker. Collision response. Game Developer, pages 11–18, March 1997.
- [HTG03] Bruno Heidelberger, Matthias Teschner, and Markus Gross. Volumetric Collision Detection for Deformable Objects. Technical Report 395, Computer Science Department, ETH Zurich, Switzerland, April 2003.
- [KHM⁺98] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k -DOPs. IEEE Transactions on Visualization and Computer Graphics, 4(1):21–36, /1998.
- [Mic] Olaf Etmub Michael. A cloth modelling system for animated characters.
- [MKE02] Johannes Mezger, Stefan Kimmerle, and Olaf Etmu . Improved Collision Detection and Response Techniques for Cloth Animation. WSI-2002-5, 2002.
- [MKE03] Johannes Mezger, Stefan Kimmerle, and Olaf Etmu . Hierarchical Techniques in Collision Detection for Cloth Animation. Journal of WSCG, 11(2):322–329, 2003.
- [VCT95] Pascal Volino, Martin Courchesne, and Nadia Magnenat Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. Computer Graphics, 29(Annual Conference Series):137–144, 1995.
- [VM95] Pascal Volino and Nadia Magnenat Thalmann. Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces. In Dimitri Terzopoulos and Daniel Thalmann, editors, Computer Animation and Simulation '95, pages 55–65. Springer-Verlag, 1995.
- [VSC01] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. Eurographics 2001, 20(3), September 2001.
- [VT] Pascal Volino and Nadia Magnenat Thalmann. Implementing Fast Cloth Simulation with Collision Response. MIRALab, C.U.I., University of Geneva, Switzerland.
- [VT94] Pascal Volino and Nadia Magnenat Thalmann. Efficient Self-Collision Detection on Smoothly Discretized Surface Animations using Geometrical Shape Regularity. Computer Graphics Forum, 13(3):155–166, 1994.

Appendix A

The Libraries

Our cloth simulation application uses different libraries. For the GUI and for handling the volume entities we used OpenInventor™ . The cloth integration part is done by the *msm* library developed by Matthias Teschner from the CGL, ETH, Zurich. For the simulations with the walking avatar, we made recourse to the open source library Cal3d. Finally, as it is also the library on which our work was based, the *Collision Control* library was also integrated in our system.

We give here a short description of these different libraries.

OpenInventor: 3D Object-Oriented Toolkit

From the SGI web-site: “Open Inventor™ is an object-oriented 3D toolkit offering a comprehensive solution to interactive graphics programming problems. It presents a programming model based on a 3D scene database that dramatically simplifies graphics programming. It includes a rich set of objects such as cubes, polygons, text, materials, cameras, lights, track-balls, handle boxes, 3D viewers, and editors that speed up your programming time and extend your 3D programming capabilities.”

OpenInventor™ is well suited for our application as it is built on top of OpenGL which is used by the collision detection library. OpenGL is widely used in the industry as a 2D and 3D graphics programming interface. With OpenInventor™ we easily get a graphical user interface that allows to interact with the scene. Also, through call-back functions, it is possible to add functionalities or to modify parameters. Fig. A.1 gives an overview of a basic OpenInventor™ GUI.

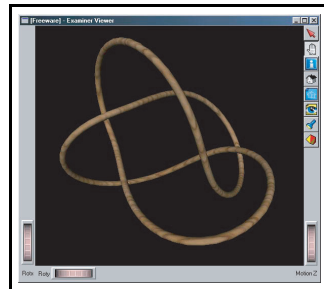


Figure A.1: The OpenInventor basic GUI

msm: Integration Library

The msm library holds the integration methods and the data-structures needed for the implementation of the cloth simulation.

The atomic class of the data-structure is *msmPoint* on which are based *msmTriangle*, *msmTet*, *msmVector*, *msmMesh* and *msmSpring*. The last class is the mass-spring element for the cloth simulation. Beside of these classes there is also the *msmParam* class which holds side-parameters such as the colors of a mesh, its physical properties, etc.

Fig. A.2 illustrates four steps of the simulation of a cloth colliding with a sphere.

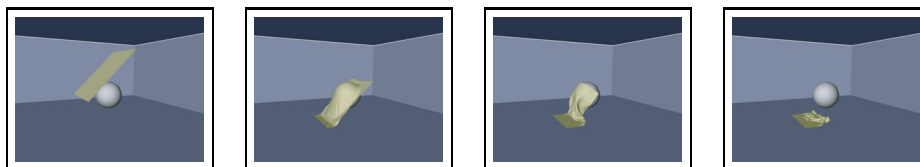


Figure A.2: Cloth simulation with the msm library

Cal3D: 3D Character Animation Library

Cal3D is a 3D character animation library written in C++ in a platform/graphic-API independent way. The core feature of the Cal3D library is a skeletal-based approach on top of which there is a control system that handles the sequence and blending of the animations. While the control system updates the state of the body, it is easy to access the mesh of the body and to use it for the collision detection.

Besides, the library comes with three different characters to animate: Cally herself, a skeleton and a fully textured character. These characters are illustrated in Fig. A.3.

For more information on the Cal3D library, you may visit the web-site at the following URL: <http://cal3d.sourceforge.net>

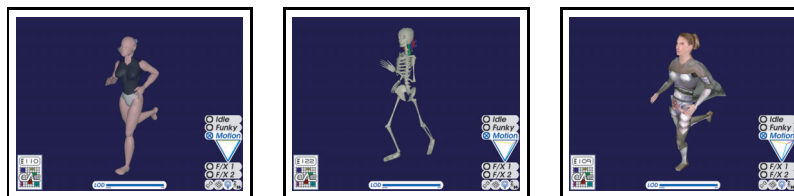


Figure A.3: The three different characters offered by the Cal3D library

Collision Detection Library

The collision detection process developed at the CGL is based on a “Layered Depth Image” (LDI) decomposition of volumes. This is a decomposition into parallel “sticks” of constant square section. The library allows to get the LDI of any closed volume. There then only needs to make a query to know whether a given point is located inside the LDI. The library also generates the LDI for the intersection of two volumes. This is represented in Fig. A.4.

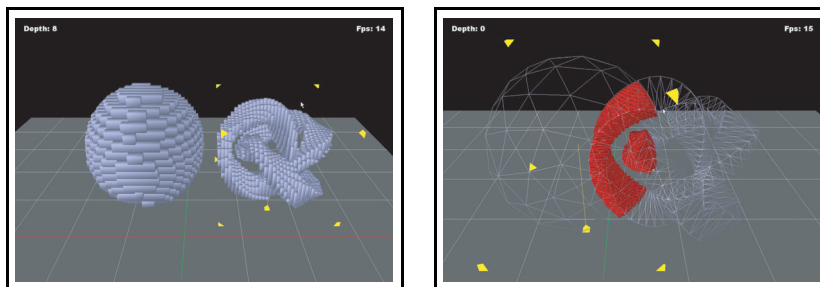


Figure A.4: The LDI representation of two volumes and their intersection

Appendix B

Measurements

The graphs we used in the analysis of our experiments are based on the measurements given in the following tables. FPS denotes the frame-per-second rate, all other units are in milliseconds (ms).

The significance of the columns headers are:

FPS: $[f/s]$ frame-per-second rate achieved;

iter: $[\#]$ number of iterations executed between two consecutive measurements (1 measurement every second);

integr: $[ms]$ time needed for the cloth integration process;

cub-coll: $[ms]$ time needed for the collision detection and response between the cloth and the cuboid;

self-coll: $[ms]$ time needed for the self-collision detection and response of the cloth;

off-coll: $[ms]$ time needed for the collision detection and response between the volume and the cloth;

self coord: $[ms]$ necessary time to transmit the coordinates of the cloth from the OpenInventor structure to the LDI structure;

self LDI: $[ms]$ generation time for the self-collision LDI;

mv model: $[ms]$ necessary time to transform the volume or Cally;

vol coord: $[ms]$ necessary time to transmit the coordinates of the volume from the OpenInventor structure to the LDI structure;

vol LDI: $[ms]$ generation time for the LDI of the volume;

visu: $[ms]$ visualization rendering time.

Torus with 32×32 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
28.996	30	1.043	0.0466	0.0178	0.434	0.0266	2.289	0.0680	0.0659	10.713	1.402
27.610	29	1.049	0.0469	0.0151	1.094	0.0286	2.076	0.0674	0.0644	10.592	1.371
24.436	25	1.022	0.0462	0.0404	2.362	0.0336	1.905	0.0607	0.0674	10.612	1.360
24.686	25	1.032	0.0470	0.3637	1.932	0.0327	2.060	0.0623	0.0640	10.562	1.368
24.425	26	1.030	0.0476	0.5543	1.542	0.0302	2.128	0.0640	0.0640	10.598	1.374
26.666	27	1.031	0.0495	0.5026	0.781	0.0277	2.094	0.0678	0.0655	10.593	1.368
27.809	29	1.032	0.0504	0.4414	0.377	0.0267	2.067	0.0694	0.0638	10.579	1.370
28.822	29	1.041	0.0524	0.4097	0.296	0.0267	2.007	0.0691	0.0632	10.601	1.372
28.407	30	1.024	0.0522	0.3856	0.213	0.0271	2.001	0.0701	0.0641	10.625	1.361
29.606	30	1.034	0.0535	0.2798	0.173	0.0267	1.945	0.0690	0.0637	10.576	1.364
29.156	31	1.027	0.0538	0.2208	0.161	0.0268	1.877	0.0699	0.063	10.62	1.370
29.889	31	1.027	0.0556	0.1435	0.160	0.0267	1.807	0.0700	0.0651	10.593	1.364
30.265	31	1.032	0.0569	0.0745	0.159	0.0260	1.764	0.0710	0.063	10.653	1.365
30.973	31	1.009	0.0607	0.0255	0.159	0.0259	1.744	0.0697	0.0668	10.621	1.354
30.330	32	1.008	0.0619	0.0115	0.159	0.0270	1.621	0.0692	0.0643	10.607	1.354
30.636	32	1.005	0.0632	0.0115	0.160	0.0257	1.403	0.0666	0.0633	10.634	1.352
30.65	32	1.014	0.0642	0.0115	0.161	0.0258	1.359	0.0665	0.0632	10.580	1.361

Torus with 64×64 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
24.308	25	0.950	0.0426	0.0353	0.397	0.0268	4.309	0.0680	0.0650	18.757	1.366
23.294	25	0.928	0.0433	0.0320	0.797	0.028	4.122	0.0658	0.0651	18.768	1.371
21.387	23	0.951	0.0420	0.0222	2.138	0.0326	3.901	0.0590	0.065	18.839	1.365
20.651	22	0.966	0.0417	0.6125	2.179	0.0342	3.672	0.057	0.0639	18.759	1.372
19.549	21	0.967	0.0419	1.6671	1.827	0.0319	3.957	0.0580	0.0635	18.766	1.415
19.943	20	0.980	0.0427	2.1644	1.450	0.0311	4.041	0.0599	0.0640	18.769	1.411
20.309	22	0.965	0.0446	1.7537	0.738	0.0275	4.003	0.0628	0.0646	18.714	1.410
21.789	22	0.958	0.0446	1.5491	0.432	0.0267	4.003	0.0683	0.0646	18.763	1.422
22.601	23	0.965	0.0451	1.2072	0.316	0.0266	3.727	0.0658	0.0649	18.729	1.423
22.870	23	0.958	0.0468	1.3704	0.248	0.0267	3.667	0.0656	0.0659	18.770	1.415
22.807	24	0.957	0.0469	0.9911	0.198	0.0268	3.535	0.0669	0.0652	18.795	1.402
23.892	24	0.949	0.0474	0.7245	0.177	0.0267	3.457	0.0681	0.0652	18.764	1.382
23.690	25	0.958	0.0481	0.5513	0.165	0.0272	3.387	0.0684	0.0662	18.786	1.374
24.220	25	0.958	0.0499	0.4258	0.164	0.0272	3.264	0.0685	0.0652	18.832	1.369
24.295	26	0.943	0.0507	0.2175	0.162	0.0270	3.235	0.067	0.0648	18.731	1.408
24.591	26	0.940	0.0532	0.1155	0.161	0.0266	3.105	0.0679	0.0644	18.971	1.367
25.252	26	0.946	0.0560	0.0261	0.161	0.0262	3.041	0.0683	0.0649	18.766	1.365
25.338	26	0.942	0.0553	0.0221	0.160	0.0265	2.923	0.0682	0.0650	18.83	1.368
23.482	25	0.913	0.0565	0.0217	0.160	0.0262	6.633	0.0684	0.0656	18.835	1.361

Torus with 128×128 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
16.896	18	0.937	0.0420	0.0815	0.3691	0.0274	11.082	0.0698	0.0726	42.568	1.328
16.745	18	0.943	0.0414	0.0773	0.5859	0.0282	10.793	0.0700	0.0676	42.592	1.330
16.291	18	0.940	0.0419	0.0712	1.0157	0.0300	10.722	0.0701	0.0731	42.888	1.334
15.697	17	0.987	0.0421	0.0605	2.3394	0.0466	9.9444	0.0675	0.0662	42.761	1.343
15.076	17	0.981	0.0408	1.2909	2.3264	0.0455	8.71	0.0674	0.0657	42.620	1.358
13.652	15	1.005	0.0409	4.9713	2.0570	0.0462	9.3366	0.0663	0.0651	42.401	1.391
13.128	14	0.993	0.0414	7.0673	1.8696	0.0427	9.5044	0.0702	0.0669	42.489	1.390
12.500	13	1.004	0.0416	9.3050	1.7999	0.0426	10.248	0.0666	0.0657	42.513	1.442
12.522	13	0.984	0.0420	9.2895	1.3375	0.0363	10.042	0.0677	0.0687	42.570	1.390
13.492	14	0.991	0.0432	6.6169	0.9083	0.0327	9.5686	0.067	0.066	44.794	1.393
13.892	14	0.988	0.0443	6.7141	0.5876	0.0302	10.061	0.0686	0.0655	42.595	1.384
13.964	15	0.988	0.0442	5.4586	0.4187	0.0286	9.4878	0.068	0.0654	42.469	1.386
14.249	16	1.000	0.0446	4.3639	0.3570	0.0281	8.8591	0.0685	0.0658	42.677	1.383
14.514	16	0.976	0.0456	4.3358	0.3367	0.0278	8.8504	0.0686	0.0676	42.510	1.446
14.171	16	0.986	0.0461	4.6710	0.2989	0.0278	8.6646	0.069	0.0662	42.557	1.389
14.745	16	0.987	0.0470	3.9713	0.2497	0.0278	8.4340	0.0696	0.0658	42.559	1.381
15.435	16	0.995	0.0477	3.3315	0.2245	0.0275	8.3976	0.068	0.0654	42.506	1.380
15.459	17	0.977	0.0472	2.6007	0.2090	0.0276	8.0935	0.072	0.0677	42.921	1.384
15.775	17	0.977	0.048	2.3710	0.2034	0.0288	8.0998	0.0695	0.0658	42.614	1.394
16.109	17	0.979	0.0481	2.0123	0.1986	0.0282	7.9292	0.0701	0.0662	42.865	1.382
16.991	17	0.990	0.0491	1.3900	0.1930	0.0302	7.6250	0.0697	0.0659	42.570	1.373
16.647	17	0.950	0.0505	0.8393	0.1836	0.0278	7.5148	0.0695	0.0667	45.402	1.345
17.463	18	0.938	0.0519	0.5335	0.1827	0.0273	7.4144	0.1148	0.0683	42.585	1.324
17.708	18	0.973	0.0537	0.3178	0.1802	0.0275	7.5285	0.0702	0.0656	42.627	1.330
17.231	19	0.941	0.0549	0.1267	0.1783	0.0271	7.2517	0.0691	0.0658	42.662	1.321
17.330	19	0.942	0.0547	0.0759	0.1758	0.0271	6.7852	0.0701	0.0671	43.029	1.333
17.611	19	0.929	0.0544	0.0601	0.1842	0.0284	6.3795	0.0697	0.0664	42.623	1.326
16.713	18	0.932	0.0560	0.0602	0.1737	0.0271	17.067	0.0683	0.066	42.898	1.328

Twist with 32×32 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
28.239	29	0.920	0.0436	0.0192	0.3661	0.0273	2.2562	0.0854	0.0737	13.062	1.398
27.518	28	0.901	0.0434	0.0177	0.8160	0.0283	2.0579	0.0847	0.0753	13.042	1.380
25.560	26	0.916	0.0428	0.0145	1.5777	0.0323	1.9162	0.0774	0.0743	13.012	1.385
24.315	26	0.951	0.0432	0.2152	1.5273	0.0329	1.9311	0.0735	0.0716	13.108	1.393
24.079	26	0.941	0.0445	0.5951	1.2167	0.0316	2.1908	0.076	0.0748	13.037	1.397
24.913	25	0.947	0.0448	0.8042	0.9847	0.0315	2.3915	0.0759	0.0724	13.030	1.390
24.162	26	0.949	0.0457	0.8931	0.8069	0.0313	2.3587	0.0782	0.0724	13.003	1.392
25.774	26	0.946	0.0461	0.7465	0.5817	0.0309	2.2040	0.0801	0.0736	13.052	1.388
26.506	27	0.944	0.0482	0.6070	0.4021	0.0310	2.1365	0.0823	0.0739	13.019	1.401
26.820	28	0.940	0.0462	0.5213	0.2628	0.0280	2.0493	0.0827	0.0725	13.072	1.388
26.300	28	0.944	0.0478	0.5816	0.2133	0.0295	2.0381	0.0836	0.0732	13.063	1.394
27.723	29	0.934	0.0487	0.2899	0.2046	0.0283	1.9310	0.0829	0.0737	13.062	1.384
28.354	29	0.929	0.0487	0.2169	0.2010	0.0279	1.8758	0.0846	0.0724	13.044	1.397
28.376	29	0.935	0.0495	0.2073	0.2012	0.0275	1.8892	0.0846	0.0731	13.006	1.411
28.375	29	0.930	0.0517	0.2014	0.2033	0.0276	1.8506	0.0843	0.0741	13.105	1.389
28.728	29	0.935	0.0515	0.1918	0.2016	0.0278	1.6839	0.0854	0.0725	13.024	1.386
28.263	29	0.972	0.0524	0.1885	0.2007	0.0287	1.6969	0.0842	0.0725	13.079	1.412
28.575	29	0.932	0.0518	0.1980	0.2014	0.0277	1.7761	0.0843	0.0732	13.100	1.391
28.646	29	0.932	0.0571	0.1941	0.2007	0.0278	1.8090	0.0844	0.0723	13.019	1.388
28.638	29	0.929	0.0503	0.1862	0.2008	0.0275	1.7644	0.0844	0.0748	13.069	1.410
28.393	29	0.949	0.0500	0.1857	0.1999	0.0275	1.8946	0.0844	0.0725	13.040	1.394
28.804	29	0.929	0.0516	0.1739	0.1986	0.0273	1.7555	0.0847	0.0723	13.041	1.384

Twist with 64×64 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
22.889	23	1.014	0.0459	0.0338	0.3475	0.0269	4.3684	0.0802	0.0800	23.030	1.394
22.703	23	1.026	0.0446	0.0331	0.6485	0.0279	4.0837	0.0812	0.0724	22.660	1.386
21.166	23	1.001	0.0452	0.0246	1.2914	0.0305	3.9865	0.0786	0.0725	22.711	1.376
20.850	22	1.021	0.0441	0.1112	1.7101	0.0335	3.514	0.0711	0.0719	22.703	1.412
20.058	21	1.037	0.0447	1.0636	1.5226	0.0337	3.6568	0.0741	0.0720	22.722	1.381
18.856	20	1.038	0.0461	2.1539	1.2382	0.0308	4.2093	0.0728	0.0722	22.671	1.454
18.820	19	1.055	0.0463	2.7675	1.0724	0.0302	4.6223	0.0758	0.0720	22.644	1.390
18.131	19	1.055	0.0466	3.2017	0.9488	0.0299	4.7096	0.0753	0.0729	22.735	1.391
18.396	19	1.049	0.0471	3.1159	0.8054	0.0291	4.4304	0.0774	0.0718	22.626	1.395
18.162	20	1.037	0.0478	2.5740	0.6155	0.0285	4.0308	0.0795	0.0727	24.005	1.409
19.388	21	1.039	0.0476	2.1197	0.4686	0.0279	3.7578	0.0808	0.0725	22.668	1.425
20.803	21	1.038	0.0476	1.6532	0.3467	0.0274	3.6507	0.0818	0.0728	22.727	1.427
20.939	22	1.042	0.0484	1.3552	0.2566	0.0274	3.5143	0.0817	0.0753	22.735	1.398
21.081	22	1.090	0.0499	1.2295	0.2169	0.0275	3.5743	0.0827	0.0730	23.588	1.390
21.211	23	1.033	0.0518	0.9827	0.2060	0.0277	3.4950	0.0833	0.0746	22.71	1.397
21.819	23	1.019	0.0598	0.807	0.2038	0.0279	3.3424	0.0838	0.0737	22.659	1.383
22.400	23	1.000	0.0535	0.6612	0.2022	0.0274	3.1441	0.0869	0.0745	22.657	1.377
22.890	23	1.007	0.0548	0.5634	0.2004	0.0273	3.059	0.0841	0.0745	22.636	1.406
22.945	23	0.994	0.0545	0.5756	0.2001	0.0273	2.9431	0.0839	0.0747	22.933	1.410
22.931	23	0.993	0.0539	0.5693	0.2014	0.0273	3.0237	0.0838	0.0744	22.705	1.378

Twist with 128×128 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
15.547	17	0.993	0.0413	0.0814	0.3129	0.0265	11.153	0.0844	0.0744	51.563	1.367
15.444	17	1.015	0.0420	0.0780	0.4981	0.0276	10.804	0.0845	0.0742	51.620	1.377
15.181	17	1.000	0.0421	0.0750	0.8240	0.0282	10.750	0.0832	0.0762	51.457	1.409
15.388	16	1.019	0.0417	0.0633	1.3931	0.033	10.498	0.0800	0.0745	52.266	1.388
15.321	16	1.036	0.0415	0.1142	1.7676	0.0403	8.8953	0.0788	0.074	51.649	1.381
14.315	16	1.040	0.0420	1.6740	1.7891	0.0414	8.6766	0.0786	0.0742	51.596	1.392
13.543	14	1.035	0.0431	5.3884	1.5145	0.0377	9.0818	0.0818	0.0755	51.620	1.448
12.531	13	1.028	0.0426	7.8464	1.3759	0.0369	10.456	0.0780	0.0740	51.605	1.391
11.759	13	1.038	0.0432	9.3740	1.2500	0.0360	10.961	0.0804	0.0742	51.558	1.401
11.940	12	1.019	0.0437	10.727	1.1651	0.0345	11.477	0.0805	0.0743	51.688	1.380
11.131	12	1.036	0.0440	12.237	1.0710	0.0341	11.564	0.0800	0.0786	51.587	1.383
11.236	12	1.034	0.0590	12.464	0.9842	0.0328	11.438	0.0801	0.0744	51.472	1.399
11.437	12	1.033	0.0447	11.843	0.8902	0.0317	10.845	0.081	0.0747	52.130	1.457
11.217	13	1.032	0.0449	10.698	0.7826	0.0306	10.175	0.0819	0.0754	51.722	1.401
11.992	13	1.033	0.0450	9.3090	0.6711	0.0317	9.569	0.0807	0.0746	51.939	1.390
12.679	13	1.041	0.0448	8.1154	0.5427	0.0287	9.1697	0.0800	0.0740	51.796	1.395
12.664	13	1.027	0.0450	7.7706	0.4564	0.0284	8.8883	0.0807	0.0741	51.806	1.385
13.376	14	1.033	0.0451	6.0947	0.3863	0.0277	8.5727	0.0827	0.0747	51.822	1.389
13.781	14	1.019	0.0451	5.4687	0.3042	0.0277	8.4276	0.0799	0.0749	51.519	1.384
13.473	15	1.026	0.0463	4.8793	0.2642	0.0271	8.3041	0.0810	0.0816	51.660	1.379
13.831	15	1.025	0.0475	4.4088	0.2494	0.0284	8.3430	0.0833	0.0747	51.625	1.386
13.964	15	1.032	0.0477	4.1236	0.2346	0.0277	8.4844	0.0815	0.0746	51.681	1.401
14.378	15	1.026	0.0484	3.7139	0.2328	0.0274	8.3344	0.0817	0.0746	51.625	1.391
14.981	15	1.097	0.0497	2.9300	0.2297	0.0271	7.7986	0.0828	0.0744	51.690	1.383
14.638	16	1.022	0.0505	2.4939	0.2250	0.0282	7.6143	0.0830	0.0748	51.601	1.378
15.013	16	1.044	0.0528	2.4212	0.2372	0.0268	9.5555	0.0834	0.0743	51.817	1.614

Cally with 64×64 LDI resolution:

FPS	iter	integr	cub-coll	self-coll	off-coll	self coord	self LDI	mv model	vol coord	vol LDI	visu
13.942	14	2.015	0.1005	0.0234	1.6370	0.0617	5.0261	0.2153	0.7052	68.540	2.053
13.187	15	1.924	0.1010	0.0232	1.8492	0.0611	5.0074	0.2144	0.7036	67.441	1.967
13.818	14	2.004	0.1012	0.0286	2.3815	0.0626	4.945	0.2150	0.7007	66.132	1.978
13.895	14	1.929	0.0989	0.0235	2.4595	0.0686	4.9062	0.2165	0.6982	65.471	1.976
13.859	14	1.928	0.1000	0.0238	2.5848	0.0644	4.8572	0.221	0.6988	65.296	1.996
13.733	14	1.926	0.0999	0.0235	2.7993	0.0673	4.8002	0.221	0.6992	65.036	2.055
13.626	14	1.949	0.0983	0.0234	2.8036	0.0646	4.7435	0.2225	0.7005	65.602	1.992
13.558	14	1.939	0.0978	0.0329	2.8345	0.0700	4.6035	0.2212	0.6996	66.309	1.970
13.477	14	1.934	0.0986	0.0567	2.7208	0.0682	4.4990	0.2247	0.7017	67.558	1.964
13.287	14	1.990	0.0998	0.0966	2.6882	0.0668	4.4498	0.2279	0.7027	68.627	2.064
13.399	14	1.988	0.0992	0.1411	2.8222	0.0695	4.2732	0.2322	0.701	66.897	1.993
13.424	14	2.000	0.0998	0.2184	2.8952	0.0705	4.234	0.2356	0.7009	66.024	1.998
13.416	14	2.009	0.0990	0.3821	2.8544	0.0684	4.4602	0.2359	0.6973	65.256	2.008
13.268	14	2.000	0.0991	0.5040	2.8976	0.0674	4.3627	0.2338	0.6983	65.548	1.990
13.047	14	2.034	0.0997	0.8004	3.0845	0.0718	4.4767	0.2362	0.6987	65.014	1.986
12.873	14	2.014	0.0980	0.8279	3.2750	0.071	4.3365	0.2374	0.6977	65.364	1.975
12.643	14	2.013	0.098	0.9148	3.2935	0.072	4.4977	0.2295	0.699	66.256	2.355
12.418	14	2.020	0.0979	1.0722	3.3597	0.0777	4.629	0.2237	0.6993	67.585	1.982
12.204	14	2.037	0.1011	1.0312	3.3657	0.0756	4.6128	0.2140	0.6998	68.630	2.072
12.349	14	2.014	0.0982	1.3243	3.2411	0.0711	4.6882	0.2219	0.708	67.014	1.988
12.668	14	2.018	0.0990	0.9047	3.2039	0.0755	4.6417	0.2211	0.7037	65.837	1.990
12.902	14	2.008	0.0987	0.7865	3.1058	0.0706	4.758	0.2204	0.6985	65.399	2.033
12.954	14	2.004	0.0990	0.7258	3.1823	0.0710	4.7895	0.2267	0.6985	64.972	2.395
12.960	14	2.000	0.0990	0.7371	3.1687	0.0727	4.8215	0.2265	0.7042	64.844	2.053
12.886	14	2.000	0.0992	0.7230	3.2199	0.0713	4.8820	0.2318	0.6979	65.407	2.005
12.877	14	2.004	0.0988	0.7320	3.0962	0.0723	4.7242	0.2274	0.7795	66.165	1.978
12.724	14	2.020	0.1003	0.6481	3.1593	0.0715	4.6644	0.2297	0.7037	67.675	2.069
12.589	14	2.000	0.1150	0.7846	3.1530	0.0712	4.5357	0.2327	0.7001	68.036	1.983
12.708	14	2.022	0.1013	0.7289	3.1273	0.0685	4.6078	0.2392	0.7039	66.891	2.035
13.021	14	2.008	0.0980	0.6467	2.9841	0.0677	4.4762	0.2367	0.701	66.186	1.987
13.089	14	2.011	0.0969	0.6592	3.0488	0.0663	4.4682	0.2374	0.7022	65.353	1.988
13.126	14	2.007	0.0982	0.7107	3.0210	0.0672	4.4283	0.2352	0.7080	65.176	1.984
13.014	14	2.016	0.0979	0.7684	3.1422	0.0708	4.4837	0.2338	0.7229	64.968	1.989
12.971	14	2.011	0.0978	0.7379	3.1928	0.0707	4.3936	0.2365	0.6997	65.093	1.986
12.881	14	2.004	0.0979	0.6965	3.1919	0.0782	4.3852	0.2276	0.6982	66.404	1.986
12.698	14	2.008	0.0980	0.7040	3.1470	0.0693	4.499	0.219	0.7025	68.076	1.993
12.520	14	2.020	0.0997	0.7025	3.1111	0.0698	4.6113	0.2140	0.7006	69.011	1.994
12.832	14	2.006	0.0981	0.7562	3.0200	0.0678	4.704	0.2246	0.7042	66.714	2.004
12.812	14	2.018	0.0986	0.9454	3.0730	0.069	4.8055	0.2245	0.7153	65.817	2.032
12.689	14	2.021	0.0987	1.0767	3.1697	0.0717	4.8654	0.2232	0.6992	65.531	1.982
12.710	14	2.008	0.0988	1.0355	3.2485	0.0747	4.8321	0.2272	0.7027	65.044	2.039
12.838	14	2.011	0.0975	0.9345	3.1854	0.0722	4.8799	0.2284	0.6985	64.967	1.983
12.700	14	2.023	0.0980	0.9424	3.2764	0.0762	4.9292	0.2261	0.7794	65.419	1.983
12.247	14	2.015	0.0985	0.9902	3.2032	0.0715	4.8867	0.2335	0.7002	68.033	2.126
12.426	14	2.008	0.0982	1.0316	3.1867	0.0697	4.9160	0.2295	0.7337	67.887	1.990
12.42	14	2.012	0.1006	1.0672	3.0874	0.0742	4.7869	0.2313	0.6994	67.985	1.979
12.595	14	2.019	0.0998	0.9290	3.1263	0.0694	4.5232	0.2373	0.7017	66.847	2.069
12.912	14	2.013	0.0979	0.7516	3.0483	0.0718	4.4357	0.2369	0.7022	65.875	1.984
13.147	14	2.020	0.0980	0.5998	2.9477	0.0706	4.2778	0.2373	0.7006	65.454	1.987
13.252	14	2.012	0.0988	0.5937	2.9977	0.0668	4.2807	0.2378	0.7004	65.019	1.987
13.175	14	2.016	0.0992	0.6495	3.0152	0.0681	4.2337	0.2395	0.7004	65.131	1.979
13.114	14	2.029	0.0998	0.6659	3.0953	0.0702	4.3102	0.2359	0.6975	65.292	1.977
12.821	14	2.017	0.098	0.6900	3.2450	0.0712	4.3205	0.2292	0.6989	66.607	1.993
12.583	14	2.007	0.0983	0.6747	3.2708	0.0707	4.4985	0.2157	0.7038	68.493	1.997
12.471	14	2.033	0.0983	0.7996	3.1594	0.0697	4.6027	0.2159	0.6967	68.357	1.986
12.817	14	2.021	0.0993	0.8111	2.9749	0.0711	4.7142	0.2187	0.7007	66.506	1.981
12.886	14	2.004	0.0997	0.9224	2.897	0.0667	4.6774	0.5703	0.7009	65.707	1.988

Appendix C

Code Description

This appendix presents the practical aspects of our application. We present here a sample file used to store the parameters for our simulations and then we show the class diagram of our application and the name of the source files associated with the different classes.

Parameter File

file: **params.cfg**:

```
MODEL poncho.cloth -30 -20 15 2 # name, translation, scaling
CHARACTER data/cally.cfg 0 0 # file , xrot, yrot
TIMESTEP 0.005 # time step 0.003
INTEGRATION Beeman # Leapfrog, RK2, RK4, Heun, Euler, RK2A, RK4A, Verlet,
# Beeman, Theta 0.5 2, Gear 5, ImplicitEuler 20,
# ImplicitTheta 0.5 20
INTSTEPS 5 # numerical integration steps per frame
COLSTEPS 5 # collision detection tests per frame
CUBE 4.0 4.0 # cuboid: Scale Factors (x, y) z
VIEW 600 600 # view size
BACKGROUND 0.1 0.1 0.2 # background color

WEIGHTH 3 # amplitude of the off body moving
OFFINCR 4 # frequency of the off body moving

OFF_LIFT 0

OFF_DAMPING 1 # damping on the off body (0: NO FRICTION, 1: MAX)
OFF_FRICTION 0.9 # friction on the off body (0: NO FRICTION, 1: MAX)
OFF_AURA 1 # distance mantained between cloth and volume
OFF_ROT_X 0 # (0: OFF, 1: ON)
OFF_ROT_Y 0 # (0: OFF, 1: ON)
OFF_ROT_Z 1 # (0: OFF, 1: ON)
OFF_DEFORM 5 # 0:NONE 1:CLOCK 2:WINGS 3:GROW 4:BRUNO 5:DRUM

EPSILON 1 # "security distance" for self-collision
```

```

SAVESCREENFLAG 0          # (0: OFF, 1:ON)
SAVESCREENFILENAME img    # will be saved in 'mov' directory
SAVESCREENWINDOWID 0x2c0004a # to be updated every session...
FRICTION1 0.3            # friction (0: NO FRICTION, 1: MAX)
FRICTION2 0.0            # friction on sphere surface (0: NO FRICTION, 1: MAX)
FASTFORCE 0              # FLAG: FastForce (0: OFF, 1: ON)
SPRINGDAMPING 0          # FLAG: Spring Damping (0: OFF, 1: ON)
POINTDAMPING 1           # FLAG: Point Damping (0: OFF, 1: ON)
FORCEFILTERING 0         # FLAG: Force Filtering (0: OFF, 1: ON)
GRAVITY 1                # FLAG: Gravity default (0: OFF, 1: ON)
VOLUMEFORCE 0            # FLAG: tetrahedron volume force
FORCELIMIT 0             # FLAG: Limit Force (0: OFF, 1: ON)
VINIT 0 0 0              # Initial Velocity: X Y Z
CAMROTX 1.3              # init cam pos: x-rot in rad
CAMROTZ -0.8             # init cam pos: z-rot in rad
END

```

Source Files

The source files headers and bodies are associated the following way:

```

-                : viewer.c++
-                : idle.c++
boundingbox.h    : -
userData.h       : -
paramset.h       : paramset.c++
model.h          : model.c++
                  model_cal.c++
                  model_off.c++
                  model_transform.c++
selfcollider.h   : selfcollider.c++
savescreen.h     : savescreen.c++
externalForces.h : externalForces.c++
                  externalForces_off.c++

```

The *viewer* and *idle* file contain the initialization and the main loop of the simulation parts, respectively. The *model* header includes the data for the volumes and for Cally: common tasks are coded in the *model* file whereas their particularities are separated in the *model_off* and *model_cal* files. The deformation functions of the volumes are coded in a further file, *model_transform*. Collision detection is coded inside the Collision Control library and the corresponding files are not shown here. On the other hand, everything pertaining to collision response is found in the two files: *externalForces* and *externalForces_off*. Finally, regarding self-collision, the algorithm is coded inside the *selfcollider* file.

The class and instance diagram is the following:

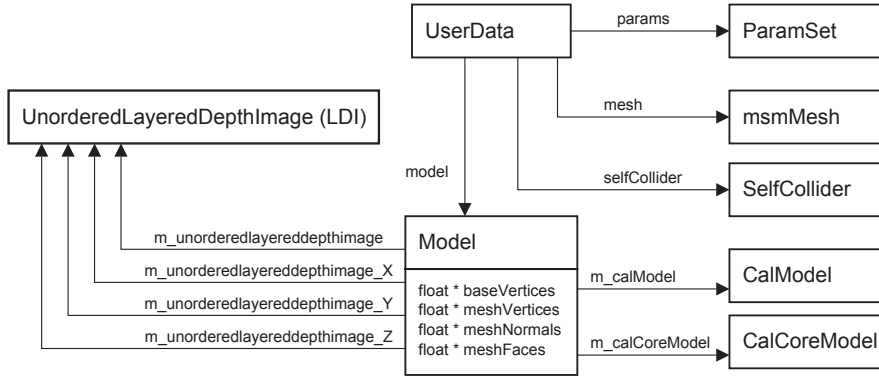


Figure C.1: The class and instance diagram of the application