DIPLOMARBEIT

A ChainMail Algorithm for Direct Volume Deformation in Virtual Endoscopy Applications

ausgeführt am Institut für Computergraphik und Algorithmen
Technische Universität Wien
in Kooperation mit dem
VRVis, Zentrum für Virtual Reality und Visualisierung

unter Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
in Kooperation mit
Dipl.-Ing. André Neubauer
Dipl.-Math. Dr.techn. Katja Bühler

von

Christopher Dräger

Matr. Nr.: 9725493

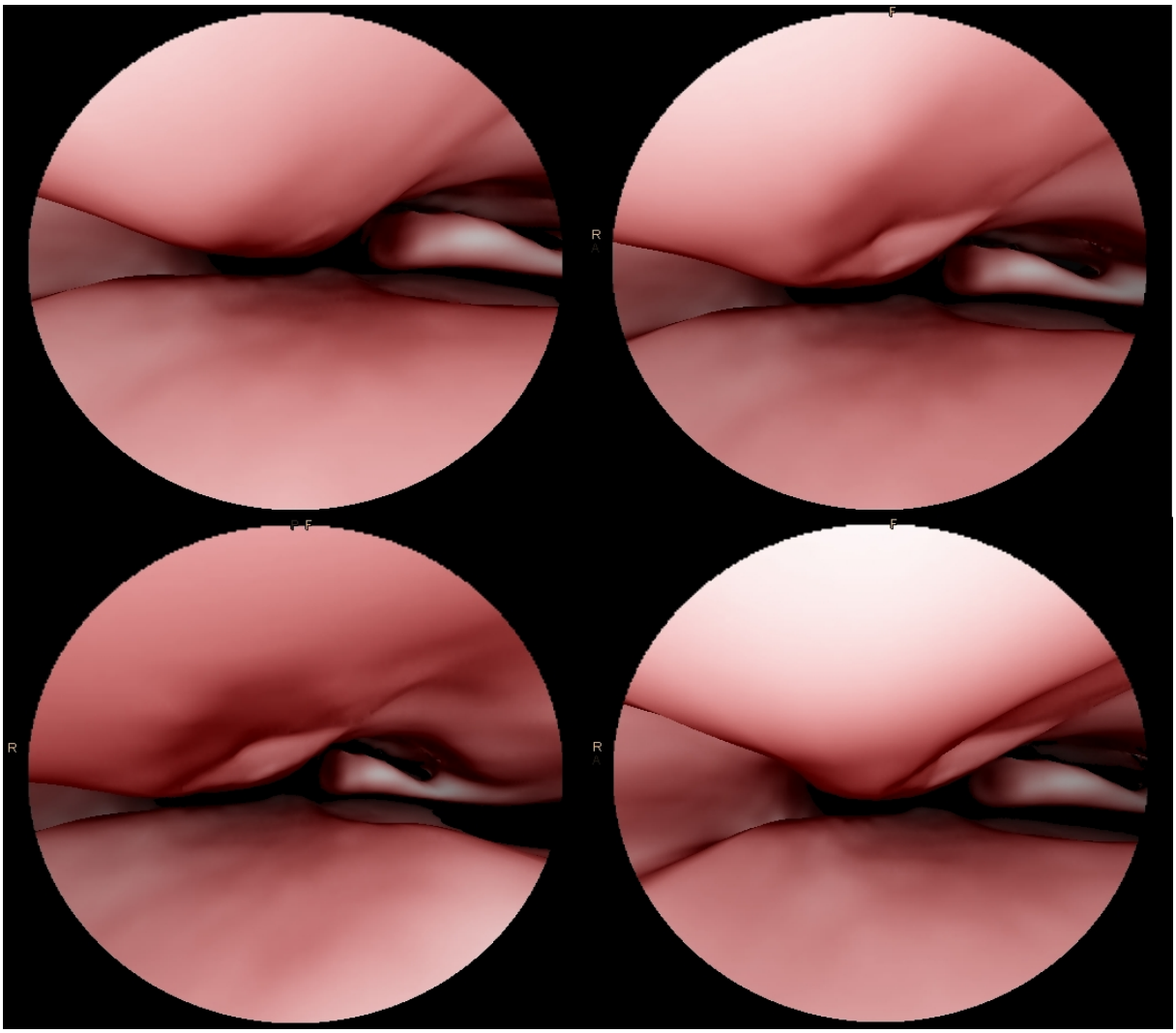
A - 1130 Wien, Trazerberggasse 6 / 1 / d / 3

Wien, im Mai 2005

Christopher Dräger

A ChainMail Algorithm for Direct Volume Deformation in Virtual Endoscopy Applications

(Diploma Thesis)



mailto:e9725493@stud3.tuwien.ac.at

Abstract

Endoscopy is an integral part of medical practice although the procedure is often unpleasant for the patient. Virtual endoscopy is a diagnosis tool which visualizes 3D image data sets to show a virtual view of the patient similar to the images generated by a traditional endoscopy. A virtual endoscopy system called STEPS was developed at the VRVis to simulate transsphenoidal endonasal pituitary adenoma surgery. This thesis describes a deformation engine extending the STEPS application to enhance the realism. The two main contributions are the development of the *Divod ChainMail* algorithm, an algorithm for *direct volume deformation* based on the ChainMail and Enhanced ChainMail algorithms, and the *integration of the engine* into the existing STEPS. We use a Constrained Particle System to directly model and manipulate the volume data instead of modeling objects extracted from the volume data. We further proposes a simple interface to ease integration and enforce reusability of the software components. This thesis describes our *Divod ChainMail* algorithm, the interface and integration details, and provides timing results. Finally, it offers a discussion of the presented methods pointing out possible improvements and further work.

Kurzfassung

Endoskopie ist ein intergaler Bestandteil medizinischer Verfahren, obwohl diese Methode oft unangenehm für den Patienten ist. Virtuelle Endoskopie ist ein Diagnosewerkzeug, das dreidimensionale Datensätze visualisiert und so virtuelle Bilder des Patienten generiert, die den Bildern der traditionellen Endoskopie sehr ähnlich sind. Ein virtuelles Endoskopie System namens STEPS wurde am VRVis entwickelt und dient zur Simulation von endonasalen, endoskopischen Operationen zur Beseitigung von Hypophysentumoren. Diese Arbeit beschreibt eine Deformations Engine welche die bestehende STEPS Applikation erweitert und so den Realismus der Simulation erhöht. Die zwei wichtigsten Beiträge dieser Arbeit sind der *Divod Chain Mail* Algorithmus, ein Algorithmus zur direkten Deformation des Volumens, welcher auf dem ChainMail und dem Enhanced ChainMail Algorithmus basiert, sowie die *Integration der Engine* in das existierende System. Wir verwenden ein Partikel System um das Volumen, anstatt aus dem Volumen extrahierte Objekte, zu modellieren und so die Deformation direkt auf dem Volumen durchzuführen. Weiters stellen wir ein einfaches Interface vor, das die Integration vereinfacht, sowie die Wiederverwendung einzelener Softwarekomponenten unterstützt. Diese Arbeit beschreibt unseren *Divod ChainMail* Algorithmus, das Interface, sowie die Details der Integration und die Resultate der Zeitmessungen. Abschliessend werden mögliche Optimierungen der vorgeschlagenen Algorithmen und Ideen, beziehungsweise Anforderungen, für eine weiterführende Arbeit präsentiert.

Contents

Chapter 1

Introduction

Motivation

In recent years, minimally invasive procedures became an integral part of medical practice [Bartz '03]. Nowadays, they are applied in many areas including surgery, neurosurgery, gastroenterology and radiology.

The main advantage of minimally invasive procedures is that they have a less harmful effect on patients than conventional methods. The endoscope, a tool typically used in minimally invasive procedures, has a very small diameter. This makes it possible to operate in very fragile regions of the human body such as the brain. The application of endoscopy can generally be divided into two classes:

- Diagnosis, done to examine the patient for medical conditions, e.g. tumors, and does not affect the patient.
- Surgery, conducted to cure the patient from illness, obviously affecting the patient.

The drawbacks of minimally invasive procedures include high costs in comparison to radiological diagnosis. However, they are in general cheaper than traditional surgery. Endoscopic examinations are unpleasant for the patient, the endoscope cannot reach all regions of the human body such as thin blood vessels and there is also the risk of infection and other complications.

Virtual Endoscopy

Virtual endoscopy addresses these drawbacks. In virtual endoscopy the region of interest is visualized as a virtual reality. The data for the visualization is typically acquired through Computer Tomography or Magnetic Resonance Tomography. In this virtual environment there are no physical limitations due to the diameter of the endoscope because the lens of the virtual endoscope is a point without expansion. Hence, all regions of interest and very small structures can be examined if the sample

rate of the data is sufficiently precise. This makes virtual endoscopy a feasible tool for diagnosis. Virtual endoscopy is cheaper and less unpleasant for patients than the conventional endoscopy.

Unfortunately, virtual endoscopy introduces new problems. The visualization of the data is not an exact representation of the human body which may lead to wrong diagnoses. Physical properties of the organs have to be modeled with computational expensive simulation engines which threatens interactivity. Hence, there is a tradeoff between speed and realism in every virtual endoscopy system. The mechanisms for interaction should be the same for the virtual and the real endoscopy. This calls for direct user interaction and manipulation of the data such as cutting and deforming. These requirements present new problems to the simulation software. The force feedback of these actions can be simulated with haptic devices. Finally, the radiation exposure during a Computer Tomography session is a risk to the patient too.

Fields of Application

Although virtual endoscopy has limitations and drawbacks it is already applied in many medical areas including the following:

- Diagnosis

  Virtual endoscopy serves as a diagnosis tool. It is used to identify e.g. colon polyps and tumors [Laghi et al. '99]. In this application there is a method introduced called Colon Flattening [Bartrolí et al. '01] to enhance the results and to aid the user.

- Planning

  The planning of a difficult and complex surgery is another field of application. The medical doctor is able to examine the region of interest form different perspectives without the patient needing to be physically present all the time.

- Simulation

  Similar to planning a medical doctor can simulate an intervention several times before performing the surgery.

- Teaching

  Students can study and practice medical procedures in the virtual reality.

- Intra–operative Support

  Virtual endoscopy can also provide intra–operative assistance in the form of navigation assistance or by displaying additional information. Virtual endoscopy also provides different perspectives and can show the position of vital blood vessels and nerves which the surgeon could not see without it.

All of these fields demand a certain degree of realism but tradeoffs are possible. Some applications, diagnosis for example, do rely heavily on the accuracy of the simulation but not so much on physical realism such as the simulation of deformations. Other

applications such as teaching and simulation benefit more from a physical realistic look–and–feel environment than from accuracy. The focus of this work is the latter case.

STEPS

This work deals with a certain type of endoscopy called transsphenoidal endonasal pituitary surgery. Its purpose is to remove pituitary adenomas. In this procedure an endoscope and surgical tools are inserted into the head through the nose and maneuvered to the sphenoid sinus. Then the sellar floor is opened so that the surgeon can reach the pituitary gland and remove the tumor.

This work builds on work previously conducted at VRVis. *A. Neubauer et al.* implemented a transsphenoidal surgery simulation system called STEPS. They use a first hit based rendering technique to visualize the volume data. Polygonal surface rendering is not flexible enough for the given problem. The topology of the object may change rapidly and each topology change would require a computational expensive surface extraction step for a polygonal surface rendering based approach. The work also includes collision detection and force feedback.

Requirements

To enhance the realism even further, an extension of the algorithm to support the simulation of deformation is desired. Hence, the main focus of this work is to introduce a deformation engine, based on a new algorithm that is fast enough to allow interactive frame rates and produces plausible deformations, to the existing STEPS system.

Approach

We propose in this work, a direct volume deformation algorithm called *Divod ChainMail* based on the ChainMail algorithm [Gibson '97] and the Enhanced ChainMail algorithm [Schill et al. '98]. Our algorithm directly models the volume data and calculates the deformation without an expensive surface extraction step. The developed algorithm does not produce physically accurate results, but takes physical components such as material properties (i.e. stiffness) into account.

Further, we present an interface for the integration of the our developed deformation engine into the STEPS application. The interface is designed to allow easy integration and code reusability.

Outlook

The next chapter presents an overview of methods used for soft tissue deformation and their application in medical simulation software with a focus on virtual en-

doscopy. The third chapter describes our *Divod ChainMail* algorithm outlining the improvements made to the original ChainMail algorithms, and discusses the weaknesses of the original approaches in regard to our requirements. The details of the implementation and the integration into the STEPS system are described in the fourth chapter. The results as well as a discussion of the strength and weaknesses of this work are presented in the fifth and sixth chapter including a description of the further work needed.

Chapter 2

Related Work

This work presents a method for deformation of volume data acquired through CT[1] which is visualized with iso surfaces. It is part of an endoscopic surgery planning and simulation system. First, this chapter provides a short overview of the related system followed by an outline of various deformation modeling techniques.

Related Software

This work is an extension of the existing STEPS system [Neubauer et al. '04]. The STEPS (Simulation of Transsphenoidal Endonasal Pituitary Surgery) system is intended to aid the planning of an endonasal surgery and to simulate such a procedure so that students can learn by example. This type of minimally invasive procedure is used for the removal of various kinds of pituitary tumors. It consists of four views. The virtual view and three section views which show different cutting planes through the data (please see figure 2.1).

Motivation

In order to enhance the realism of the simulation a mechanism for simulating the deformation of soft tissue is desired. Currently, if the user pushes the virtual endoscope against the tissue the movement of the endoscope is blocked in order to keep the eye point of the endoscope outside of the tissue. The real life behavior of the tissue is different. In real life the tissue would deform according to its material properties and the forces applied. For example a surgeon is able to push interfering tissue aside. Overall, deformation and movement of the virtual tissue gives a much more realistic feeling of the simulated environment. Bringing movement to the virtual reality is also a first step of injecting life to it.

[1] Computer Tomography

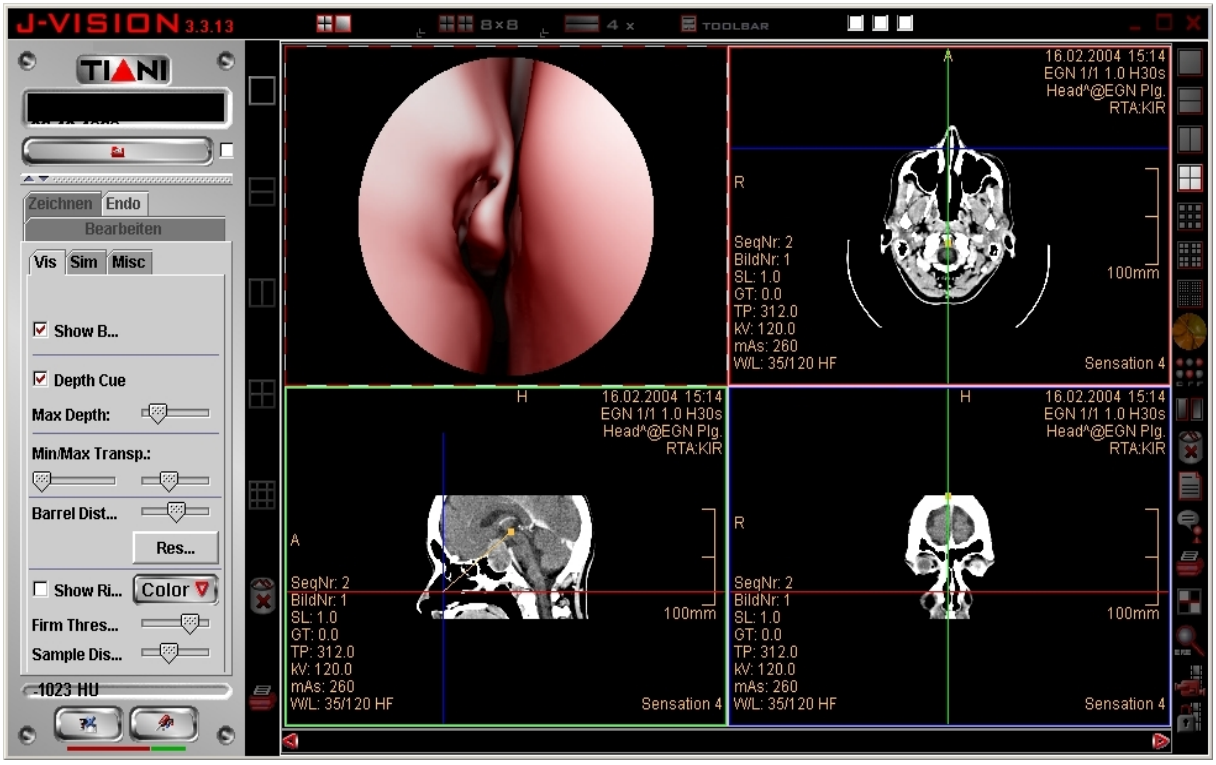


Figure 2.1: This is a screenshot from the STEPS application. The top left image shows the three dimensional view. The other three images are the cutting plane views also called section views. On the left side the user interactively adjust parameters such as the barrel distortion, color and fog.

Requirements

The algorithm for calculating the deformations has to produce interactive frame rates and must be integrated into existing software. The most important existing software component is the ray caster [Neubauer '01]. It directly renders the iso surface on basis of volume data. The shape of the rendered surface depends on a threshold. This threshold defines which iso values are interpreted as tissue and which iso values are interpreted as air. The data is acquired by CT images and the threshold can be adjusted interactively. This allows the user to properly fit the threshold to the data so that reasonable structures are rendered. This is a useful tool for users but it presents a massive limitation for the deformation approach. It is not feasible to calculate a mesh for the iso surface, since the iso surface can change rapidly and hence, the mesh would need to be recalculated. This would lead either to low performance or to the loss of the interactive threshold adjustment feature. Hence, an algorithm has to be chosen that does not rely on the use of meshes. Our proposed *Divod ChainMail* algorithm directly manipulates the volume data and thus does not use meshes.

Another limitation is presented by the large data which cannot be kept entirely in the memory. Hence, it would be necessary to swap the currently needed parts of the data; a time consuming process. Therefore, global deformation is not possible at interactive frame rates. This means, that the algorithm has to calculate the deformation locally.

Outlook

Allowing real time deformations in the STEPS system is the main focus of this work. The next sections provide an overview of existing deformation algorithms and their application in virtual reality environments, emphasizing on those used for surgery simulation and soft tissue modeling.

The conclusion discusses the pros and cons of the various presented algorithms. It also argues why the ChainMail algorithm 2.7 was chosen as approach for our work, the *Divod ChainMail* algorithm, is based on.

2.1 Volume Deformation Algorithms

Much research has been done in the field of deformable modeling for various fields of application such as animation of cloth, animation of muscles, simulation of car accidents, computer games, virtual reality or surgery simulation.

The different approaches can be divided into geometrically-based and physically-based. Geometrically-based methods change the shape of an object indirectly by moving control points or adjusting parameters of the function which defines the surface. The advantage of these methods is their high performance. The drawbacks are:

- they do not model physical properties of material
- they do not use physical laws to calculate the new shape
- since the deformation is calculated indirectly it is difficult to provide direct manipulation of an object and intuitive interaction

In contrast to geometrically-based methods the physically-based methods incorporate physical properties of the modeled material and deform the object directly which offers intuitive and direct user interaction. This comes with high computational costs which makes it hard to achieve interactivity.

Deformations can also be divided in plastic [O'Brien et al. '02], [Terzopoulos, Fleischer '88] and elastic deformations. An elastic deformation assumes that the object attempts to keep the original shape. Hence, the object tries to stay in its original shape against the applied forces. If the applied forces are removed from the object a perfectly elastic object will restore its original shape. In contrast, plastic deformations permanently change the shape of an object. If

the applied forces are removed the object keeps the deformed shape. Most physical deformations have a plastic and an elastic component. A system that models this behavior is described in [Teschner et al. '04].

2.2 Particle Systems

Particle systems are a particularly good method for modeling objects that change over time by flowing, billowing or expanding such as clouds, water, smoke, fire, etc. [Hearn, Baker '97]. In a particle system an object is modeled through a set of particles. Each particle may hold a set of properties such as lifetime, shape, transparency and color. For example, water is modeled through a large number of drops. *A. Witkin* gives a basic introduction to particle system dynamics in [Witkin '01b].

Particle systems are a wide spread method to model the dynamics of body fluids such as blood. *U. Kühnapfel et al.* use a particle system based method in the software package called KISMET of their surgery simulation system to model blood to enhance the realism of the entire simulation [Kühnapfel et al. '00].

However, particle systems are not only used to simulate fluids. *M. Nakao et al.* use a particle system combined with adaptive tetrahedral subdivision to model accurate and real-time soft tissue cutting and deformation in a framework for advanced surgery simulation [Nakao et al. '03].

Particle systems are also applied in the modeling of cloth. For example, *B. Eberhardt et al.* use particle systems to model the draping of textile [Eberhardt et al. '00]. They present a flexible method which allows modeling of very stiff materials. Their approach treats nonlinear forces correctly which is especially important to produce accurate results in the context of high stiffness. Due to the high flexibility the system can be suited to different applications such as virtual reality and high accuracy simulation.

2.2.1 Constrained Dynamics

In a physical model particles and their movements are governed by a set of constraints. For example two particles may only be a certain distance apart or a particle must move along a specific move path.

The problem that arises from these constraints is to make the particles obey the constraints as well as the applied forces such as Newton's law of gravity. This may result in a very complex system. The computational effort to meet all constraints with respect to the applied forces within such a system may become very time consuming and the calculation itself numerically unstable.

For a deeper understanding of these problems and basic concepts to counter them please refer to [Witkin '01a].

Some of the outside forces applied to an object may occur through the collision with another object. A short overview over this topic is presented in the next section.

2.2.2 Collision and Contact

Collision and contact is a vital part to simulate interaction between objects and user interaction within a virtual reality. In the real world colliding objects usually do not interpenetrate each other but they deform or change their move path in response to the collision. Hence, the first step to model deformations accurately is to detect collisions and contacts between objects. The second step is to check if the detected collision results in an invalid state of the system. If so, the objects have to be deformed according to their material properties so that the sanity of the system is restored.

S. Baraff [Baraff '01] gives an introduction to the topic as part of the SIGGRAPH 2001 course notes.

2.2.3 Modeling

Particle systems in general can be used for soft tissue modeling. However, their main field of application is the modeling of fluid or gasiform objects. Constrained Particle Systems are better suited for soft tissue modeling. An algorithm based on particle systems with constraint dynamics namely the ChainMail algorithm is outlined in section 2.7.

2.3 Mass Spring Model

A very wide spread physically-base approach is the Mass Spring Model. An object is defined through a set of mass points which are connected by dampened springs. If a deformation occurs it is propagated through the object by the dampened springs in respect to the forces applied and the characteristics of the springs. The method is easy to implement and provides fast computation.

The deformation is calculated through solving differential equations. This family of equations, especially the so called *initial value problem* class, is very important for physically based modeling. A profound introduction to the topic can be found in [Witkin, Baraff '01].

A way to exploit the advantage of the Mass Spring Model is to model different layers of material with different layers of Mass Spring Models as presented by *D. Terzopoulos* and *K. Waters*. They use three Mass Spring layers for facial animation. The three layers are associated with three anatomical layers of facial tissue (dermis, subcutaneous fatty tissue, and muscle)[Terzopoulos, Waters '90].

Another example for the potential of Mass Spring Models is found in a paper done by *U. Kühnapfel et al.* who use the Mass Spring technique for endoscopic surgery

training simulation. Their paper presents a surgical training system called "Karlsruhe Endoscopic Surgery Trainer"[Kühnapfel et al. '00]. The software for the system called KISMET uses a Mass Spring Model to simulate the elastodynamics of the simulated objects.

A method for real time muscle deformations based on the Mass Spring Model is presented by *L. P. Nedel et al.*. Their main goal is to produce fast and plausible results. Hence, they did not strive for a perfect simulation. They introduce a new type of springs called *angular springs* to control the muscle volume during simulation [Nedel, Thalmann '98]. They also provide two different levels of muscle representation:

- muscle shapes: A surfaces based model fitted to the boundaries of the medical image data.
- action lines: Represent the force a muscle produces at the connected bone.

The works described above had to cope with the drawbacks of the Mass Spring Model. The drawbacks include unrealistic behavior for large deformations and finding appropriate values for the spring constants to produce realistic behavior. This is especially troublesome for rigid objects such as bone, a problem referred to as *stiffness*. Modeling stiff objects results in a huge loss of performance. Many calculations have to be done in a short period because only small steps can be taken.

An approach similar to the Mass Spring Model is presented by *M. Teschner et al.* [Teschner et al. '04]. This approach is based on tetrahedral meshes and derives three distinct forces based on potential energies at the mass points. These three forces are modeled to:

- preserve distances between mass points
- preserve the surface area of the object
- preserve the volume of tetrahedra

The properties of the material are governed by weighted stiffness coefficients. Each material property consist of three coefficients – one for each potential energy. The three different forces are calculated through the derivation of the corresponding potential energy function.

The big advantage of this approach is that it models both plastic and elastic deformations. This is done by splitting the calculation in two components, a plastic and an elastic component, where only the elastic component contributes to the deformation energy of the object.

The computational effort of this approach is comparable with the effort of a Mass Spring algorithm. Hence, it produces interactive frame rates for environments with several thousand deforming primitives. One of the keys for this performance is the Verlet [Verlet '67] algorithm for numerical integration. It only needs one force computation per integration step which is the most expensive operation in this approach. Figures 2.2 and 2.3 show the results of this work.

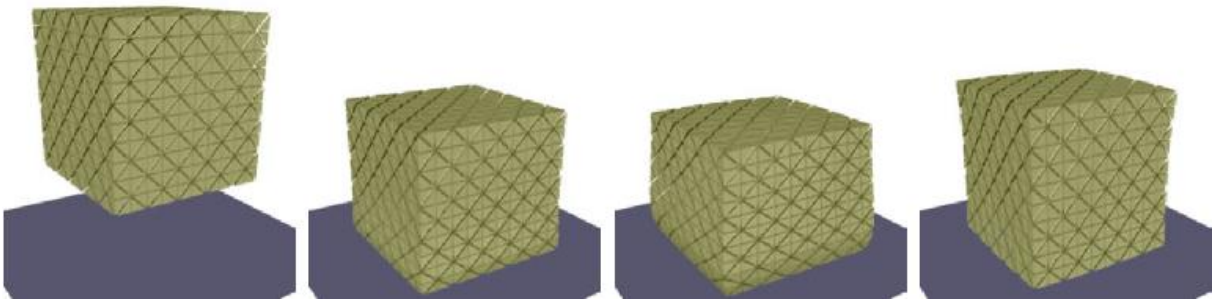


Figure 2.2: Deformation of a falling cube [Teschner et al. '04]

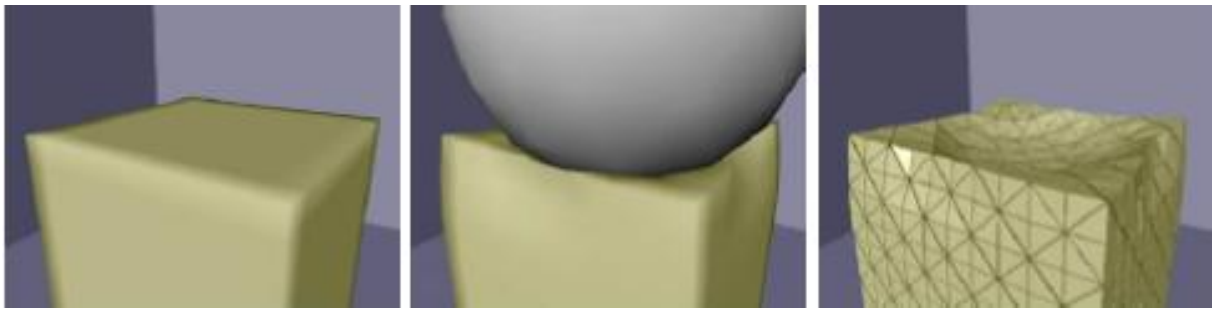


Figure 2.3: Sequence of a plastically deformed cube. [Teschner et al. '04]

Due to the versatility and effectiveness of this approach it qualifies for a wide range of applications. The authors currently work on integrating their approach in surgery simulation such as hysteroscopy simulation and simulation of stent placement.

Mass Spring Models are among the most commonly used techniques for soft tissue modeling. Even though they suffer from some minor drawbacks their speed combined with the capability to model physical attributes of material make them a good choice for soft tissue simulation.

Another commonly used approach called Finite Element Method is presented in the next section.

2.4 Finite Element Method

The Finite Element Method is a very wide spread approach to volume deformation in all fields of application. It is a physically-based method which is capable of producing very realistic results.

In the linear elastic model the Finite Element Method assumes small deformation steps which is true for stiff objects such as metal. For soft tissues this assumption does not hold since large deformation steps are possible. However, the major drawback

of the Finite Element Method (FEM) is its high computational cost which does not allow interactive frame rates the nowadays hardware performance. In general, using fewer nodes increases the computational speed but decreases the accuracy of the results. A different approach is presented by *Q. Zhu* who applies FEM to simulate the macroscopic dynamics of muscle [Zhu '98] emphasizing on a multi resolution hierarchy on the grid to accelerate the computational speed. *F. Ganovelli et al.* address another drawback of the Finite Element Method:

> "However, the use of FEM requires a preprocessing phase strongly depending on the topology of the object, hence preventing the possibility to cut the object." [Ganovelli et al. '00]

Nienhuys and *A. F. van der Strappen* also address the problem of cutting using conjugate gradients [Nienhuys, Strappen '96]. In [Bro-Nielsen, Cotin '01] *M. Bro–Nielsen* and *S. Cotin* introduce three ways to speed up FEM with their algorithm (see figure 2.4). They exploit the sparse structure of the force vector, explicitly invert the system matrix and use condensation which compresses the system matrix and results in a system with the complexity of a surface model but the behavior of a volumetric model.

The Finite Element Method is often used because it is capable of modeling complex physical properties of different materials and therefore, able to produce realistic results. These come at a very high performance cost. Hence, this method is governed by the tradeoff between performance and realism even more than others. In contrast to physically-based methods the next sections presents Radial Basis Functions, a geometrically-based technique.

2.5 Radial Basis Functions

Radial basis functions are a geometrically-based method used for volume deformation. They are applied in many fields such as 2D and 3D computer animation, medical applications as well as reconstruction of 3D scattered data. Basically the approach uses radial functions as an interpolation function between a set of control points. The class of radial functions has the characteristic that their response increases/decreases monotonically with the distance from a center point.

N. Kojekine et al. divide radial basis functions into three classes. The first one, called "native methods" [Savchenko, Schmitt '01] is used for small data sets and is restricted to small problems. Its high computational cost makes it unfeasible for real-time animation. The second class of radial basis functions allows modeling of large data sets since it consists of fast methods. The third class are the so called CSRBFs (Compactly Supported Radial Basis Functions) described in [Wendland '95].

N. Kojekine et al. improve CSRBF to models for surface deformations by optimizing the algorithms for speed as well as memory consumption [Kojekine et al. '02]. A sample animation is presented in figure 2.5.

An approach using radial basis functions for medical imaging is presented in [Carr et al. '97] by *J. C. Carr et al.*. They use Radial Basis Functions (RBFs) to visualize human skull from depth maps obtained by X-ray or CT data even over defect areas. They speed up the algorithm by making assumptions about the geometric constraints of the nodes of interpolation.

P. Reuter et al. use point based CSRBF approach. The surface is modeled through a set of surface points which are rendered directly without the need for the creation of a polygonal mesh [Reuter et al. '03]. Their approach also allows direct user interaction through manipulating the surface points.

Another geometrically based approach, the so called Free Form Deformation, is presented in the next section.

2.6 Free Form Deformation

Free Form Deformation (FFD) was first introduced by *T. Sederberg* and *S. Parry* [Sederberg, Parry '86] and is a fast tool for representing and modeling flexible objects. It is originally a graphically-base method but recent work focuses on the integration of physically based techniques, for example [Hirota et al. '99].

G. Hirota et al. describe a physically-based extension for Free Form Deformation. The governing physical law of their approach is the conservation of mass. In other words, the algorithm they developed is volume preserving. This allows more intuitive modeling and enables designers to easily keep the desired proportionality of objects with respect to their volume in a complex design with multiple objects.

Since Free Form Deformation is a graphically-based technique the shape of an object is defined by control points. Intuitive deformation through direct interaction with the model is not possible. *P. Borell* and *D. Bechmann* [Borell, Bechmann '91] and *W. S. Hsu et al.* [Hsu et al. '92] address his problem. They both present models for direct interaction and manipulation of the model which makes the user interface more intuitive. Both methods calculate the necessary adjustment of the control points for a given set of input points by means of least-square formulation.

Free Form Deformations are also applied in surgery simulation. *C. Basdogan et al.* use a FFD approach to model local deformations for laparoscopic surgery simulation. The authors decided to use FFD because of its high speed which enables them to create interactive frame rates [Basdogan et al. '98].

Another example for surgery simulation based on FFD is presented by *G. Sela et al.*. They introduce an algorithm for real-time incision simulation for meshed surfaces and volumetric models which supports polynomial as well as spline based models. A new type of FFDs called DFFDs which support discontinuities is used to calculate the immediate geometry change due to cuts and the real-time response of the local tissue due to tension and internal forces [Sela et al. '04].

Free Form Deformations present an efficient way for deformation. With the introduction of physically-based concepts and direct object manipulation FFDs are well suited for surgery simulation. The next section presents another technique, called 3D ChainMail, which is geometrically-based with extensions to model physical properties of an object.

2.7 3D ChainMail

A promising approach to soft tissue deformation called 3D ChainMail is presented by *S. F. F. Gibson* [Gibson '97]. The method was originally created for the deformation of volumetric objects as needed in surgical simulation for example. It is geometrically-based but it is capable of simulating material properties to some extent. The algorithm is extended to model the differences between types of tissue and their interaction in [Schill et al. '98]. The underlying data structure can be compared to a chain mail in the 2D case extended by a third dimension in the 3D case. It is the key to the entire algorithm. (compare figure 2.6). The basic idea of the algorithm works as follows. The elements of an object are linked together like elements of a chain mail. If one element is moved there is a chance that it will also make its neighbor elements move since they are connected like a chain. If the moved element stays within the boundaries of the neighbors, the neighbors do not have to be moved. If the moved element violates the boundaries of its neighbors, the neighbors have to be moved to satisfy the boundary constraints again. If a neighbor is moved, then its its neighbors are moved too if necessary. Like this, the movement of one element is locally propagated through the object. A sample deformation of a two dimensional object is shown in figure 2.7.

The 3D ChainMail algorithm consists of two steps to calculate the deformation that occurs when an element is moved. The first step calculates the movement of the neighbor elements of the moved element. If any neighbor element has been moved, the movement of their neighbors is calculated too and so forth. In the second step, a relaxation step, the object is relaxed by locally adjusting the elements until the object reaches a valid state of minimum energy.

The main advantage of the 3D ChainMail algorithm is its performance. *S. F. F. Gibson* has shown that each element has to be processed at most once. This allows the algorithm to work on a large data set and still produce interactive response times. A topology change can easily be done by linking or unlinking elements hence, it supports actions like cutting for example an important feature required for surgery simulation.

However, the proposed algorithm has some major drawbacks. First of all, it is restricted to the use of rectilinear grids and second it only works on homogeneous data. Two papers have been published which introduce methods to overcome these restrictions.

The restriction to rectilinear grids is addressed by *Y. Li* and *K. Brodlie* [Y. Li '03] who introduce a Gerneralised ChainMail algorithm. This approach allows any number of neighbors for an element and does not make assumptions about the topology of the neighbors. The original 3D ChainMail assumed at most six neighbors and made assumptions about their respective position. The six designed positions are left, right, top, bottom, front, back. The restriction of rectilinear grids is overcome by using relative rather than absolute values for the boundary constraints.

> "Note the significant difference from the original ChainMail algorithm, in that the softness and shearing parameters are expressed relative to the length of the original link between A and B, rather than as absolute distance values." [Y. Li '03]

M. A. Schill et al. introduce an algorithm to enable the modeling of inhomogeneous data [Schill et al. '98]. The basic idea is to change the chain boundaries of the elements. The movement is governed by the shape of the boundary assigned to a chain mail element. Different types of tissue are modeled with different shapes of chain regions. The problem that occurs with the introduction of inhomogeneous chain regions is that it can not be proven that each element only has to be processed once. Hence, the speed advantage of this algorithm is lost. *M. A. Schill et al.* solved this problem by the use of sorted lists during the neighbor movement calculation. This increases the computational cost but the algorithm is still able to produce interactive frame rates.

The 3D ChainMail algorithm is a very fast and capable method for soft tissue modeling. It can be compared to the Mass Spring Model and allows cutting without further improvements. The next section classifies the different approaches by their underlying data structure.

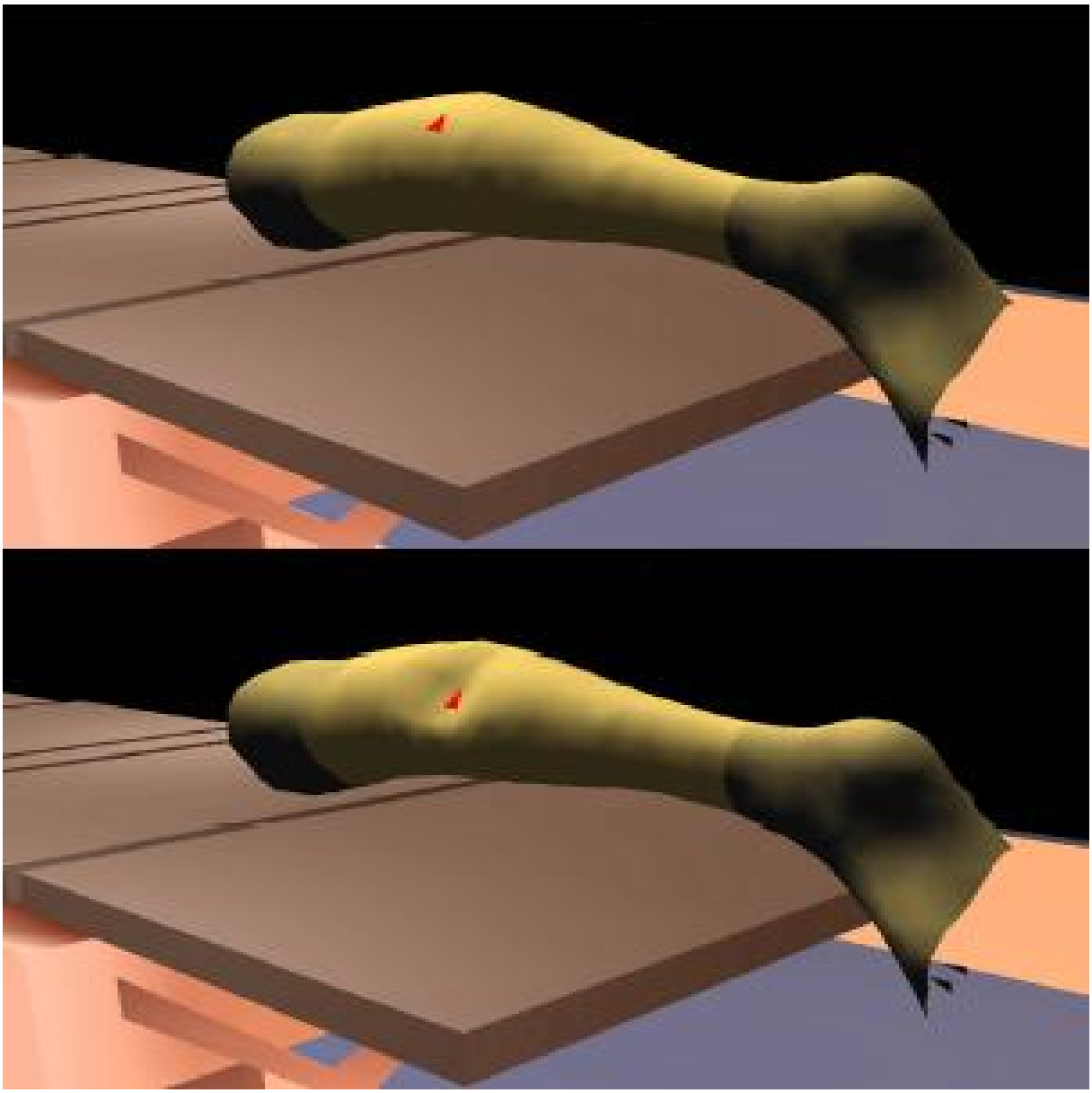


Figure 2.4: Simulation of a deformation using Finite Elements and Condensation [Bro-Nielsen, Cotin '01]

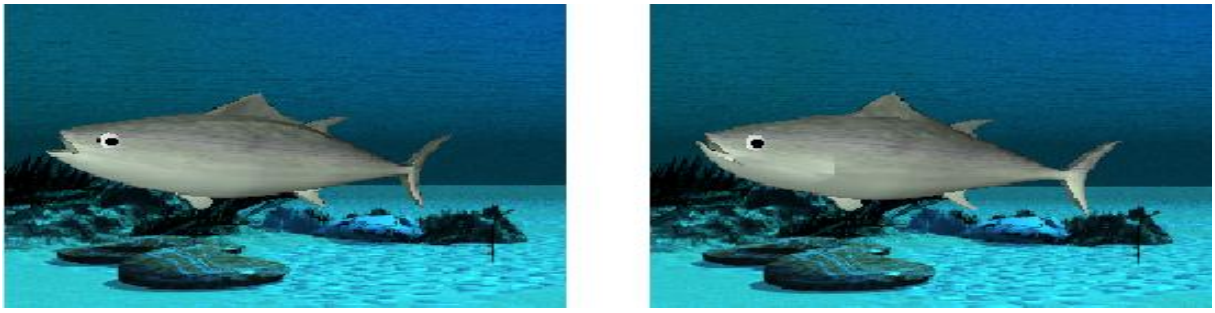


Figure 2.5: CSRBF Sample Deformation [Kojekine et al. '02]

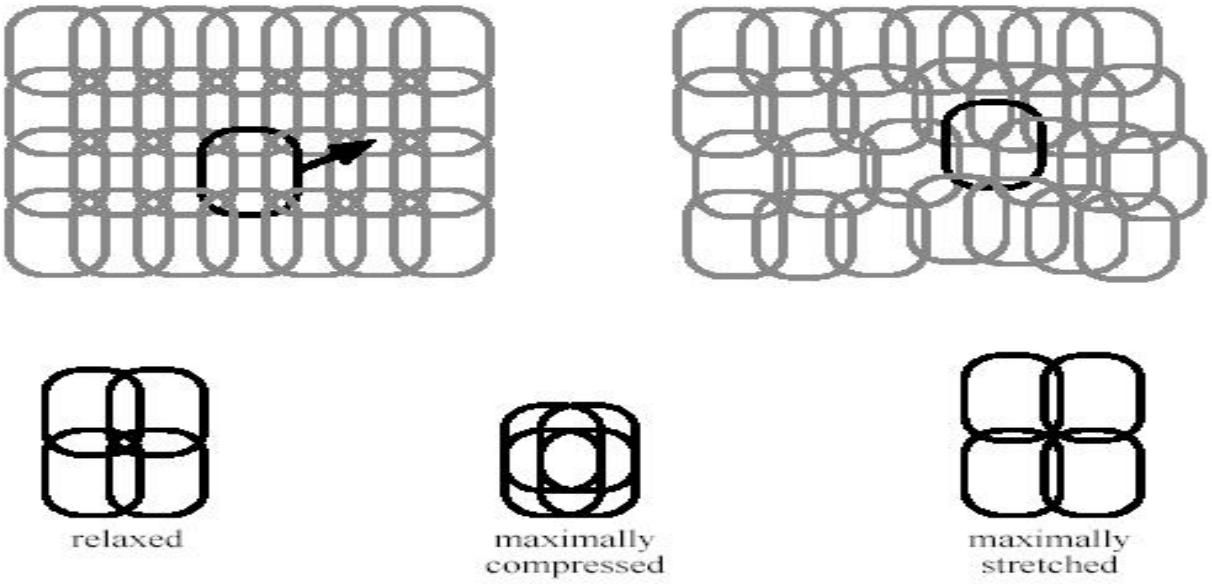


Figure 2.6: 2D ChainMail structure [Gibson '97]

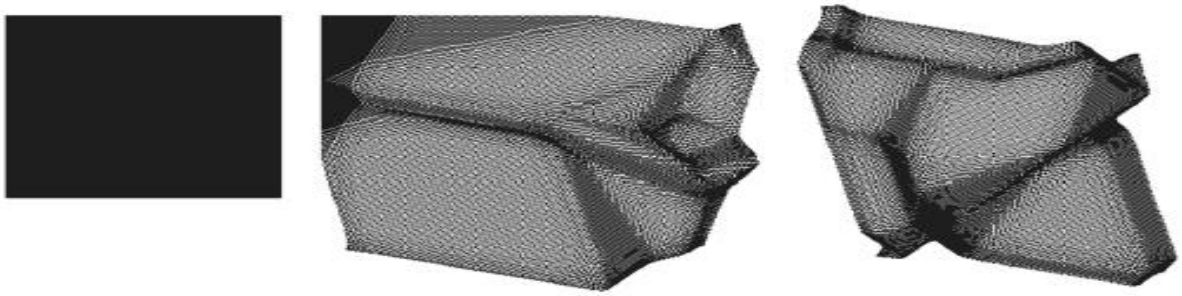


Figure 2.7: 2D ChainMail deformation example [Gibson '97]

2.8 Comparison and Classification by Data Structure

To outline the main difference between these deformation modeling approaches and our approach we introduce a third classification of the algorithms based on the data structure they use for modeling the object.

Generally, three underlying types of data structures can be identified:

- Meshes: The object is modeled through a surface or volumetric mesh. Typically, the nodes of the mesh represent mass points and store additional information such as color. Commonly used surface meshes are triangular meshes. Tetrahedral meshes are their volumetric dimensional counterpart.
  See: [Teschner et al. '04], [Bro-Nielsen, Cotin '01], [Terzopoulos, Waters '90]
- Control Points: The shape of the object is indirectly defined by a set of control points. An interpolation function is applied to render the surface depending on the interpolation parameters, the position and the properties of the control points.
  See: [Sederberg, Parry '86], [Kojekine et al. '02]
- Particles: An object is modeled by a large amount of small particles. The relationship and movement and movement of the particles is usually governed by a set of constraints.
  See: [Kühnapfel et al. '00], [Reuter et al. '03], [Eberhardt et al. '00]

All algorithms described in this chapter use these data structures to model the shape of an object. However, the key concept in our approach is to directly model the volume data without extracting object and topology information. Despite, we propose an algorithm to directly model and manipulate volume data.

In contrast to the presented approaches, our approach directly models the volume data on the base of a Constrained Particle System. It exploits the structure of the volume data set. A volume data set consists of density values which are arranged in a rectilinear grid. We use a particle system can to model these density values. In our approach each particle represents one density value. Additionally, neighborhood and movement constraints are introduced to govern the particles' movement.

2.9 Conclusion

This chapter presented a vast variety of methods and algorithms for soft tissue deformation, both physically and geometrically based. Each of these methods has its advantages and disadvantages, since every solution is tailored to a specific problem.

Requirements

There are two key requirements for this work which influence the choice of the algorithm applied. The first and most important feature is interactivity. The algorithm

has to allow interactive frame rates and direct user interaction. The second restriction arises from the work of *A. Neubauer* [Neubauer et al. '04]. His algorithm allows the user to interactively change the threshold to adjust the rendered iso surface. This feature must be preserved which makes every mesh based approach unfeasible due to the high computational cost of the mesh generation. Instead we propose an algorithm that directly models the volume data.

Mesh–Based Approaches

The two criteria described above instantly rule out Mesh based approaches as well as highly realistic physically based approaches. These algorithms come with a high computational cost. The use of meshes is not flexible enough because the topology of the object may change very quickly.

Geometrically–Based Approaches

Although there are direct user interaction extensions to the geometrical algorithms they do not satisfy the requirements either. These algorithms, such as the Free Form Deformation and the Radial Basis Functions, are based on control points. Hence, the control points need to be generated from the iso surface. This presents a problem because the control points need to be updated during each threshold change. Such a control point update requires a surface extraction step. Additionally, although the main focus of this work is not physical correctness the algorithm should be physically plausible to a certain degree.

Particle Systems

Although a particle system where the particles are connected through dampened springs is possible, the vast majority of the wide spread Mass Spring Model uses meshes as data structure which makes it unfeasible for this work.

Finally, constrained particle systems meet the two criteria. The do not involve meshes and they are fast enough for interactivity. A model of Constrained Particle Systems described in this chapter is the ChainMail algorithm. This algorithm satisfies the requirements for direct user interaction, speed and flexibility. It was originally designed to model deformation of volumetric objects. However, we use it to directly model and deform volume data instead.

The *Divod ChainMail* algorithm including the necessary adjustments to the ChainMail algorithm is presented in the next chapter.

Chapter 3

The *Divod ChainMail* algorithm

3.1 Overview

Basically, the Direct Volume Deformation ChainMail algorithm (*Divod ChainMail* algorithm) for local direct volume deformation consists of the three parts deformation, mapping and memory management. The first part calculates the deformation of the *ChainMail object*. The second part is responsible for mapping the original volume data to the *ChainMail object* and mapping the deformed *ChainMail object* back to the volume data. The third part handles the loading of the necessary portions of the *ChainMail object* into memory.

The calculation of the deformation is based on the 3D ChainMail algorithm first introduced by *S. F. F. Gibson* [Gibson '97] and its enhancement to inhomogeneous data presented by *M. A. Schill* [Schill et al. '98]. The two algorithms are described in section 3.2 and 3.3. The Generalised ChainMail algorithm [Y. Li '03] was not used for our approach since the modeled voxels lie in a rectilinear grid. Therefore, an extension to a non rectilinear grid as proposed by the Generalised ChainMail algorithm is not necessary.

The shortcomings of these approaches in respect to *A. Neubauer's* work are laid out in section 3.4 followed by a presentation of our solutions to these problems in section 3.5.

Pseudo–code listings of our algorithm are presented in the next section. A detailed description of the *Divod ChainMail* algorithm and the adjustments made is given in section 3.6.

The mapping algorithm described in 3.6.3 uses barycentric coordinates to calculate the new voxel values of the volume. A voxel is a single volume element which represents the density information at the given volume position.

The memory management outlined in section 3.6.4 exploits the fact that the deformation propagates locally and outwards through the object. The entire volume is subdivided into a macro grid of cubes which are loaded on demand to minimize memory usage.

Finally, section 3.7 rounds off the chapter with a conclusion and an outlook to further work.

3.2 Original Algorithm [Gibson '97]

The 3D ChainMail algorithm was originally designed as a fast algorithm for deforming volumetric objects. It is also capable of modeling a wide range of material properties and anisotropic materials. These are materials which have different properties along different axes. The approach is based on techniques used in volume graphics, physically-based graphics and soft tissue modeling with Finite Element Methods.

Data Structure

The data structure of the algorithm called *ChainMail object* consists of elements which represent the volume data. Each element may hold various properties such as color and transparency. In our case the elements only store density values. Additionally, each element holds information about the deformation and elasticity parameters. These parameters are the same for every element of a *ChainMail object* in the original ChainMail algorithm. An element also stores its left, right, top, bottom, front and back neighbors and its last position. This is necessary to quickly revert a movement that resulted in an invalid object state. The algorithm also allows cutting of the object by simply unlinking elements.

The processing step of the ChainMail algorithm requires the use of six unordered *candidate lists* which are further outlined in the description of the algorithm. A moved elements list is used to track the moved elements.

Deformation Parameters

Two neighboring elements are related to each other by their deformation parameters called *chain region constraints*. In the 2D case an element must lie within a relative horizontal distance between min_{dx} and max_{dx} from its left and right neighbor and within the relative vertical distance between min_{dy} and max_{dy} from its top and bottom neighbor. These distance constraints govern the stretch and contraction of the object. Additionally, the element must lie within the relative horizontal distance between $\pm shear_{dx}$ from its top and bottom neighbors and the relative vertical distance between $\pm shear_{dy}$ from its left and right neighbors. These constraints govern the shearing of the object. In the homogeneous case all deformation constraints have to be equal for all elements of a *ChainMail object*. Please refer to figure 3.1 for a 2D and figure 3.2 for a 3D example. Figure 3.3 shows how chain regions work together to define the valid regions.

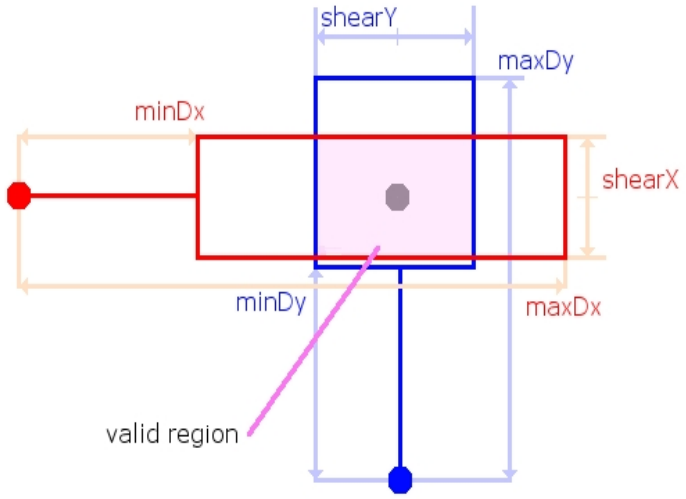


Figure 3.1: In this figure, the valid region for the element is defined by its left and bottom neighbor. The left neighbor's constraints are shown in red, the bottom neighbor's constraints are shown in blue.

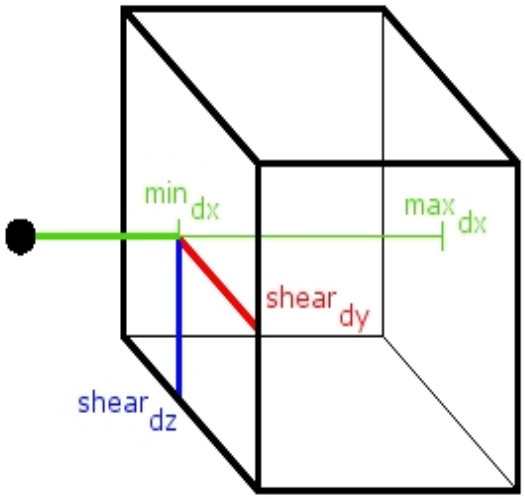


Figure 3.2: This figure shows a sample chain region for the right neighbor. In this case the three dimensional the chain region is defined by a cube with side lengths $2 * shear_{dy}$, $2 * shear_{dz}$ and $max_{dx} - min_{dx}$.

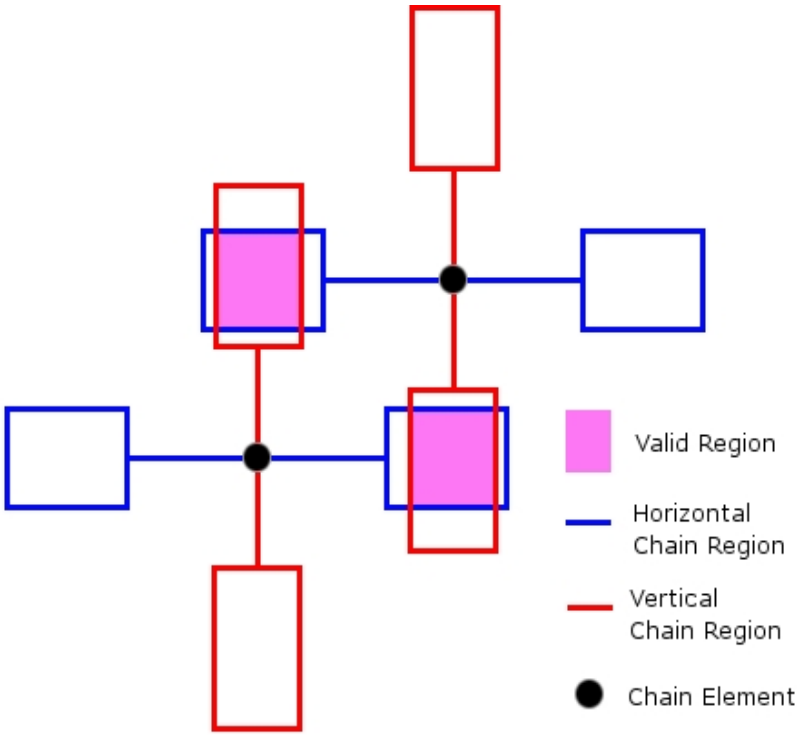


Figure 3.3: This figure shows the chain regions for two elements. Note that all horizontal regions have the same shape and size, as well as all vertical chain regions have the same shape and size. The valid region for a neighbor is the intersection between the corresponding horizontal and vertical chain regions.

In the 3D case the additional parameters min_{dz}, max_{dz} and $shear_{dz}$ are introduced to govern the movement along the third axis.

Algorithm

Each neighbor element has to satisfy the chain region constraints otherwise the *ChainMail object* is not in a valid state and the violating elements are moved until they satisfy the region constraints.

The behavior of the elements in the one dimensional case is like the behavior of a chain. An example where the object's elements model the elements of a chain is given in figure 3.4.

Extending this analogy to the second dimension the elements behave as chain elements of a chain mail. A single element may be moved a certain path without interfering with the neighbors, but if it is moved too far the neighbor chain mail elements are dragged and moved too.

Hence, if one ChainMail element is moved because of a collision or interaction and

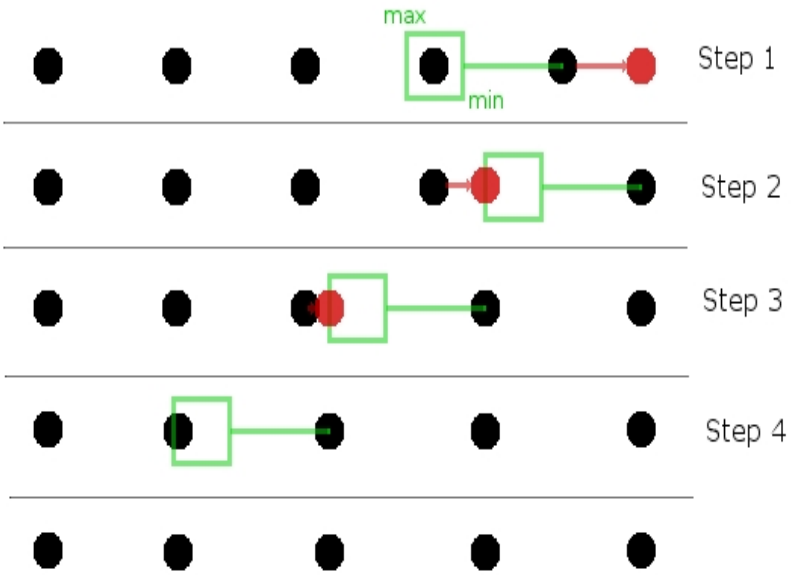


Figure 3.4: A sample deformation of a one dimensional chain. The initial movement of the right most element forces the left neighbors to be dragged in order to satisfy the region constraints. Note, that the drag necessary in the third step is smaller than in the second step. In the fourth step no drag is necessary and the deformation terminates.

the deformation constraints are violated this also moves the violating neighbor elements. The initially moved element is the so called *sponsor* of the neighbor element's movement. If the neighbor element is moved it becomes a potential sponsor for its neighbor elements too. They also have to be moved if they do not satisfy the constraints after the violating element was moved and so forth. Through this mechanism the deformation is propagated locally through the *ChainMail object*. With each adjustment the overall violation decreases until it is below the constraint limit.

Candidate Lists

The checking and tracking of the potentially violating chain elements is done using unordered candidate lists. If an element is moved all its neighbors are assigned to the respective candidate list. There are six lists for a three dimensional object: right, left, front, back, top and bottom. The element is also added to the moved elements list and stores its last position.

The candidate lists are processed one by one until no candidates remain or the system

enters an invalid state, e.g. collision with another object. In the latter case all moved elements are set to their last position and the deformation is retried using a smaller step size for the initial movement. The order of processing of the list is right, left, top, bottom, front, back.

The processing of the right candidate lists begins with checking the violation constraints between the first element with the coordinates $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ in the list and its sponsoring element $\begin{pmatrix} x_{left} \\ y_{left} \\ z_{left} \end{pmatrix}$ which is the left neighbor. In case of a constraint violation the candidate element is moved a minimum distance to satisfy the constraint using the following formulas:

$$\text{if}(x - x_{left} < min_{dx}), x = x_{left} + min_{dx}$$

$$\text{else if}(x - x_{left} > max_{dx}), x = x_{left} + max_{dx}$$

to calculate the stretch and contraction and:

$$\text{if}(y - y_{left} < -shear_{dy}), y = y_{left} - shear_{dy}$$

$$\text{else if}(y - y_{left} > shear_{dy}), y = y_{left} + shear_{dy}$$

$$\text{if}(z - z_{left} < -shear_{dz}), z = z_{left} - shear_{dz}$$

$$\text{else if}(z - z_{left} > shear_{dz}), z = z_{left} + shear_{dz}$$

to calculate the shear.

In case the element was moved, its right, top, bottom, front and back neighbors are added to the respective candidate lists. Afterwards, the candidate is removed from the list and the next candidate of the list is processed until no candidates remain. The processing of the other lists is similar except that for the left candidate list the sponsoring element is the right neighbor for the top list it is the bottom neighbor and so forth. For the top and bottom lists the right and left neighbors do not have to be added to the respective lists. If a candidate of the top candidate list is moved its top, front and back neighbors are added for example. For the front and back list only the non sponsoring neighbor has to be added to the other lists. If a back candidate is moved only its back neighbor is added to the candidate list for example.

Performance

S. F. F. Gibson identifies three reasons for the speed of the algorithm:

1. each element in the object is considered at most once for each deformation

2. each element is compared to only one neighbor (its sponsoring neighbor) to determine if and how it must be moved
3. the deformation is propagated outwards from the selected point and the propagation is terminated as soon as possible

While 1 and 3 follow directly from the way the candidate lists are processed point 2 results from the following theorem

> "In the 3D ChainMail algorithm, each element can be compared to a single neighbor when the object has constant deformation limits throughout its volume." [Gibson '97]

The proof of this theorem can be found in [Gibson '97] on page 5.

The ChainMail Algorithm can also model elastic relaxation of an object. It is done similar to the deformation but instead of deformation parameters relaxation parameters are defined to check if an element meets the relaxation constraints with its neighbors.

The methods for visualization of the object include the following:

- point cloud: visualizing all elements of the object
- surface points: showing only the surface elements
- surface mesh: showing a mesh extracted from the surface elements

S. F. F. Gibson presented a fast algorithm capable of modeling rigid, deformable, elastic and plastic objects with hundreds of thousands of elements through simple calculations on a large number of elements.

3.3 Enhanced ChainMail [Schill et al. '98]

Problem of Original ChainMail Algorithm

Although, the original ChainMail algorithm does support anisotropic data it does not allow inhomogeneous data. This means that all chain regions within a *ChainMail object* have to be equal. This is a huge drawback because the modeled objects often consist of different material with different properties. A human head for example consists of softer tissue such as skin or muscle which deforms easily while it also consists of harder tissue such as bone which does not deform. Hence, it is desirable to define chain regions with different shapes for different types of tissue within a single *ChainMail object*.

The problem that occurs when doing this is that the theorem the speed of the ChainMail algorithm is based on demands constant deformation limits throughout the *ChainMail object*. Hence, if this requirement is not satisfied there are two possible outcomes. First, the elements have to be compared more than just once to satisfy

the deformation constraints which results in a heavy increase of computation time. Or second, the elements are only processed once and thus inconsistencies, such as holes, within the *ChainMail object* are created.

Sound Wave Approach

M. A. Schill et al. [Schill et al. '98] present an interesting approach to this problem. They interpret the ChainMail algorithm as the travel of sound through the object.

In their interpretation of the algorithm, information related to the deformation is the key concept. The initial move creates the initial information about the deformation. This information is then propagated through the object by comparing two neighbors. Upon adjusting neighbors to meet the deformation constraints the amount of information is decreased. Hence, the information is propagated through the object until it is fully consumed. This propagation of information is equated with the propagation of sound in the Enhanced ChainMail algorithm.

The authors studied how sound is propagated through an inhomogeneous object and applied this knowledge to the ChainMail algorithm: a sound wave travels faster through stiffer material. Hence, the deformation information should travel faster through stiffer material too. Stiffness is determined by the material properties between two neighbors in the Enhanced ChainMail algorithm and influences the order in which elements are processed. The order of processing defines where and how fast the information is propagated. This observation is used to make the speed of the deformation information propagation dependent on the material properties.

Enhanced Algorithm

Instead of six unordered candidate lists which are processed in term on a first moved first served basis they only use one ordered list. The criterion for ordering the elements in the list is the degree of constraint violation. The bigger the violation the earlier the element is processed to make sure that the information is propagated appropriately. Hence, the element with the biggest violation is always processed first. For two horizontal neighbor elements with the positions v_1 and v_2 the constraint violation is calculated as follows:

$$amount\,of\,constraint\,violation = distance(v_1, v_2) - max_{dx}$$

The use of an ordered list increases the computational cost for inserting an element depending on the implementation of the list used by $O(n)$ to $O(ld(n))$ which is still fast enough to produce interactive frame rates because all other benefits such as the "each element processed at once most" rule are still intact.

3.4 Problems

The Enhanced ChainMail algorithm is a capable application for soft tissue deformation. It can model volumetric data from objects with anisotropic and inhomogeneous material properties. However, this work can not be applied "as is" to the current problem of extending the STEPS system by adding support for deformations. The problems are described in this section.

3.4.1 Visualization

Most of the problems are rooted in the fact, that the STEPS system directly visualizes the volume using a ray casting algorithm [Neubauer '01] which has to be reused. The ray caster can not be used to directly render the *ChainMail object*. First of all, the ray caster assumes that all voxels lie in a rectilinear three dimensional grid with constant distances between them along each axis for performance reasons. If a *ChainMail object* is deformed, the elements usually do not satisfy this assumption.

Another requirement is the fact that the volume data has to show the deformation too. In the STEPS application a user has four views. One view shows the three dimensional virtual reality and the other three views show a cutting plane of the multiplanar reconstruction the so called section views. These section views have to reflect the deformation too.

Hence, it is necessary to map the *ChainMail object* to the volume. First of all, this is because the ray caster visualizes the volume. Hence, if the volume is deformed the ray caster can be applied without further changes and shows the deformation in the virtual view. The same applies for the section views which directly render the volume data too.

3.4.2 Large Amount of Data

The second problem arises from the huge size of the data. A typical computer tomography data set contains $512 \times 512 \times 64$ voxels. This means that the complete *ChainMail object* would consist of over 16 million elements.

Estimated Memory Usage

An estimate for the memory used shows the problem. An initialized element requires 224 bytes and a moved element requires 256 bytes because the memory to save the last position is required. These values were acquired using a Java VM 1.4.2 on a Windows XP platform. Java is used because STEPS is programmed in Java.

In Java there is no requirement for a sizeof() method such as in C and the memory an object requires may vary for different implementations of the Java Virtual Machine. The only methods available are Runtime.totalMemory() which returns the

total memory available to the Virtual Machine Runtime.freeMemory() which returns the free memory left in the Java Virtual machine and finally Runtime.maxMemory() which returns the maximum memory the Java Virtual machine will attempt to use.

Determining the Size of an Element

Therefore, the size of the ChainMail element data structure was determined by creating a large number of elements and examining the memory usage before and after the elements were created. To minimize the influence of overhead and objects that were not collected by the garbage collector 150000 objects were created. The current memory usage is given as:

$$usage = totalMemory - freeMemory$$

The variable $before$ stores the memory usage before the elements are created and $after$ stores the memory usage afterwards. Hence, the size for a single element is given as $size = (after - before)/150000$.

Overall Memory Requirement Estimation

A typical volume contains $512 * 512 * 64 = 2^{24}$ elements hence the memory used to model the entire volume with a single *ChainMail object* is $2^{24} * 256 = 2^{32}$ bytes (approximately 4 GB). The resources available are not sufficient for this amount of memory required.

Not being able to load the entire *ChainMail object* into memory is a general problem. Because the ChainMail algorithm calculates local deformations it is sufficient to only load the necessary parts. Therefore, it is necessary to implement a memory management in the current application which allow to hold only those parts in memory which are required for the calculation.

3.4.3 Interaction

Interaction with the object is another problem. In the original algorithm, the user selects an element and moves it to start the deformation. In our application the user can penetrate the object's surface with the endoscope at any point. He does not have any knowledge about where the ChainMail elements are. Hence, it is very unlikely that the user will precisely hit a single ChainMail element to initiate the deformation as it is required by the ChainMail algorithm.

The endoscope is rather likely to penetrate between the ChainMail elements. Therefore, it is necessary to introduce a new type of initiating the deformation.

The second reason why a different way for interaction is required arises from the fact that the endoscope has a certain diameter. This means that the intersection occurs in a circular area and not in a single point. To correctly model this case multiple elements must be moved to initiate the deformation.

3.4.4 Data Structure – Z–Scale

The so called z–scale of the volume data presents another challenge because it causes a distortion of the image along the z–axis. The z–scale is a measure for the distance between two voxels along the z–axis. This distance is constant for a single data set but varies between different data sets. The effect of the distortion of the deformation is proportional to the z–scale. An example image that shows the effect for a z–scale of 2.2 is shown is figure 3.5.

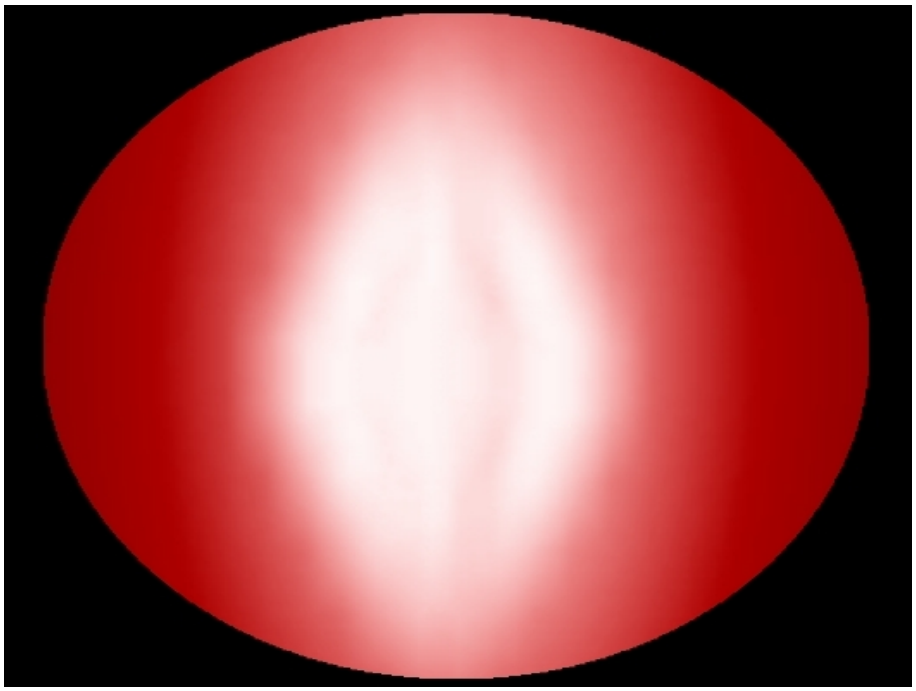


Figure 3.5: This picture was taken from a volume with a z-scale of 2.2. The shown deformation appears approximately twice as big along the z–axis than along the x–axis.

To ease this effect the shape of the chain regions and the calculation of the movement and dragging of the ChainMail elements are adjusted in regard of the z–scale.

3.4.5 Physical Plausibility

Additionally, to the shape of the deformation the global behavior should also be physically plausible. It should not be possible to move bone tissue or to deform the tissue by an unnaturally large amount. The shape of the deformation is governed by the chain region constraints. However, the original algorithm does not offer a way to model global constraints.

3.4.6 Relaxation

After the deformation step the original ChainMail algorithm allows a relaxation of the object. However, it does not describe the relaxation algorithm nor does it define circumstances or requirements for such a relaxation algorithm. The authors do point out that the calculation of the relaxation is a very time consuming process and should be calculated in a separate thread. This thread shall work during the idle time of the processor because the performance of the deformation calculation would otherwise be affected. Although, a relaxation of the tissue would make the simulation more plausible and enhance the experience of the user this work does not implement a relaxation for the following reason.

High Computational Cost

The benefit of the quality improvement for the visual effect gained by a relaxation does not justify risking interactivity. The relaxation itself is a very time consuming process. However, each relaxation step would also require the mapping of the relaxed object to the volume because it is done in a separate thread. Hence, the effort of the entire application would approximately double. Another thing to consider is that, after the relaxation the eye-point of the endoscope must still be outside of the tissue. Therefore, the relaxation algorithm would have to calculate iso values in the vicinity of the eye point to check if the calculated relaxation meets this requirement. This check would require the use of the ray caster because the iso surface is dependent on the threshold value set and can only be performed at the end of the calculations. Finally, an endoscope leaves an imprint on the deformed tissue. Hence, the relaxation must not fully relax the object but leave an amount of deformation intact. For these reasons, implementing the relaxation is a very complex task which is out of scope for the present work.

3.4.7 Conclusion

Possible solutions and the solutions chosen to be implemented to the problems described in this section,

- visualization: the required visualization method using a ray caster is currently not supported and the three section views have to reflect the changes made by the deformation.
- large amount of data: need for memory management.
- interaction: endoscope does not directly interact with ChainMail elements but interacts with the rendered surface.
- z–scale: causes a distortion effect because the distance between the voxels along the z–axis is different than their distance along the other two axes.
- physical plausibility: need for global constraints.

- relaxation: the computational cost are too high to implement relaxation in our algorithm.

as well as a discussion regarding the pros and cons of the different approaches are presented in the next section 3.5.

3.5 Solutions

3.5.1 Mapping

Why Mapping is Necessary

One way to avoid mapping the object to the volume as described in the latter section would be to implement a new ray caster that works on non rectilinear grids. But for performance reasons and memory requirements this approach is unfeasible and the current first hit based ray caster of the STEPS system has to be used. Additionally, the section views need to be updated to reflect the changes which requires mapping the object to the volume. Hence, the original volume data needs to be updated.

Volume Based Mapping

We propose an algorithm to map the deformed *ChainMail object* directly to the volume data instead of mapping an object or a surface extracted from the *ChainMail object*. It would be very time consuming to extract object or surface information form the *ChainMail object* because the object's elements represent the voxel values of the volume and not the real object. Hence, the loaded elements cannot be interpreted as a point cloud of the object and the boundaries of the object cannot be considered to represent the surface.

The volume data used for this specific application supports voxel values between -1000 and 3095. These values are also known as *Hounsfield Units* (HU). The corresponding *Hounsfield Scale* assigns *Hounsfield Units* to different body tissues [O.Ennemoser et al. '86]. This interval is internally mapped to the interval $0 - 4095$.

Reusability of STEPS Components

The mapping also addresses the section view problem because it updates the volume data. Another benefit is that the currently implemented ray caster can be used without further changes and the possibility for interactive threshold changes remains intact. The ray caster has no knowledge of the mapping process because it directly visualizes the volume data.

The mapping algorithm consists of two parts. The first part handles the mapping from the volume to the object and the second part is responsible for mapping the object to the volume.

Mapping the Volume to the Object

The volume data is stored in a rectilinear grid where the volume voxels are located at the intersection points of the grid lines. In the *ChainMail object* each element represents a voxel of the volume. Additionally, the mapper only maps to not deformed objects and the coordinates of the ChainMail elements and the volume data voxels are the same for not deformed objects. Hence, it is sufficient to copy the voxel value from the volume to the newly created ChainMail element with the same coordinates.

Mapping the Object to the Volume

Mapping a deformed object back to the volume presents a bigger challenge. The quality of the visual effect of the deformation depends heavily on the performance of the mapping algorithm. Another thing to consider is the requirement for interactivity. Thus, the mapping algorithm needs to be fast enough to meet this requirement.

First of all it is necessary to identify which voxels of the volume may have changed due to the deformation and need to be recalculated. Which volume voxels are affected is dependent on the algorithm used for the mapping.

First Approach – ROI based Mapping

The first approach we implemented to update the volume data was to define a region of influence around the volume voxel positions. The region of influence has the shape of a square or a cube in 2D and 3D respectively with a side length of 1. Please see figure 3.6 for a sample 2D region of influence.

The middle of such a region of influence is the corresponding volume voxel position. For example the region of interest for the voxel at (3/3) is given as (2.5/2.5), (3.5/2.5), (2.5/3.5) and (3.5/3.5). All elements inside this region of interest are taken into account for the recalculation while elements outside of this region are discarded.

Because the calculation of the new voxel value is dependent on the distance and the value of the surrounding ChainMail elements each moved element may have changed the value of a volume voxel. These affected volume voxels need to be recalculated. Affected voxels are those voxels that are part of the volume grid element that encloses the last position or the current position of a moved element.

A volume grid element in the 2D case consist of 4 voxels that form a square of size 1 (for example (2/2), (2/3), (3/2) and (3/3)). Please see figure 3.7 for a sample 2D grid element.

In the 3D a grid element is defined as 8 volume voxels building a cube with a volume size of 1. It is necessary to recalculate the voxels for both positions because the movement away from the old position causes a change in the old neighborhood whereas the movement to the new position also causes change. For example, in the 2D case if an element is moved from (2.3/3.2) to (2.4/3.7) the enclosing grid element

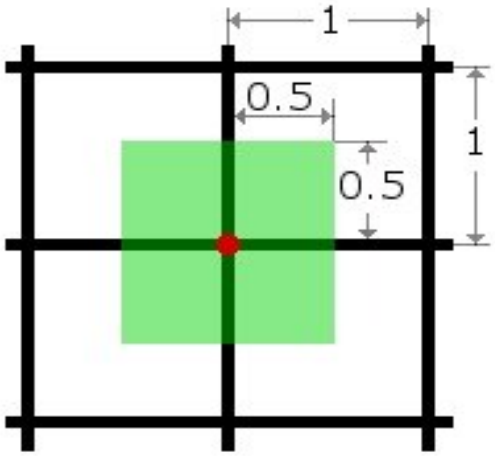


Figure 3.6: A sample region of influence in 2D. The region is a square with the same size as a a grid element. The center of the region is the position of the corresponding volume voxel depicted by the red dot.

for current and last position is the same. Hence, the volume voxels at (2/3), (2/4), (3/3) and (3/4) need to be updated. If the element was moved to (2.4/4.1) then the voxels at (3/5) and (2/5) also need to be recalculated.

The calculation of the new volume values for the affected voxels starts with collecting all object elements that lie inside the region of interest. For each element in the collection the contributing element value is calculated. This is done by first calculating the distance of the current element position v_{curr} to the volume voxel position vol_{pos} along each axis.

$$d = |vol_{pos} - v_{curr}|$$

The contributing element value $contrib$ is then given as

$$contrib = element_{value} * (1 - d_x) * (1 - d_y) * (1 - d_z)$$

This ensures that object elements which lie exactly on a volume voxel position contribute the complete value because the distance vector is the zero vector. Additionally, the value contributed decreases with the distance of a voxel. Finally, for n contributing elements the respective contributed values are summed up to calculate the new volume voxel value.

$$value_{new} = \sum_{i=1}^{n} contrib_i$$

This approach is fast but inflexible and likely to create artefacts. As an example that leads to an artefact imagine a chain region which allows a maximum distance

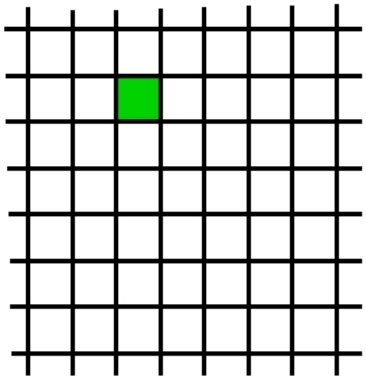


Figure 3.7: The green region shows a single grid element in the 2D case.

along the x–axis of 1.2. The object is not deformed hence, all object elements have the same coordinates as the volume voxels. If an element at position *pos* is initially moved by the move vector $(0.6, 0, 0)$ its neighbor will be moved by $(0.4, 0, 0)$ to satisfy the chain region constraint along the x–axis. This leads to a hole at the position *pos* because there are no contributing elements left in the region of influence. This case is depicted in figure 3.8.

The new voxel value results as 0 which is not correct. Additionally, the resulting values may exceed the valid interval defined by the *Hounsfield Scale*.

Tri–linear Interpolation based Mapping

Tri–linear interpolation assumes that the eight enclosing elements for a position form a parallelepiped. A parallelepiped consists of six pairwise congruent and parallel surfaces. Unfortunately, this assumption does not hold for deformed objects. Therefore, we could not use tri–linear interpolation to calculate the new density values.

Hence, we developed another method for mapping the *ChainMail object* to the volume using barycentric coordinates.

Barycentric Coordinates based Mapping

Because of the problems with the region of influence based approach we introduced a new way for calculating the voxel values using barycentric coordinates. The basic flow of the algorithm is to find the eight elements which form a cube that encloses the volume position to be updated. This cube is then split to tetrahedrons. Then the barycentric coordinates for the volume voxel position in the enclosing tetrahedron are calculated and used to generate the new density value for the voxel.

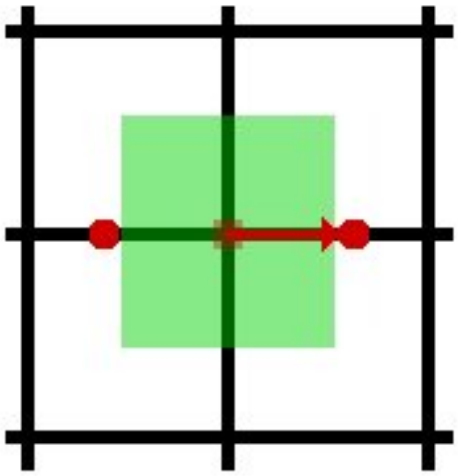


Figure 3.8: After the element is moved and its neighbor is dragged, the region of influence remains empty.

Because we use barycentric coordinates the newly calculated values stay within the valid interval and artefacts are avoided.

The details of this approach are presented as part of the entire algorithm in section 3.6.3.

Deformation Near Volume Boundaries

The mapping is also capable of extending the *ChainMail object* over the boundaries of the volume data set to support deformations near the volume boundaries.

3.5.2 Memory Management

Idea

The desired memory management is supported by our approach because the deformation is locally propagated through the object. Therefore, it is possible to start with loading the part of the volume into memory which holds the elements involved for the initial movement and then as the deformation propagates load the missing parts on demand. The existing object is extended by the loaded parts through connecting them to it. Also for subsequent deformations the missing parts of the volume at the boundaries of the *ChainMail object* are loaded and connected. Upon the initiation of a new deformation outside the currently loaded *ChainMail object* a new object is created and the old object is removed from memory.

Drawbacks of Clearing the Old Object

This approach offers one major drawback because it is possible to elude the *movement constraint* described in section 3.5.5. The purpose of this constraint is to avoid unnaturally large deformations. When the object is cleared, the deformation stays intact but the information about the movement of the elements is lost. Hence, if the user starts a new deformation at the already deformed tissue a new *ChainMail object* is created where the movement information of the volume is out of sync with the movement information of the object. This allows the user to deform the volume ignoring the movement constraint.

Possible Solutions to the Drawbacks

Several strategies can be applied to avoid this. The first one is not to clear *ChainMail objects*. The problem that occurs from this approach is that at some point two or more *ChainMail objects* need to be merged. To avoid merging, the object can also be extended step by step until the new point is loaded but this leads to huge objects and requires a lot of computation time. Another way to avoid this would be to store the movement information before clearing the object. This can be equated to dumping the object which is also very time consuming. The problem of merging however cannot be avoided by this approach because the dumped objects may also overlap.

A completely different approach would be to revert the deformation when clearing an object. This means destroying the movement information in the volume too. Hence, the movement information in the volume and a newly created *ChainMail object* are in sync. However, the deformation has a plastic component. The endoscope leaves an imprint in the deformed tissue. Hence, a complete reversion of the volume would not be physically plausible. For performance reasons none of these approaches were applied in the current implementation.

3.5.3 Interaction – Multi Move

The interaction between the endoscope and the surface is vital for the realism of the system. Because the endoscope penetrates at a random surface position and does not interact directly with the ChainMail elements a certain ChainMail element needs to be selected to initiate the movement. At first, the nearest element was selected for the movement but his procedure does not produce physical plausible results.

Idea

To model the endoscope's diameter and maximize the probability that the eye point lies outside of the tissue after the deformation the eight elements which enclose the calculated intersection point are initially moved. The part of the iso surface enclosed

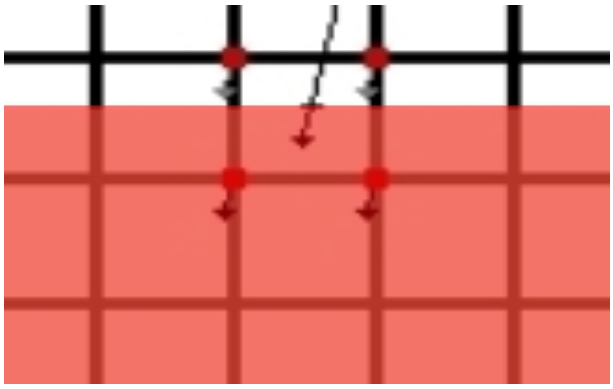


Figure 3.9: The eye point of the endoscope lies inside the tissue

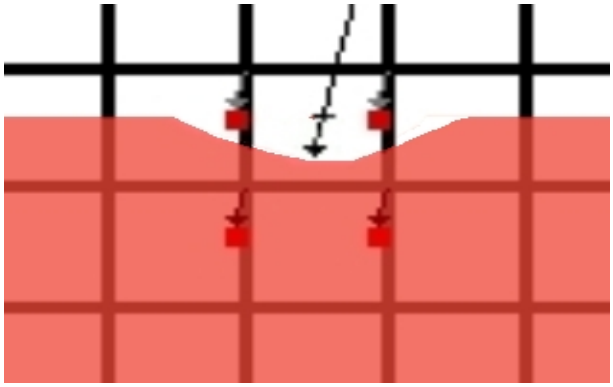


Figure 3.10: The enclosing elements are moved

by the eight elements is approximately moved the same way as the elements. It is not moved exactly the same way because the shape of the iso surface also depends on the neighbors of the eight moved elements. However, using this procedure the endoscope is likely to lie outside the tissue after the deformation (see figures 3.9 and 3.10).

The direction of the endoscope defines the direction of the movement. The length of the movement is given by the distance between the eye–point of the endoscope and the intersection point. It is scaled by a configurable scale factor to adjust the depth and size of the deformations.

First Approach – Moving each Voxel Separately

However, in the original ChainMail algorithm a deformation is initiated by moving a single element. In contrast, our work proposes to initiate the deformation by moving eight elements at once.

Hence, a first approach to the *multimove* problem would be to initiate eight deformations, one for each element. This approach requires a lot of computation time because eight separate deformations need to be calculated for each collision. Additionally, it is very likely that the first moved element causes its neighbors to be dragged by a certain length $length_{drag}$. Therefore, the length of the original movement vector $length_{movement}$ must be adjusted. If it is be applied to the seven subsequently moved elements without adjustment their movement would be incorrect given as: $length_{movement} + length_{drag}$.

Moving All Initial Elements at Once

Instead, it is easier to move all eight elements at once and add the neighbors of each initially moved element which are not initially moved elements themselves to the candidate list. Because all initial elements are moved the same distance and direction they do not have to be added to the candidate list since their relative position does not change and hence there chain region constraints are not violated.

Processing Elements Multiple Times

From here, the algorithm has to be adjusted to avoid that an elements gets processed more than once. If the ChainMail algorithm is applied "as is" then the elements would get processed multiple times because there are eight elements that start the movement. For example assume the two elements v_{top} and v_{bottom} where v_{top} is the top neighbor of v_{bottom} are moved to initiate the deformation. Following the 2D algorithm their neighbors are added to the candidate lists. Examining the situation for the right neighbors where r_{bottom} is the right neighbor of v_{bottom} and r_{top} is the right neighbor of v_{top} already shows the problem. r_{bottom} and r_{top} were both added to the candidates list after the initial movement. If r_{top} gets processed and is moved to satisfy the constraints its top, right and bottom neighbors are added to the candidates list. But since r_{bottom} which is the bottom neighbor of r_{top} is already entered in the candidates list it gets processed more than once. Then, if r_{bottom} also needs to be moved r_{top} will be reentered in the list of candidates. Please see figure 3.11.

Hence, the speed benefit of the ChainMail algorithm is lost.

Adjusting the Add to Candidate List Rule

To avoid the multiple processing of elements the way the neighbors are added to the candidates list is changed. For the given example the problem occurs if r_{top} adds r_{bottom} to the candidates list and vice versa. If this is avoided the problem does not occur in the given example. However, this has to assured for all right neighbors. The right neighbor of r_{top} may not add its bottom neighbor and so forth.

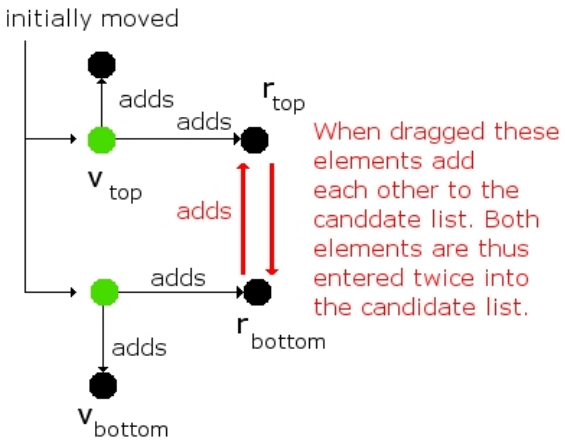


Figure 3.11: This figure shows the case where the elements r_{top} and r_{bottom} get added to the candidate list twice. They are added by the initial element and they add each other to the list too if dragged.

This is done by assigning three *ignore direction flags*, one for each direction, to each element. This flag stores the direction in which no neighbors may be added. The information is initially created by the initially moved elements. The directions to ignore are equal to the directions in which the other initially moved neighbors lie. To propagate the ignore directions through the object during the dragging step the sponsor passes its ignore direction flags to its candidates. If a candidate becomes a sponsor, then it also passes the flags to its candidates and so forth.

Please refer to figure 3.12.

Possible Inconsistencies

This procedure equals cutting the object (in the example the object was cut in the along the horizontal axis) and calculating the deformation in the two parts independently. This presents another problem because in the inhomogeneous case the chain region constraints may be violated along the cut. In the given example r_{bottom} and r_{top} are never checked against each other. In the homogeneous case all chain regions are equal since the initially moved elements are all moved the same way the dragged elements are dragged the same distance too and therefore satisfy the constraints. This observation does not apply to the inhomogeneous case because the distance the dragged elements are moved varies depending on the shape of the chain region. Hence, a violation of the chain region constraints may occur. For example, if the direction of the deformation is straight left and the material on the top side of the cut is softer than the material on the bottom side of the cut the elements on the

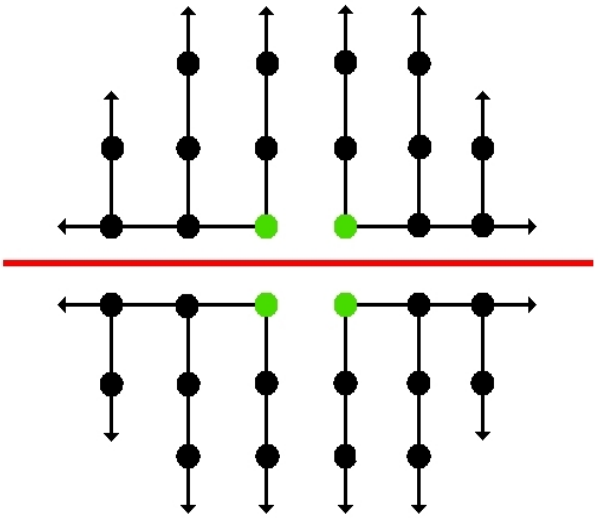


Figure 3.12: This figure shows the effect of the adjustment to the *add candidate rule*. The object is cut along the horizontal axis depicted by the line an no neighbors along this line are added to the candidate list. The deformation propagates separately through the two parts of the object. The green elements are the initially moved elements which do not add their initially moved neighbors either. The arrows show the directions in which neighbors are added to the candidate list.

bottom side may be moved further than the elements on the top side. This may lead to a shear constraint violation of the elements along the cut line (see figure 3.13).

However, the shape of the chain regions that are used in this application do not differ by a huge amount. Thus, it is unlikely that a chain region constraint violation occurs. Most importantly, no visual artefacts were produced from the constraint violation problem.

Misplaced Deformation

Another problem with this approach is that the center of the deformation does not appear to be the point of impact.

This effect occurs if the point of impact is near a voxel or near a side of the initially moved cube. The center of the moved cube appears as the center of the movement. Hence, if the point of impact is far away the impression of a misplaced deformation is created. Please see figure 3.14 for a 2D example.

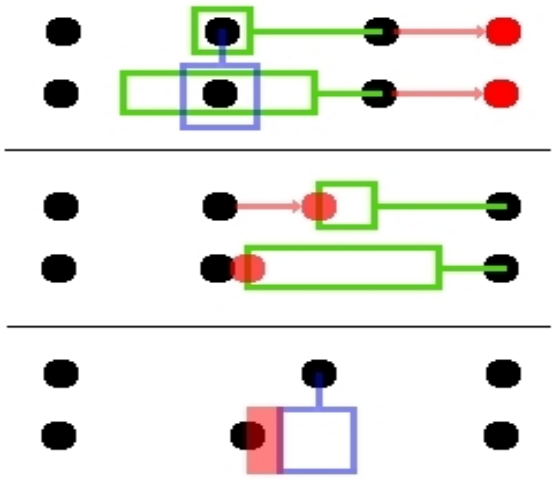


Figure 3.13: The elements along the top row are dragged further than the elements along the bottom row. This results in a shear constraint violation marked by the red region.

Proposed Solution to the Misplacement Effect

To soothe this effect, it was proposed to move each of the eight cube elements individually depending on their distance from the point of impact. At first this seems to be a promising approach. However, the problems of moving the elements appropriately in accordance with the chain mail constraints becomes a complex task. It requires to check all constraints for all eight elements in regard to the point of impact. The problem becomes more complex for already deformed elements. Additionally, due to the chain region constraints, movement of the individual elements may not differ by a great amount. For example, a chain region offset of 0.2 allows a maximum difference of 0.2 between the movement of each element for an initialized object which is not enough to counter the misplacement effect.

Finally, the effect shows very well on even surfaces but not on uneven surfaces as they are modeled in the STEPS application. Hence, the approach of moving the elements depending on their relative position to the point of impact was disregarded.

3.5.4 Z–Scale Adjustment of Region Constraints

Motivation

A distortion is caused by the fact that the voxel distance along the z–axis is different than along the other two axes. This ratio is called z–scale and differs for various volume data sets. All movements propagated along the z–axis appear scaled by the z–scale factor. This distorts the visualization of the deformation which looks stretched along the z–axis (see figure 3.5).

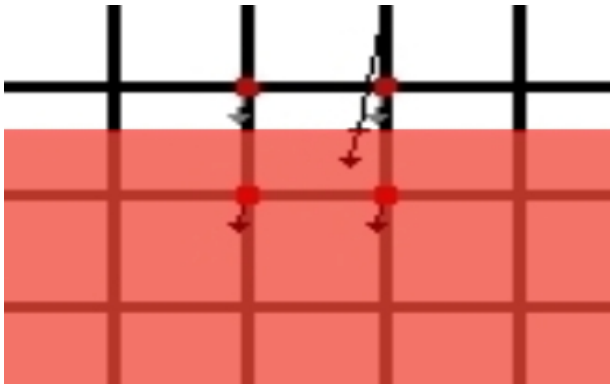


Figure 3.14: The endoscope penetrates near a surface of the enclosing cube. However, the deformation looks exactly as figure 3.10 giving the impression that the deformation is misplaced.

Ratio of Information Propagated

The problem occurs because the ratio of information propagated along the z–axis and the other two axes is $zscale : 1$ because the voxels are lying further apart along the z–scale. If information is passed to a neighbor along the x-axis the distance the information travels equates one distance unit whereas the distance the information travels equates z–scale times distant unit along the z–axis.

Idea

To solve the problem we used the analogy of a sound wave traveling through the material introduced by *M. A. Schill et al.*.

In the case of the z–scale the information propagated along the z–axis is exaggerated by the z–scale factor. Hence, the amount of information propagated along the z–axis has to be divided by the z–scale. There are two channels through which the information is passed along the z–axis. The first channel we called *direct channel* propagates the movement information handled by the *min* and *max* constraints of the chain region. The second channel we called *shear channel* propagates the movement information derived from the *shear* constraints. In this case it is the shear along the x–axis and along the y–axis.

Adjusting the Propagation along the Direct Channel

Hence, to soothe the effect two adjustments are applied. In the first adjustment we scale the move vector along the z–axis by $1/zscale$. The valid ratio of information propagated along the direct channel of the z–axis and the other two axes is restored.

Adjusting the Propagation along the Shear Channel

To ease the effect along the shear channel we implemented a second adjustment affecting the chain regions. The deformation information is also propagated along in the z–axis through the shear constraints if the sponsoring element is a neighbor along the z–axis and the shear constraints along the x–axis or y–axis are violated. This results in a drag of a neighbor along the z–axis and thus, to a propagation of the information along the z–axis. Similar to the direct channel the information has to be divided by the z–scale factor. We do this by extending the shear constraints for the x shear and the y shear regions in z-direction. Our shears used for checking the shear constraints along the z–axis are hence:

$$zshear_x = zscale * shear_x$$

and

$$zshear_y = zscale * shear_y$$

By adapting the shears in this manner neighbors are dragged later and a smaller distance along the z–axis. This means that the information is consumed faster by the factor z–scale along the z–axis. Therefore, the ratio of information propagated along the shear channels is restored.

Adapting the shear constraints of the chain regions and the z component of the initial move vector to the z–scale restores the original ratio of the information propagated along different axes and soothes the visual distorting effect of the z–scale factor.

3.5.5 Global Constraints

To ensure physically plausible results we introduce two global constraints.

Movement Constraint

The *movement constraint* avoids an unnaturally big deformation of the tissue. In the real world the movement of the tissue is restricted. It is not possible to replace parts of tissue by a large distance. This constraint is modeled by restricting the allowed maximum distance an element may move from its original position. A violation occurs if the length of the initial movement is bigger than the movement constraint or a subsequent deformation causes an element to be moved a bigger distance than allowed. Our approach makes the size of the movement constraint $mc_{element}$ for a single element e dependent on the density value of the element $e_{density}$ and the configurable global movement constraint mc_{global}. Bigger density values are interpreted as stiffer material with a more rigid movement constraint. We calculated the movement constraint mc follows:

$$mc_{element} = (1 - (e_{density}/4096)^2) * mc_{global}$$

Bone Constraint

The *bone constraint* is introduced in the form of a bone threshold. Because bone tissue does not deform it is necessary to identify bone elements and stop the deformation process if such a bone element is about to be moved. If an element is tried to be moved which has a value that is bigger than the bone threshold the constraint is violated. The *Hounsfield Scale* associates values around $100HU$ with spongy bone tissue and the hard bone is associated with values around $1000HU$.

Violation Handling

If one of these constraints is violated the calculation of the deformation is stalled and the movement is reverted. Then the deformation is retried with a smaller initial step size.

A detailed description of our adapted version of the ChainMail algorithm supporting *multimove* and z–scale as well as the mapping and memory management algorithms are presented in the next section.

3.6 Algorithm Outline

3.6.1 Component Overview

As described earlier the *Divod ChainMail* algorithm consists of three components:

- Deformation: calculating the deformation
- Mapping: mapping the *ChainMail object* from and to the volume
- Memory Management: managing the memory loading required parts and releasing unnecessary parts

The mapping and the memory management are tightly coupled together. Whenever a portion of the volume needs to be added to the *ChainMail object* both components are involved. First, a new piece of the *ChainMail object* is created to hold the required part of the volume. In the second step the mapper maps the volume to the newly created piece. Finally, it is integrated into the existing object. The request for such a load operation is issued by the component calculating the deformation. It is issued if the calculation requires an element that is currently not loaded (for example if an element that is not loaded needs to be moved).

Hence, the interaction of the three components looks as follows (see figure 3.15).

A detailed description of the individual components is presented in the sections 3.6.2, 3.6.3 and 3.6.4.

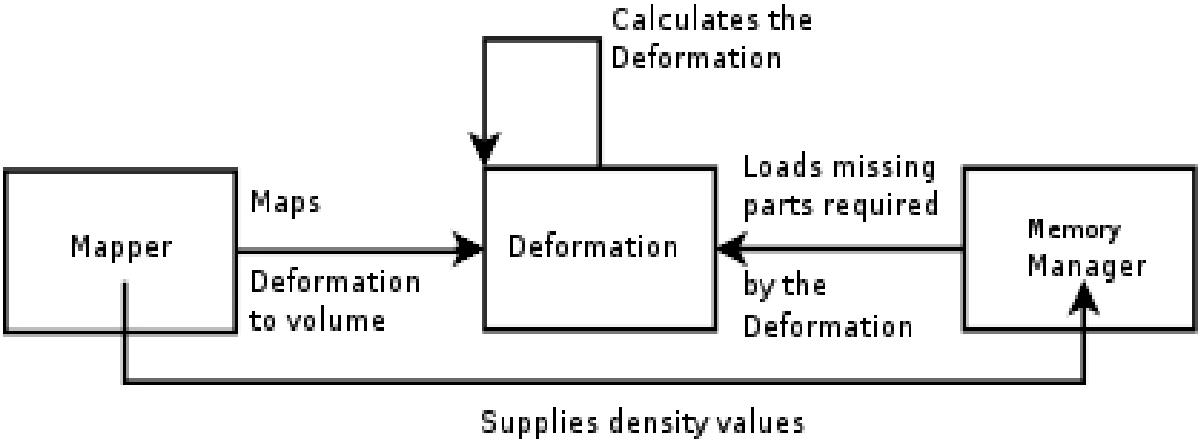


Figure 3.15: Interaction of the individual components

3.6.2 Deformation

Basic Flow

The next pseudo code listing presented in figure 3.16 shows the basic flow of the deformation algorithm.

```
adjustToZScale(movement.vector);

// check if the object encloses the collision
if (positionIsLoaded(movement.position)) {
  // if not, then the object is cleared and a new deformation is started
  clearObject();
  // load the part of the volume that encloses the position
  MemoryManager.loadChunk(movement.position);
}

// find the eight enclosing elments to initiate the
// deformation with, it might be necessary to load
// chunks if not all eight enclosing elements are loaded
initialElements = findEnclosingElements(movement.position);

// move the eight elements and add their neighbors to the candidate list
multimove(initialElements);

// process all candidates in the candidate list
// if necessary missing neighbors are loaded by the
// MemoryManager and connected to the object.
processCandidateList();
```

Figure 3.16: Flow of Deformation Algorithm.

```
lastPos = STEPS.lastEyePoint;
virtualPos = STEPS.currentEyePoint;
step = virtualPos - lastPos;
step.scale(stepsize);
lastVirtualPos = virtualPos;
while(isInsideTissue(virtualPos)) {
lastVirtualPos = virtualPos;
  virtualPos.sub(step);
}
intersection = 0.5 * (vitualPos + lastVirtualPos);
force = STEPS.currentEyePoint - intersection;
object.performDeformation(intersection, force);
```

Figure 3.17: Algorithm for calculation of deformation information.

Collision Detection

The request for the deformation is issued from the STEPS system. It detects the collision of the endoscope with the iso surface and passes the necessary deformation information consisting of the collision's position and the force vector to the object.

Calculating the Intersection Point

STEPS stores the current and last position of the eye point. The direction of the force applied to the tissue is given from the old position of the eye point to the current position of the eye point.

At first the intersection point is assumed at the eye point of the endoscope which lies inside the tissue. This point is then repeatedly moved towards the last position of the eye point by a small step size until it lies outside of the tissue. Finally, the middle of the last two calculated points, one lies inside the tissue the other one outside of the tissue, becomes the starting point of the deformation. Please see figure 3.17 for a pseudo code listing.

Scaling the Force Vector

The deformation algorithm starts with scaling the z–component of the force vector by the z–scale. This is done to reestablish the valid ration of information propagated along the z–axis and the other two axis as described in 3.5.4.

Cases at the Beginning

For the given position of collision three cases may occur:

1. The object does contain *cubicles* and the enclosing elements for the given position are loaded: The calculation can proceed.
2. The object does not hold any *cubicles* : A request to load the *cubicle* that holds the desired position is issued to the memory manager. After loading the *cubicle* finding the enclosing cube is successfully retried and the calculation can proceed.
3. The object contains *cubicles* but the enclosing eight elements can not be found in the currently loaded elements: It is assumed that a new deformation is initiated and the memory is released by clearing the object. Now when retrying to find the enclosing elements the procedure is the same as in case 2.

Finding the Nearest Element

Now, the eight enclosing elements, we also call *enclosing cube*, for the given position are acquired. Finding the enclosing eight elements begins with finding the element which lies nearest to the requested position. Finding the nearest element is done by getting the top corner element of the *cubicle* which originally holds the requested position. Then, the distance of this element to the point is compared to the distances of its neighbor elements. If a neighbor is closer the process is repeated with the neighbor that is closest to the position and so forth. The iteration ends when no neighbor element is closer to the requested point. Please see figure 3.18 for a pseudo code listing.

During the search for the nearest element it might be necessary to load missing cubicles. This occurs if a neighbor has to be checked that is currently not loaded.

```
DeformVoxel findNearest(Position) {
  voxel = cubicle.getVoxels().getTopCornerVoxel();
  try {
    // calculate the distance
    act_dist = voxel.getDistance(position); // distance of actual element from
    do {
      // get the neighbor that is nearer
      newVoxel = voxel.getNearer(position);

      // calculate the distance of the nearer element
      new_dist = newVoxel.getDistance(position); // distance of currently processed neighbor
      if (act_dist > new_dist) {
        // the neighbor element is nearer. it is set
        // to the actual element and the process is repeated
        act_dist = new_dist;
        voxel = newVoxel;
      }
    } while (act_dist <= new_dist);
  } catch (PositionNotLoadedException pnle) {
    // the voxel.getNearer() method may throw a
    // PositionNotLoadedException in case a neighbor that could be
    // nearer is not loaded. In this case the missing cubicle is
    // loaded and the process retried.
    throw new PositionNotLoadedException(pnle.getMissingPosition());
  }
  return voxel;
}
```

Figure 3.18: Pseudo code listing of the find nearest element algorithm.

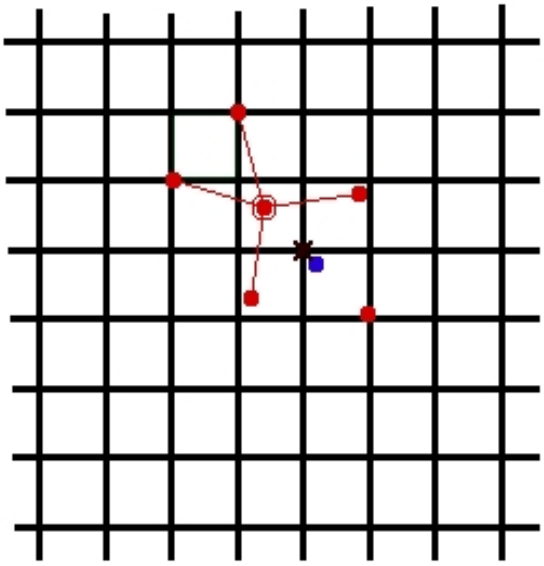


Figure 3.19: All neighbors of the encircled element are further away from the grid element marked with X than the encircled element. Hence, the search for the nearest element will finish without finding the blue element which is the correct nearest element.

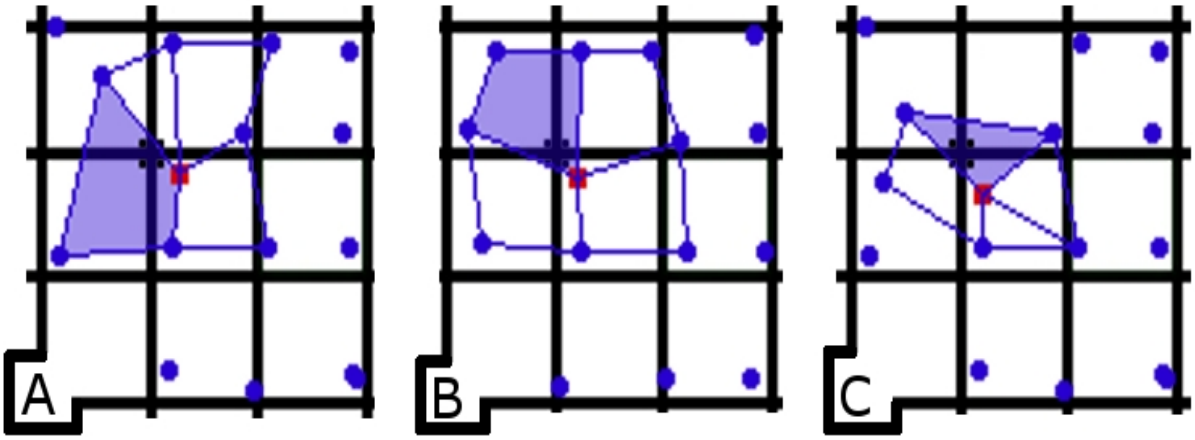


Figure 3.20: The nearest element must be part of the four enclosing elements for the grid position marked with a square. It is not possible to determine which neighbors are the enclosing neighbors from the relative position of the element to the grid position.
Picture A: the other two enclosing neighbors are bottom and left.
Picture B: the other two enclosing neighbors are top and left.
Picture C: the other two enclosing neighbors are top and right.
Only the case where the bottom and right neighbor are part of the enclose can be ruled out in this case.

This algorithm works very well for non deformed objects but is flawed when working on a deformed object. Consider the case that all neighbors of an element are further away from a given point but that the next neighbor of the neighbors is the nearest element. This case is depicted in figure 3.19. In this case the algorithm stops but does not find the nearest element to the given position. If this happens all elements of the current object are checked to find the nearest element.

The nearest element to a position is calculated to exploit the fact that such an element must be part of the enclosing cube because the neighborhood of the elements remains intact during a deformation. That means the left neighbor of an element always stays on the left side of this element and so on. Please refer to figure 3.20 for an example in 2d.

```
boolean checkEnclose(List cube,Point3d position) {

  foreach (tetrahedron = cube.splitToTetrahedrons()) {

    barCoor =
        calculateBarycentricCoordinates(tetrahedron, position);

    if (allComponentsBetweenZeroAndOne(barCoor)) {
    return true;
    }
  }
return false;
}
```

Figure 3.21: Pseudo code for checking if a point is enclosed by eight elements.

Finding the Enclosing Cube

To find the other seven elements, the eight cubes that can be formed with the nearest element as a corner element are generated. Each of these cubes is checked if it contains the position. Unfortunately, the relative position of the desired cube cannot be identified from comparing the nearest element to the requested position. If an element lies on the right, top side of the position the enclosing square (in the 2D case) can either include the top, right neighbor or the right bottom neighbor or the top, left neighbor. The only two neighbors that cannot be part of the enclosing square at the same time are the bottom and left neighbor. Hence, all possible cubes have to be checked. The check is done by splitting the cube to tetrahedrons and examining the resulting barycentric coordinates of the given point depending on the tetrahedrons. A detailed description of this procedure is given in section 3.6.3. If all components of the barycentric coordinate vector are between 0 and 1 inclusive the position lies inside the tetrahedron. A pseudo code listing is presented in figure 3.21.

```
void moveInitialCube(List elements) {

  // iterate through all elements
  foreach (v = cubeVoxels.next()) {
    getCubicle(v).addMovedVoxel(v);  // add to moved element list

    v.move(path); // move the element

    // add all not initially moved neighors to the candidate list
    // directions are left, right, top, bottom, front, back
    foreach (direction dir = directions.next()) {
      neighbor = v.getNeighborAt(dir);

      // find the element that was not moved
      if (neighbor.isInitiallyMoved()) {
        dir = -dir;
        neighbor = v.getNeighborAt(dir);
      }

      // set the ignore direction, neighbors in this
      // direction will not be added to the candidate list
      v.setIgnoreDeformationDirection(-dir);
      // Add to neighbor candidate list
      deformCandidates.addCandidateToList(v, neighborVoxel);
    }
  }
}
```

Figure 3.22: Listing of initial movement.

Moving the Initial Elements

After the eight enclosing elements for the position were found they are all marked as initially moved elements and moved by the deformation vector passed to the object from the STEPS system.

As laid out in section 3.5.3 it is necessary to avoid the multiple adding of the neighbors to the candidates list. Hence, each element holds an ignore deformation direction flag which tells the element which directions must not be added to the candidate list. Each sponsoring element passes this information on to the candidates is sponsors and so forth. The directions to ignore are first created for the initially moved elements which are the first sponsoring elements. For example, for the top, left, front of the initially moved cube the three ignoring directions are right back and bottom. Hence, only the neighbors at the top, left and front are added to the list of candidates. Please see figure 3.22.

Processing the Candidate List

After the initial elements were moved, the processing of the candidates list begins. The first candidate in the list, which is the candidate with the biggest constraint violation is processed. In the current implementation, the list saves entries of candidate sponsor pairs.

```
// calculating the necessary adjustment for  the min and max
// constraints of the left and right neighbors
if (Math.abs(candPos.x - sponPos.x) < minDx) {
  path.x = sponPos.x - candPos.x;
  if (candidate.getNeighborhood(sponsor) == left) {
    path.x += minDx; } else { path.x -= minDx;
  }
} else if (Math.abs(candPos.x - sponPos.x) > maxDx) {
  path.x = sponPos.x - candPos.x;
  if (candidate.getNeighborhood(sponsor) == left) {
    path.x += maxDx; } else { path.x -= maxDx;
  }
}

// calculating the necessary adjustment for the
// shear constraints along the y-axis
// of the left and tight neighbor
if (candPos.y - sponPos.y < -shearDy) {
  path.y = sponPos.y - shearDy - candPos.y;
} else if (candPos.y - sponPos.y > shearDy) {
  path.y = sponPos.y + shearDy - candPos.y;
}
// shear along z axis analogue

/* adjustment for z-scale */
// only the shears are adjusted
if (candPos.y - sponPos.y < -shearDy*zScale /* <-- adjustment */ ) {
  path.y = sponPos.y - shearDy*zScale - candPos.y;
} else if (candPos.y - sponPos.y > -shearDy*zScale /* <-- adjustment */) {
  path.y = sponPos.y + shearDy*zScale - candPos.y;
}
// shear along x axis analogue
```

Figure 3.23: Listing of the chain region constraint check.

Depending on the neighborhood the candidates position is checked against the chain region constraints of the sponsor (refer to figure 3.23). If the candidate is a horizontal (along x–axis) or vertical (along the y–axis) neighbor, the chain regions are applied without changes and the path to meet the chain region constraints is calculated. If the element is a neighbor along the z–axis then the chain region constraints need to be adjusted if the z–scale adjustment flag was set in the configuration. This is necessary because the visual effect of the deformation would be distorted along the z–scale. Please refer to section 3.4.4 for a description of the problem. To avoid the z–scale effect the shear along the x–axis and the y–axis is multiplied by the z–scale. This keeps the deformation from expanding faster along the z–axis than along the other axis.

```
/* sample adding a candidate along the x--axis */
// sponsor is on the left side
if (sponsoredBy == left) {

// add right, neighbor
addCandidateToList(
      sponsor.getRight(), right,
      sponsor.getViolationInDirection(right));

// add either top or bottom neighbor,
// the other neighbor will be added by another sponsor
  int direction = -sponsor.getIgnoreDeformationDirectionY();
  addCandidateToList(sponsor.getNeighborAt(direction),
      direction,
      sponsor.getViolationInDirection(direction));

// add either front or back neighbor
// the other neighbor will be added by another sponsor
  direction = -sponsor.getIgnoreDeformationDirectionZ();
  addCandidateToList(sponsor.getNeighborAt(direction),
      direction,
sponsor.getViolationInDirection(direction)); sponsor.getViolationInDirection(direction));


/* sample adding a candidate along the y--axis */
if (sponsoredBy == top) {

  // left and right neighbors do not need to be added

// add bottom, neighbor
addCandidateToList(
      sponsor.getBottom(), bottom,
      sponsor.getViolationInDirection(bottom));

// add either front or back neighbor
// the other neighbor will be added by another sponsor
  direction = -sponsor.getIgnoreDeformationDirectionZ();
  addCandidateToList(sponsor.getNeighborAt(direction),
      direction,
sponsor.getViolationInDirection(direction)); sponsor.getViolationInDirection(direction));

/* sample adding a candidate along the z--axis */
if (sponsoredBy == front) {

  // left, right, top, bottom neighbors do not need to be added

// add bottom, neighbor
addCandidateToList(
      sponsor.getBack(), back,
      sponsor.getViolationInDirection(back));
```

Figure 3.24: Listing of adding the neighbors to the candidate list.

A element is moved by calling the move() method and each moved element is added to the moved list of the *cubicle* the element belongs to. If an element is moved its neighbors which are not lying in an ignore direction are added to the candidate list as shown in figure 3.24.

With each adjustment the constraint violation decreases until all elements satisfy the chain region constraints and no further deformation is necessary.

Chain Region

In the current application the voxels have an initial distance of 1 to each neighbor. Because an initialized object must be valid the chain region's min_{dx}, min_{dy}, min_{dz} must be smaller or equal to 1 and the respective max_{dx}, max_{dy}, max_{dz} constraints must be bigger or equal to 1 while the shear constraints must be bigger or equal to 0.

To have the deformation propagate symmetrically through the object all chain regions we use in this implementation have the shape of cubes. Hence, they are configured using an offset value $offset$. This offset is added to 1 to calculate the max constraints and subtracted from 1 to calculate the min constraints. The values for shear constraints are set to $offset$.

To model inhomogeneous data the size of the chain region depends on the density value of the corresponding element. Because tissue with a higher density value is assumed to be stiffer the region is smaller for bigger values.

Finally, we propose to calculate the chain region constraints as follows. The values for the parameters of one type of constraints (max, min, $shear$) are the same (e.g. $max_{dx} = max_{dy} = max_{dz}$):

$$adjustment = offset * (voxel_{value}/4096)^2$$

$$max = 1 + offset - adjustment$$

$$min = 1 - offset + adjustment$$

$$shear = offset - adjustment$$

Since the adjustment increases with the value the size of the chain region decreases with higher values. Additionally, this approach ensures that the initial object is valid because the highest density value is 4095 and thus the adjustment is always smaller than the offset which lies between 0 and 1 exclusive.

The deformed object is then passed to the mapper which maps it to the volume. The algorithm for the mapping is described in the next section

```
// only moved elements cause a change of the volume data
List movedElements = DeformObject.getMovedElements();
foreach (movedElement = movedElements.next()) {
  // the positions to update are the voxels of the
  // voxel grid elements the moved element's current
  // and last position lie in.
  // there are 16 positions to calculate, 8 for the
  // current position, and 8 for the last position
  foreach(updateVoxelPos =
    movedElement.getVoxelPositionsToUpdate())
  {
    // to avoid multiple calculations, already calculated
    // values are stored in a hash map
    if (!hashMap.containsEntry(updateVoxelPos) {

      // in contrast to the find element method of the
      // deformation, this method does not need to
      // load missing parts of the volume
      cube = findEnclosingElements(updateVoxelPos);

      // find the enclosing tetrahedron to calculate
      // the barycentric coordinates of the poition to
      // update
      tet = getEnclosingTetrahedron(cube, updateVoxelPos);
      barCoords = getBarycentricCoordinates(tet, updateVoxelPos);

      // finally caluclate the new danisty value
      newDensityValue =
          tet.getElement(0).densityValue * barCoords.x +
          tet.getElement(1).densityValue * barCoords.y +
          tet.getElement(2).densityValue * barCoords.z +
          tet.getElement(3).densityValue * barCoords.w;

      // enter new value into the hash map
      hashMap.enter(updateVoxelPos, newDensityValue);
      writeToVolume(updateVoxelPos, newDensityValue);
    }
  }
}
```

Figure 3.25: Basic flow of the mapping algorithm.

3.6.3 Mapping Algorithm

Basic Flow

This pseudo code listing in figure 3.25 shows the basic flow of the mapping algorithm. Note that the memory management component described in detail in subsection 3.6.4 is not required during the mapping.

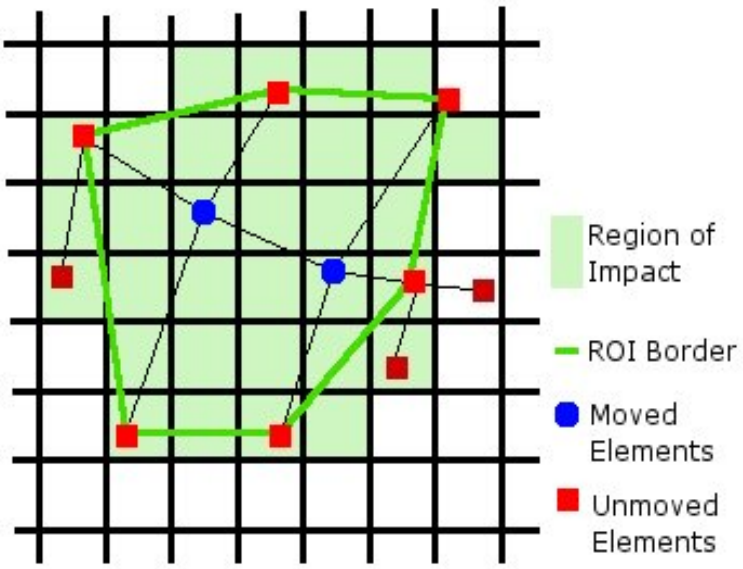


Figure 3.26: The blue dots represent the moved elements. The red squares are elements that were not moved in the last deformation step. The green grid elements have to be updated.

Mapping the Object to the Volume

Influenced Voxels

The algorithm starts with finding all volume voxel positions that need to be updated. Since the deformation is propagated locally in an outwards fashion through the object, the unmoved elements that have at least one moved neighbor element present a border for the region the deformation impacted. Here it is important to only consider those elements as moved that have been moved in the last deformation step to avoid unnecessary calculations. Hence, all volume voxels that belong to grid elements which are inside the region of impact or which contain the borderline of this region have to be updated.

For grid elements that are completely outside of the region of impact no influencing element was moved and they not need to be recalculated. An examples for the region of impact is given in figure 3.26.

The approach applied to find the region of impact exploits the shape of the chain regions to identify the voxels that have to be updated. There must be at least one element inside two neighboring grid elements. This is because the maximum allowed distance of two elements given by the chain region is smaller than the maximum length of two grid elements combined and the maximum allowed shear is smaller than the length of one grid element. Hence, it is sufficient to calculate the grid

```
// calculating the voxels in the neighborhood
// of the current and last position of the
// moved element
foreach(v = movedVoxel.next()) {
  for (i=0;i<=1;i++)
  for (j=0;j<=1;j++)
  for (k=0;k<=1;k++) {
    p.set( Math.floor(v.getCurrentPosition().x) + i,
           Math.floor(v.getCurrentPosition().y) + j,
           Math.floor(v.getCurrentPosition().z) + k);
    calculateChangedPoint(p, DeformObject);

    if (v.getLastPosition() != null) {
    p.set( Math.floor(v.getLastPosition().x) + i,
           Math.floor(v.getLastPosition().y) + j,
           Math.floor(v.getLastPosition().z) + k);
    calculateChangedPoint(p, DeformObject);
}
}
```

Figure 3.27: Listing of how the affected voxels are calculated.

element of the moved elements current and last position as well as their neighboring grid elements.

For each voxel that needs to be recalculated the eight enclosing cube elements are acquired. In the second step the elements are split to tetrahedrons. The corresponding numbers of the cube elements are shown in figure 3.28.

Splitting the Enclosing Cube

Asymmetrically splitting the cube to five tetrahedrons where the tetrahedrons consist of the respective cube elements (C_1, C_4, C_5, C_6), (C_0, C_1, C_3, C_4), (C_3, C_4, C_6, C_7), (C_1, C_2, C_3, C_6) and (C_1, C_3, C_4, C_6) creates asymmetrical deformations. For that reason splitting the cube is done in a symmetrical manner.

The algorithm used for splitting the cube requires the calculation of a virtual element that lies in the middle of the cube. The virtual element M for a cube consisting of the elements $C_0 ... C_7$ is calculated as follows:

$$M = \frac{1}{8} * \sum_{i=0}^{7} C_i$$

With this virtual element it is possible to split the enclosing cube to twelve tetrahedrons where the virtual element belongs to each tetrahedron and the other three elements are taken from the enclosing cube. This adaptation of the split creates symmetric visual results. An example of the split for one surface is given in figure 3.29.

We split the cube into twelve tetrahedrons in the following manner, where the virtual center element is denoted with M and the number of the cube elements is shown

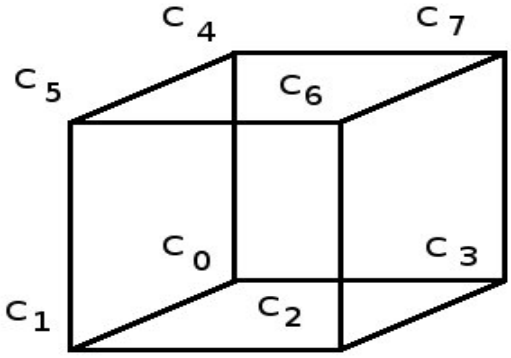


Figure 3.28: Corresponding element numbers for the splitting of the cube to tetrahedrons

in figure 3.28: (M, C_0, C_1, C_2), (M, C_0, C_2, C_3), (M, C_0, C_4, C_7), (M, C_0, C_3, C_7), (M, C_0, C_1, C_5), (M, C_0, C_4, C_5), (M, C_6, C_1, C_5), (M, C_6, C_1, C_2), (M, C_6, C_2, C_3), (M, C_6, C_3, C_7), (M, C_6, C_4, C_5), (M, C_6, C_4, C_7).

Calculating the Barycentric Coordinates

After splitting the cube, the enclosing tetrahedron of the volume voxel that is calculated is taken to calculate the barycentric coordinates. Given the barycentric coordinates b for a point P and the tetrahedron with the points (A, B, C, D) in the three dimensional case the absolute coordinates for point P are given as:

$$P = b_x * A + b_y * B + b_z * C + b_w * D$$

where the barycentric coordinates are related through:

$$b_x + b_y + b_z + b_w = 1$$

One tetrahedron yields the following 4 equations:

$$\begin{aligned} b_x * A_x + b_y * B_x + b_z * C_x + b_w * D_x &= P_x && (3.1)\\ b_x * A_y + b_y * B_y + b_z * C_y + b_w * D_y &= P_y && (3.2)\\ b_x * A_z + b_y * B_z + b_z * C_z + b_w * D_z &= P_z && (3.3)\\ b_x + b_y + b_z + b_w &= 1 && (3.4) \end{aligned}$$

This system is solved using the Gaussian elimination algorithm. If $0 \leq coordinate \leq 1$ for all four coordinates of the barycentric coordinates the element lies inside the tetrahedron and thus inside the cube.

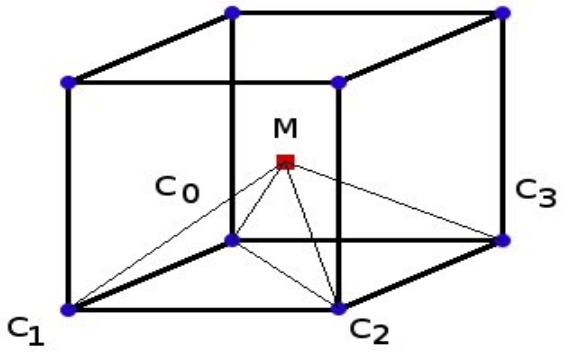


Figure 3.29: Using the bottom surface elements and the middle element yields two tetrahedrons (M, C_0, C_1, C_2) and (M, C_0, C_2, C_3). The cube could also be split using the other diagonal (C_1, C_3). The key of the splitting is that it is symmetrical.

Calculating the new Density Value

Hence, the new voxel value *val* is calculated by multiplying the values of the object elements the tetrahedron consists of with the barycentric coordinates of the point calculated.

$$val = b_x * A_{val} + b_y * B_{val} + b_z * C_{val} + b_w * D_{val}$$

To avoid multiple calculations for a single volume voxel and save processing time, since it can scheduled for recalculation more than once, we store already calculated voxel values in a look–up table. If a volume voxel needs to be updated it can first be checked if the voxel was already recalculated by looking it up in the table.

Mapping the Volume to the Object

Because each element models a single voxel it is sufficient to assign the corresponding density value to the element. The initial position of a new element is the same as the position of the corresponding voxel. Hence, a voxel is mapped to an element by simply copying the density value.

Deformation out of Volume Bounds

A scenario to consider occurs if the deformation is initiated close to a volume boundary. In this case the deformation might propagate over the volume boundaries. If this happens the object is extended with virtual elements so that the deformation calculation proceeds unaware of the problem.

In this case the mapping from the object cannot use the original voxel values to fill the *ChainMail object* with because the requested positions are out of bounds. Instead the object elements are initialized with zero.

It is also possible that a position out of the volume bounds is scheduled for recalculation. These positions are ignored because they cannot contribute to the visual image of the deformation.

Performance

The mapping algorithm is computational expensive because it requires to solve an equation system given by a 4x4 matrix $8 * 12$ times per position calculated in the worst case to find the enclosing cube or tetrahedron for a given position. Therefore, optimizations are applied. Only cubes and tetrahedrons that contain at least one moved element are calculated during the mapping process. Additionally, the result of the method also includes all other necessary information for calculating the new value. Implementing a faster algorithm for solving the equation system would have a great impact on the overall performance because it is also applied for checking if eight elements enclose a point.

Alternatively, a different algorithm for deciding if a point is inside a cube can be implemented. In this case after finding the cube the enclosing tetrahedron and the corresponding barycentric coordinates need to be calculated separately.

The algorithm currently used for finding the nearest element (please see section 3.6.2) to a given position consumes a lot of computation time too if the first try fails because the second try considers all currently loaded elements. Because the nearest element needs to be found for every calculation of a new voxel value an improvement of this algorithm would also decrease the computation time consumed, especially for larger objects.

However, the performance of the mapping is sufficient for the current application as laid out in section 6.1.

3.6.4 Memory Management

Macro Grid

For the memory management the entire volume is partitioned by a macro–grid consisting of cubes. In a *ChainMail object* the representation of such a cube is called *cubicle* . A *cubicle* consists of the mapped *ChainMail object* elements. The purpose of these cubes is to install a new grid separating the volume into equally sized memory chunks. This grid is then utilized by the memory management algorithm. A *cubicle* in 3D consists of $size^3$ ($size^2$ in 2d) grid elements (described in section 3.5.1) and the corner elements of a cube lie on a volume voxel. In the 2D case, the first cube lies at (0,0), (0, size), (size,0), (size,size). Its right neighbor has the coordinated

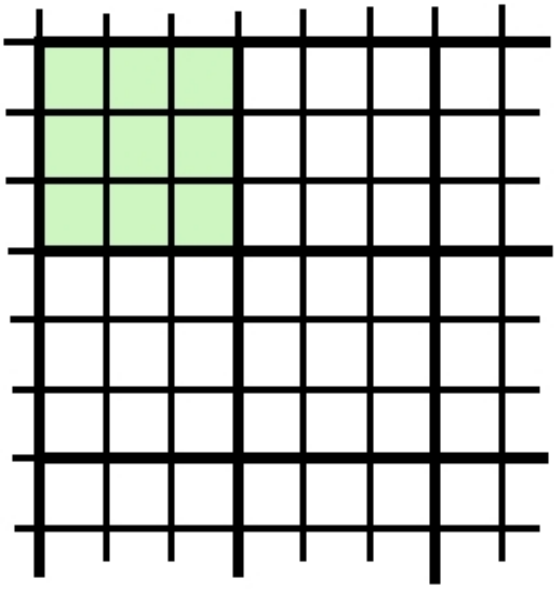


Figure 3.30: The thicker lines represent the lines of the macro–grid overlaying the original grid represented by the thinner lines. A single element of the macro–grid is marked by the green region.

(size, 0), (size,size), (2*size,0), (2*size, size) and so forth. Please see figure 3.30 for a 2D example.

Basic Flow

The algorithm for the memory management creates the necessary elements as depicted in figure 3.31. It assigns them the corresponding density values and creates the neighborhood links. The mapper is used to map the original volume density value to the newly created elements.

Prior to adding a new cubicle to the object it is checked if the cubicle was not already loaded and if a direct neighborhood required for the connection exists. Then all neighbor cubicles that are already loaded by the object are acquired and connected to the new cubicle. The basic flow looks as presented in figure 3.32.

Loading a Cubicle

To load a *cubicle* the mapper maps the requested volume cube to the *cubicle* (see section 3.6.3). The second step in this process is to create the neighborhood links between the elements. Please refer to figure 3.33

```
// map the requested position the corresponding position
// of the chunk in the macro grid
chunkPosition = getMacroGridPosition(requestedPosition);
// create a DeformChunk at the requested position
DeformChunk chunk = new DeformChunk(chunkPosition);
for (int i=0; i<size; i++)
for (int j=0; j<size; j++)
for (int k=0; k<size; k++) {
  element =
      new DeformVoxel(DeformMapper.getValueAt(position+(i,j,k)));
  chunk.addElement(element);
}
establishNeighborhoodLinks(chunk.getElements);
```

Figure 3.31: Pseudo code listing of the creation of a new *cubicle*.

```
if (chunkAlreadyLoaded(chunk.getPosition()) {
  throw new SamePositionException();
}
// chunk can be connected to the object
if (!directNeighborhoodExists(chunk.getPosition()) {
  throw new NoDirectNeighborhoodException();
}
// find all neighbor chunks of the object to the
// new chunk
neighborChunks =
      DeformObject.getNeighborChunks(chunk.position);
// connect the new chunk to all neighbor chunks
while(!neighborChunks.empty()) {
  chunk.connectWith(neighborChunks.next());
}
```

Figure 3.32: Pseudo code listing for adding a new *cubicle* .

Connecting a new Cubicle to the Object

To add a *cubicle* to an existing object the elements of the *cubicle* are linked with the elements of the *ChainMail object* according to their neighborhood. For example, if a new *cubicle* is inserted at the right side of the an already loaded *ChainMail object cubicle* , then the elements on the surface on the right side of the existing *cubicle* are be connected with the surface elements on the left side of the new *cubicles* . However, there may already be another neighbor to the newly inserted *cubicle* too (top or front for example). Hence, the surface elements of the new *cubicle* are connected with the corresponding surface elements of those object *cubicles* which are in the direct neighborhood. This means, that only *cubicles* that are part of the direct neighborhood of the existing *ChainMail object* may be loaded into memory. Otherwise the respective elements cannot be connected and the deformation could not propagate to these unconnected parts.

```
// create cubicle at the desired position
cubicle = new Cubicle(position);

// calculating the voxels in the neighborhood
// of the current and last position of the
// moved element
for (i=0;i<=1;i++)
for (j=0;j<=1;j++)
for (k=0;k<=1;k++) {

  // get the density value at the requersted position
  desityValue = mapper.getDensityValueAt(
    cubicle.position.x+i,
    cubicle.position.y+j,
    cubicle.position.z+k);

  // create the element
  h = new ChainMailVoxel(densityValue,
              cubiclePosition.x+i,
              cubiclePosition.y+j,
              cubiclePosition.z+k);

  // add the element to the elemtent list
  cubicle.voxelList.add(h);
}
// establishing neighborhood links
foreach(DeformVoxel v = cubicle.voxelList.next()) {
  v.setLeft(findLeftNeighbor(v, voxelList);
  v.setRight(findRightNeighbor(v, voxelList);
  v.setTop(findTopNeighbor(v, voxelList);
  v.setBottom(findBottomNeighbor(v, voxelList);
  v.setFront(findFrontNeighbor(v, voxelList);
  v.setBack(findBackNeighbor(v, voxelList);
}
```

Figure 3.33: Pseudo code of how a *cubicle* is loaded and the elements are connected.

3.6.5 Connecting two Cubicles

To connect a *cubicle* to a neighbor all voxels of the two surfaces facing each other are connected. In the first step the corner elements are used to find two opposing corner elements of the *cubicles* . Depending on the neighborhood the top or bottom corner element of the object's *cubicle* is used to find the element which opposes the bottom or top corner voxel of the neighbor *cubicle* to connect with. Please see figure 3.34 for an example.

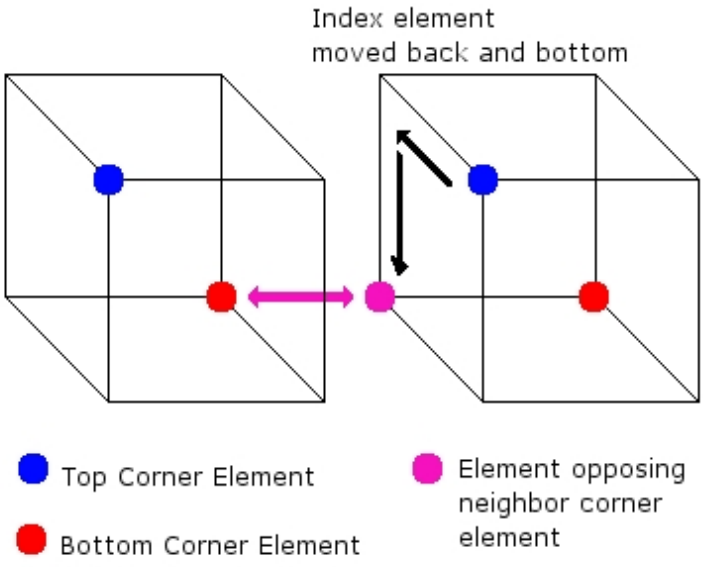


Figure 3.34: This figure shows two cubicles to be connected with each other. The top corner element is used to first step to the back corner element and then to the bottom corner element. Now, two opposing corner elements are found and can be used to connect all surface elements. (See figure 3.35)

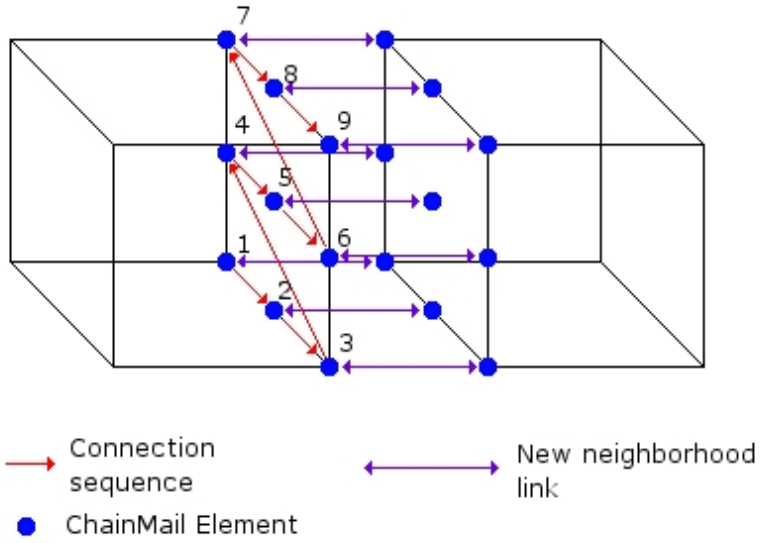


Figure 3.35: Picture of connecting two neighboring *cubicles* . The connection starts at the bottom corner element of the left *cubicle* and its opposing element. The first row of elements is connected (1-3). In the second step the top row elements get connected (4-6). Finally, the last row is connected (7-9).

```
/* example for connecting a cubicle to a left neighbor */
myConnector = getBottomCornerConnectorVoxel();
neighborConnector = neighbor.getTopCornerConnectorVoxel();

// move the myConnector element until it
// opposes the neighbor connector element
for (int i = 0; i<cubicle_size-1; i++) {
  myConnector = myConnector.getTop();
}
for (int i = 0; i<cubicle_size-1; i++) {
  myConnector = myConnector.getFront();
}
// my start element of the first row to connect
myLeadingEdge = myConnector;
// neighbor start element of the first row to connect
neighborLeadingEdge = neighborConnector;

// iterate through rows and elements in each row
for (int i = 0; i<cubicle_size; i++) {
  for (int j = 0; j<cubicle_size; j++) {
    // assign neighborhood
    neighborConnector.setLeft(myConnector);
    myConnector.setRight(neighborConnector);
    // get next element in row
    myConnector = myConnector.getBack();
    neighborConnector = neighborConnector.getBack();
  }
  // get my beginning element of next row
  myLeadingEdge = myLeadingEdge.getBottom();
  myConnector = myLeadingEdge;
  // get neighbor beginning element of next row
  neighborLeadingEdge = neighborLeadingEdge.getBottom();
  neighborConnector = neighborLeadingEdge;
}
```

Figure 3.36: Sample listing of connecting two *cubicles*.

For example, if the neighbor *cubicle* is connected with a *cubicle* of the object at the right side then the right surface of the neighbor is connected with the left surface of the object *cubicle* . Hence, the top corner voxel of the object's *cubicle* is used as the staring voxel to find the voxel that opposes the bottom corner voxel of the new neighbor by stepping to the back and bottom. These two voxels are then connected along with all their row front neighbors. Then the rows at the top side of the two voxels are connected and so forth. Figure 3.35 depicts a sample connection and figure 3.36 shows sample pseudo code for connecting two *cubicles* .

Memory Management near Volume Bounds

The memory management algorithm works the same if the deformation propagates out of the volume bounds. The grid defined by the cubes can be extended in any direction. The only difference occurs at in the mapping part. Because no volume values can be mapped the elements values are set to 0. This allows the calculation of the deformation near volume boundaries without further changes to the algorithm.

3.7 Conclusion

Fundamental ChainMail Algorithms

The first part of this chapter presented the ChainMail and the Enhanced ChainMail algorithm this work is based on as well as their shortcomings in respect to the requirements of STEPS. The ChainMail algorithm introduced by *S. F. F. Gibson* is a fast an flexible algorithm for the deformation of volumetric objects. It performs simple calculations on a large number of elements to achieve complex behavior and is capable of modeling tissue with different material properties and even anisotropic material. The approach of *M. A. Schill et al.*, called Enhanced ChainMail, extends the ChainMail algorithm to inhomogeneous data. They use the analogy of a sound wave that travels quicker through stiffer material than through softer material. This finding is used to change the order the elements of a *ChainMail object* are processed during the calculation of a deformation.

However, the algorithms could not be applied "as is" to the target environment. The task was to extend the STEPS application [Neubauer et al. '04] by introducing deformations. The STEPS is a virtual endoscopy system that visualizes volume data using a ray casting algorithm. Therefore, we introduce the *Divod ChainMail* algorithm which uses a *ChainMail object* as a means for calculating the deformation. The result of the calculation is then mapped to the volume data. Because of the large size of such a volume data set we implemented a memory management algorithm since the entire volume can not be modeled through a single *ChainMail object*.

Mulitmove

To integrate the ChainMail algorithm into the STEPS system we developed a *new way for interaction*. This allows to initially move eight ChainMail elements. Moving eight elements makes it likely that the endoscope's eye point lies outside of the deformed tissue and models the fact that the endoscope has a certain diameter. The drawback of this approach is that it may lead to an inconsistent object. However, this problem was disregarded for performance reasons and because no visual artefacts are produced.

Z–Scale

A visual distorting effect caused by the z–scale of the volume was soothed by dividing the amount of information propagated along the z–axis by the z–scale factor. This is done by dividing the z–component of the initial move vector by the z–scale and multiplying the chain region shear constraints for the x and y shear along the z–axis by the z–scale.

Global Constraints

To avoid unnaturally large deformations and the deformation of bone tissue we introduced two global constraints. One constraint governs the maximum allowed movement of an element while the other restricts the movement of elements which belong to bone tissue. If a violation occurs, the deformation is reverted and retried with a smaller step size.

Mapping & Memory Management

The last two presented parts of this chapter are the mapping and the memory management. The mapping of the object to the volume is of great importance for the quality of the visual image because the ray caster directly renders the volume data. The first approach implemented using regions of influence to calculate new element values has the risk of creating holes inside the volume. The barycentric coordinates based method uses a symmetrical approach for the tetrahedron generation because the asymmetrical approach produces asymmetrical results in the visualized deformation image.

The memory management uses the fact that the deformation propagates locally through the object. The object is extended at the boundaries if the deformation exceeds the currently loaded *ChainMail object*. If a new deformation is started outside of the volume the current object is cleared and a new object is created. This procedure allows to bypass the movement constraint but for performance reason this problem was not addressed.

The mapping and the memory management work together. Whenever a *cubicle* is loaded into memory, the volume values need to be mapped to the new *ChainMail object* part. If a deformation runs out of the volume bounds, the memory management proceeds normally and generates a virtual *cubicle* . Then the mapping initializes the new elements with zero value. This way the algorithm can be applied without further changes to deformations near the volume boundaries.

A description of the implementation of the described algorithm is presented in the next chapter.

Chapter 4

Implementation

4.1 Implementation Overview

This chapter presents a description of the current implementation. First, it describes the framework, the related STEPS software. This description is followed by an outline of the interface we developed to ease integration, and the data types used.

4.2 Related Software

The STEPS system shows the user a three dimensional view of the volume data and three cutting planes. A user sees the virtual reality in the three dimensional view through the viewpoint of the endoscope. Additional parameters such as the iso value threshold, barrel distortion or the viewing angle of the endoscope can be configured interactively. STEPS also supports force feedback if the endoscope collides with the tissue. For the sake of platform independence the STEPS application was entirely developed in Java.

4.3 Software Design

To allow flexibility of the software an object oriented approach was chosen for the design. The focus was to strictly separate the individual software components from each other. This allows flexibility and code reuse.

The first component implemented in the original plugin is a ray caster which visualizes the volume data. The volume data represents the second component. It is used as a clear separation between the deformation algorithm and the visualization because the ray caster does not need to have any knowledge of the newly implemented deformation algorithm. A general deformation algorithm for extending the STEPS system can be split into three distinct components.

- Deformation: calculation of the deformation on the deform object, depending on the input, typically the point of impact and a move vector.
- Mapping: mapping the volume to the object and mapping the deformed object to the volume data using a mapper object.
- Memory Management: loading parts of the object necessary for the calculation of deformation and release unnecessary parts.

To support this splitting a minimal API is desired. The idea of this API is to allow interchanging different component implementations. For example using different mapping algorithms for one deform object. However, this separation cannot be achieved in all cases because the mapping might depend on the corresponding object's data structure which holds information necessary for the mapping as well as the memory management depends on the shape and internal structure of the deformation object.

However, using the API presented in the next section a clean separation between the mapper and the object is achieved through the introduction of a DeformVoxel interface.

4.4 Interface

The purpose of this interface is first of all to outline how the components work together and describe the important steps of the work flow. Secondly, it separates the individual components and was created to enforce code reusability and flexibility. For example the implemented mapper is able to map any object that implements the given interface without changes. The interface diagram is presented in figure 4.1.

4.4.1 DeformVoxel

The DeformVoxel is the smallest unit of the interface. The DeformChunk consists of such DeformVoxels. It only offers a minimal interface, sufficient to be used by implementations of the DeformMapper interface. It implements a minimal set of functions, because each algorithm needs to implement its own voxel element class to add the algorithm dependent functionality.

Attributes

A DeformVoxel has the following attributes

- currentPosition: The current position of the element.
- lastPosition: The last position before the element was moved.
- originalPosition: The position the element was created at.
- value: The density vale of the element.

- left: The left neighbor element.
- right: The right neighbor element.
- top: The top neighbor element.
- bottom: The bottom neighbor element.
- front: The front neighbor element.
- back: The back neighbor element.

This set of attributes is sufficient for a wide range of mappers since the most important data for the mapper is the value, position and neighborhood of the elements that need to be mapped. Thus, a separation of the deformation calculation and the mapping is achieved.

Methods

- move(Vector3d): This method moves the element by the given vector. First the last position is set to the current position. Then vector is added to the current position.

4.4.2 DeformMovement

The purpose of the DeformMovement interface is to provide means for extending the information passed to the deformation object within the interface. For example, an application might also want to add information like normal vectors. However, classes that implement this simple interface can still be used "as is".

Attributes

The DeformMovement consists of two attributes.

- position: The starting position of the deformation.
- path: The deformation vector, i.e. the magnitude and direction of the deforming force.

4.4.3 DeformChunk

The DeformChunk interface provides the basic methods to for the DeformMemoryManager 4.4.4 and the DeformObject 4.4.5 to handle the memory chunks. Such a memory chunk consists of a list of elements and is created by the DeformMemoryManager. The DeformObject consists of these memory chunks and hence, is responsible for their tracking and connecting.

Attributes

- voxels: Supplies the contained DeformVoxels to the DeformObject.
- position: The position of the chunk. Each chunk is used to load a certain volume region. The position can be used to identify the region this chunk loaded.
- movedVoxels: A list of references to the moved element which belong to this chunk. This was added to the interface to offer mapping algorithms a way for directly accessing moved elements.

Methods

- connectWith(DeformChunk): This method connects the two neighboring DeformChunks. Because a DeformChunk has no knowledge of other loaded DeformChunks the DeformObject is responsible for connecting a new DeformChunk all neighboring DeformChunks of the entire object.

This method throws the following exceptions

- SamePositionException: This exception is thrown if the two chunks to connect have the same position to avoid unnecessary loading of chunks. Additionally, the two chunks can not be connected to each other.
- NoDirectNeighborhoodException: This exception is thrown if the two chunks cannot be connected because they are not direct neighbors.
- NeighborhoodAmbiguousException: This exception is thrown if the two chunks can not be connected because the direction of the neighborhood is not uniquely defined. To connect two chunks they must share a surface. In this case they share at most an edge.

These errors must be handled by the DeformObject when adding a new chunk.

4.4.4 DeformMemoryManager

The DeformMemoryManager is responsible for loading requested volume parts in the form of DeformChunks. Therefore, it maps the volume voxels to DeformVoxels and assigns them to the DeformChunk. To map the volume density values to the voxels a DeformMapper is used (4.4.6).

Attributes

- DeformMapper: Used to map the volume data to the newly created DeformVoxels.

Methods

- loadChunk(Point3d): This method is called by the DeformObject to create a new DeformChunk. It returns the DeformChunk at the given position. The chunk is created by mapping the requested volume potion to a list of DeformVoxels. This includes setting the current and original position and setting the neighborhood information of the DeformVoxels. The list of DeformVoxels is then assigning the new DeformChunk.

  Because the underlying mapper handles out of bounds errors they do not need to be considered in this method.

4.4.5 DeformObject

The DeformObject calculates the deformation and is passed to the mapper for updating the volume. It consists of DeformChunks and is responsible for keeping track of them and needs to connect new chunks to itself to keep the objects integrity.

However, the main purpose is the calculation of the deformation which is done in the performDeformation(DeformMovement) method. This method also updates the volume data using the DeformMapper interface.

Attributes

The DeformObject interface consists of the following attributes.

- DeformChunks: The list of chunks the object consists of.
- Mapper: The mapper that is used to map the object to the volume data.

Methods

The following methods are offered by the DeformObject interface.

- getVoxels(): Returns the list of DeformVoxels belonging to the object by calling the getVoxels() method for each contained chunk.
- getMovedVoxels(): Returns the list of moved DeformVoxels belonging to the object by calling the getMovedVoxels() method for each contained DeformChunk.
- getChunk(Point3d): Returns the DeformChunk at the given position if it is loaded, null otherwise.
- performDeformation(): Calculates the deformation and updates the volume data using the mapper attribute.

This interface does not export any errors. This minimizes the necessary effort for integrating it into an existing software. performDeformation() is the only method that should be used by the calling application. The mapper should be set at object creation. The other methods are interfaces offered to the other components.

4.4.6 DeformMapper

The DeformMapper interface provides the methods to map the volume to a DeformChunk and to map a deformed DeformObject to the volume. To separate mappers from the object's implementation the DeformVoxel interface is introduced. This interface offers information sufficient for a wide range of mapping algorithms.

Attributes

- volume: The volume to be used during the mapping process.

Methods

The three methods the DeformMapper offers to the DeformObject and the DeformMemoryManager are:

- updateVolumeData(DeformObject): This method maps the given object to the volume attribute. If the method uses the DeformVoxel interface to calculate the new values from the deformed object it works with every object which also implements this interface.
- getVolumeValueAt(): This method returns the density value of a given position.
- getZScale(): Returns the z–scale of the volume.

The mapper must handle all out of bounds errors internally so that the other components remain unaware of the error.

The next section describe the actually implemented classes for realizing the ChainMail algorithm.

4.5 Data Structure

4.5.1 ChainMailVoxel

The ChainMailVoxel implements the DeformVoxel interface. Further, it contains the following attributes and methods.

Attributes

- id: A unique identifier for the ChainMailVoxel.
- chainRegion: The corresponding chain region of the ChainMailVoxel.
- ignoreDeformationDirectionX,
  ignoreDeformationDirectionY,
  ignoreDeformationDirectionZ: The direction along the respective axis where no

neighbors may be added to the candidate list because they are added by another initially moved element. For example if the ignoreDeformationX flag is set to left then the left neighbors of the ChainMailVoxel are not added to the candidate list Please refer to section 3.5.3 for a description of the problem.

- isInitiallyMoved If set to true if the ChainMailVoxel is an initially moved element.
- wasMovedDuringDeformationStep: Set to true if the ChainMailVoxel was moved during the last deformation step.

Methods

- move(): This method first checks the ChainMailVoxel value against the bone constraint. If this check is passed the ChainMailVoxel's current position is stored in last position. Then the ChainMailVoxel is moved by the given vector, and marked as moved during the last deformation step. Finally, the new ChainMailVoxel position is checked against the movement constraint. If the movement constraints is violated the method throws a MovementViolatesMoveConstraintsException. If the bone constraint is violated the method throws a MovementViolatesBoneConstraintsException. These exceptions are caught by the ChainMailDeformObject which reverts the current deformation and retries it with a smaller step size.
- undoMove(): The method is called during a reversion of the deformation. This method clears the moved flag and sets the current position to last position.
- getViolation(): The method calculates the amount of the chain region constraint violation for two ChainMailVoxels. It is the ordering criteria for the candidates in the candidate list.

4.5.2 Chain Region

Attributes

- maxDx: The maximum allowed distance between two ChainMailVoxels along the x–axis.
- maxDy: The maximum allowed distance between two ChainMailVoxels along the y–axis.
- maxDz: The maximum allowed distance between two ChainMailVoxels along the z–axis.
- minDx: The minimum allowed distance between two ChainMailVoxels along the x–axis.
- minDy: The minimum allowed distance between two ChainMailVoxels along the y–axis.
- minDz: The minimum allowed distance between two ChainMailVoxels along the z–axis.

- maxHorzDx: The maximum allowed shear between two ChainMailVoxels along the x–axis.
- maxVertDy: The maximum allowed shear between two ChainMailVoxels along the y–axis.
- maxDepthDz: The maximum allowed shear between two ChainMailVoxels along the z–axis.
- zScale: The z–scale of the volume necessary to adjust chain regions as described in section 3.5.4.

Methods

- calculatePathToMeetConstraints(): This method calculates the necessary adjustment for a ChainMailVoxel to satisfy the chain region constraints. The method takes a candidate ChainMailVoxel and its sponsor ChainMailVoxel and returns the path the candidate has to be move to satisfy the constraints.

4.5.3 ChainMailCubicle

The ChainMailCubicle implements the DeformChunk interface. Additionally, it extends the interface by the following attributes.

Attributes

- left: The left neighbor ChainMailCubicle.
- right: The right neighbor ChainMailCubicle.
- top: The top neighbor ChainMailCubicle.
- bottom: The bottom neighbor ChainMailCubicle.
- front: The front neighbor ChainMailCubicle.
- back: The back neighbor ChainMailCubicle.
- leftTopFrontCornerVoxel: The voxel in the left, top and front corner of the DeformCubicle.
- rightBottomBackCornerVoxel: The voxel in the right, bottom and back corner of the DeformCubicle.

  These two voxels are required for connecting two neighboring ChainMailCubicles.

4.5.4 ChainMailDeformObject

The *ChainMail object* is responsible for calculating the deformation and associates all other components. It strictly implements the DeformObject interface.

4.6 Configuration

Upon creation the ChainMailDeformObject reads the configuration and sets the parameters for itself and the associated components. The configuration is a stored in a simple text file. The entries are key value pairs separated by a '='.

The configurable parameters are:

- cubicle size: Sets the size of the *cubicles* . Bigger *cubicles* means less loading operations but a single load takes more time.
- scale factor: The factor the initial move vector is scaled by. This allows adjusting the visual effect.
- offset: This value defines the shape and size of the chain regions. A smaller offset means smaller chain regions and stiffer behavior of the object.
- maximum allowed movement for element: Defines the movement constraint.
- bone threshold: Defines the bone constraint.
- adapt regions to z–scale: A flag that toggles the z–scale adjustment.
- max tries: Defines the number of retries in case a constraint violation occurs during the calculation.
- step scale: Defines the factor by which the initial move vector is scaled at the start of each retry. For example, if the movement is retried the first time the vector is $move_{retry1} = move_{original} * step_{scale}$ and for the second retry it is $move_{rety2} = move_{original} * step^2_{scale}$. Therefore, the step scale should be between 0 and 1 exclusive.

This section described the main data structures used in this implementation.

4.7 Integration

Requirements

The requirement for the integration of the deformation algorithm was that it must not affect the ray casting algorithm in any way. Secondly, only a minimum of code should be subject to integrate the ChainMail algorithm into the STEPS application.

The second requirement was met by providing a minimum interface for the deformation to the STEPS system. The existing isCollision method that detects a collision was extended to calculate the intersection point and the deformation vector using an iterative approach. The intersection point and the deformation vector are then passed to the ChainMailDeformObject via the performDeformation() method with calculates the deformation and updates the volume. Finally, no further error handling is necessary in the STEPS system because no errors a propagated through he deformation interface minimizing the change to the code necessary in the STEPS application.

Changes to the STEPS System

The first requirement was met by directly updating the volume data. The ray caster only works on the volume and does not need any additional information for the visualization. Therefore, using the volume as the interface between the deformation and the visualization the ChainMail algorithm can be added without having to change the ray casting algorithm. The ray caster is completely unaware of the new component.

The only component that has to be changed in the current STEPS system is the user interaction. In the original application the user cannot interact with the tissue. However, the user can set a "detect collisions" flag. This flag tells the application to check for collisions between the iso surface and the endoscope. If a collision occurs the eye point of the endoscope is set back to the last valid position.

This is done to ensure that the eye point of the endoscope is kept outside the tissue. Obviously, this function presents a good starting point for the integration of the ChainMail algorithm for two reasons.

- It is able to detect collisions between the iso surface and the endoscope. Such collisions trigger the start of a deformation.
- It is responsible for replacing the endoscope's eye point to keep it outside the tissue. This is also a requirement for the deformation.

Hence, the so called isCollision() method was changed to introduce the ChainMail extension to the existing STEPS system.

Acquisition of Deformation Data

The calculation of the deformation depends on three things as input. The volume data, the starting point of the deformation as well as the deformation vector which gives the length and direction of the deformation.

Obviously, the intersection point of the endoscope and the iso surface is the starting point of the deformation. Unfortunately, the isCollision() method only checks if the eye point of the endoscope lies inside the tissue. That means that the iso value calculated at the position of the eye point is bigger than the threshold. Hence, the method has to be extended to calculate the intersection point.

The calculation of the intersection point is done in an iterative fashion. A virtual point is repeatedly moved from the eye point towards the last position of the eye point by a small step size until it lies outside of the tissue. That means, the calculated iso value at the position is smaller than the threshold. Then the middle of this position and the last position inside the tissue gives the point of impact.

The vector of the deformation is given by the point of impact I and the eye point E of the endoscope. Hence, the deformation vector v is given by $v = E - I$.

Finally, the volume data needs to be passed to the deformation engine. This is done at object creation because the ChainMailMapper demands a volume at creation.

The objects are integrated in the same class that holds the isCollision method, namely the STEPSVolume class. The STEPSVolume class is extended by two private member variables to hold the ChainMailDeformMapper and the ChainMailDeformObject. The two objects are initialized if a first collision occurs.

To initiate a deformation, the "detect collision" flag must be set by the user. Then, if the endoscope penetrates the tissue, the intersection point and the deformation vector are calculated. Together they are used to create a DeformMovement object which is the input for the deformation calculation.

The *ChainMail object* provides the performDeformation() method as the interface to calculate the deformation. This method takes the DeformMovement as input and also updates the volume data using the ChainMailMapper member after a successful deformation. Hence, from the STEPS view only a single call is necessary to calculate the deformation and update the volume data.

Error Handling

No additional error handling needs to be implemented in the STEPS system because no errors are propagated trough the performDeformation() interface method. All errors that occur during the deformation calculation are handled entirely inside the ChainMail extension. In the worst case the volume will not get updated. However, this is still a valid case for the STEPS system. Hence, it is completely unaware of these errors which eases the integration of the ChainMail extension.

The next section concludes the chapter with a review of the presented implementation details.

4.8 Conclusion

This chapter described the implementation and integration of the *Divod ChainMail* algorithm into the existing STEPS application.

STEPS consists of 4 views: a virtual reality view and three cutting plane views. The goal of this work was to extend the STEPS plugin by adding a deformation engine. The deformation should then be visible in all four views. This is achieved using three components, a mapper, a memory manager and a component for the deformation calculation.

To allow easy integration, flexibility and code reusability an interface was introduced to allow the splitting of these three components. Additionally, the interface required by the STEPS application was minimized to a single call. This call does not propagate any errors hence, no additional error handling has to be done upon integration.

The separation is based on the fact that all four views directly render the volume data. Hence, the volume data is used to separate the extension from the visualization process. The only additional thing that needed to be done within the existing plugin was to calculate the point of impact and the force vector of the collision. These are the parameters necessary to calculate the deformation and requires to change the method STEPSVolume.isCollision().

The actual implementation of the *Divod ChainMail* algorithm follows the interface. For example, the implemented mapper is capable of mapping all objects that implement the ChainMailVoxel interface.

The object that calculates the deformation implements the DeformObject interface and consists of *cubicles* which hold the actual elements. The *cubicles* and elements implement the DeformChunk and DeformVoxel interface respectively. The object is mapped to the volume by a mapper that implements the DeformMapper interface. This mapper is also used to map the volume to the object. This is done if the object requests a missing chunk from the memory manager which implements the DeformMemoryManager interface.

The results of this work as well as a discussion and outlook to further work is presented in the next chapters.

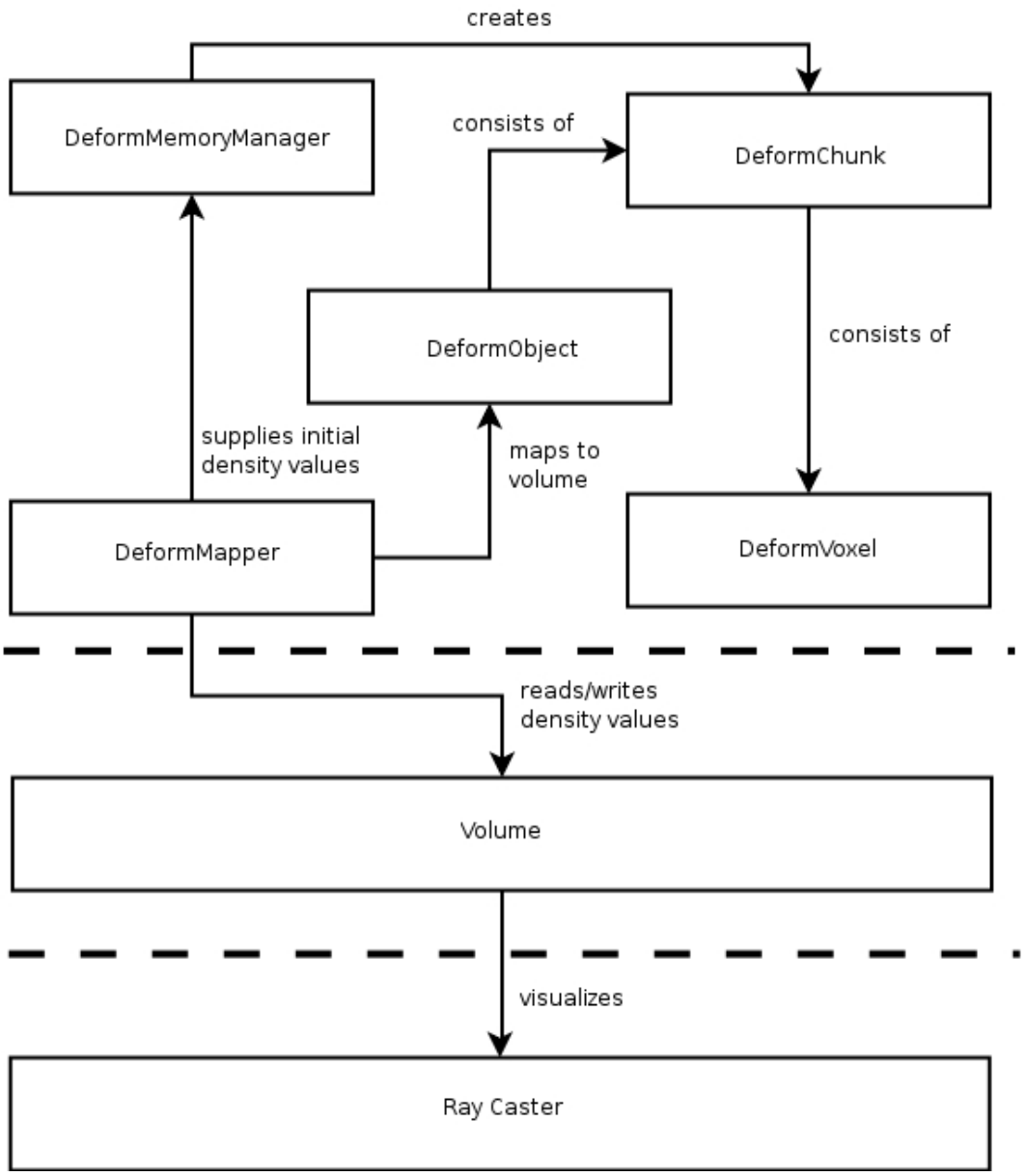


Figure 4.1: Interface diagram.

Chapter 5

Results

5.1 Overview

This chapter presents the results of this work. To show the functioning of the *Divod ChainMail* algorithm and the impact the parameters have on the resulting image the next section first shows results from artificial test data sets before continuing with screenshots from the actual application. There are two artificial data sets used. One is a homogeneous cube with the density values of 2000, the other is an inhomogeneous cuboid with density values of 1000 and 2000 to 2500.

The visual results are followed by an analysis of the algorithms performance based on timing tests. This analysis also refers to different parameter tunings and relates the visual results to the achieved performance.

The analysis is concluded by a discussion of the results and an outlook on further work in the next chapter.

All tests were conducted on an Acer Travelmate 6000 with an Intel Centrino 1600 MHz and 768 MB RAM. The operating system used for the tests is Windows XP Home Edition SP 2. The images were taken using the Tiani JVision 3.13 and the STEPS .15 plugin. The timing tests were done with Eclipse 3.0 and the Java SDK 1.4.2, Java RE 1.4.2.

5.2 Visual Results

The first set of results presented in this section were created by deforming a cube 5.1.

For each deformation the initial deform vector is $(0, 1, 0)$. Hence, the depth of the deformation only depends on the scale factor defined in the configuration. The second parameter that influences the shape of the deformation is the offset value which defines the size of the chain regions. Unless stated otherwise, the offset used for the images is 0.2 and the scale factor 1.2.

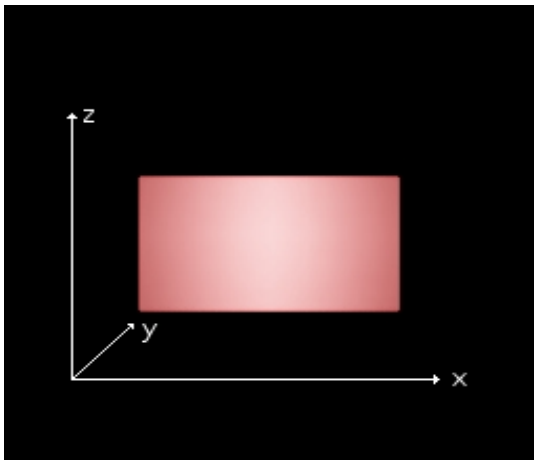


Figure 5.1: Test cube with the corresponding axis alignment.

The next section presents the results of the timing tests used to measure the performance of the implementation.

Z–Scale

Figure 5.2 shows the effect of the z–scale adjustment. The deformation on the left side has no z–scale adjustment. Hence, it appears stretched along the z–axis. Please note, that the deformation is too large along the z–axis and not too small along the other two axes. Therefore, the propagation along the stretch has to be diminished. The picture on the right shows the effect of the adjustment. The deformation still appears stretched by a small factor. This is because the deformation is propagated in a discrete manner by the Chain Mail algorithm. The effect gets for smaller for smaller offsets because it is a better approximation to continuous data. The offset value used for this image is 0.25.

The effect is also visible in the cutting plane views shown in figure 5.3.

Angular Deformation

Figure 5.4 shows the difference between an angular and non angular deformation. The middle of the deformation is shifted in the direction of the force shown by the yellow vector. The increase of the surface is steeper at the pushed part left of the deformation's middle than on the pulled part on the right which is physically plausible.

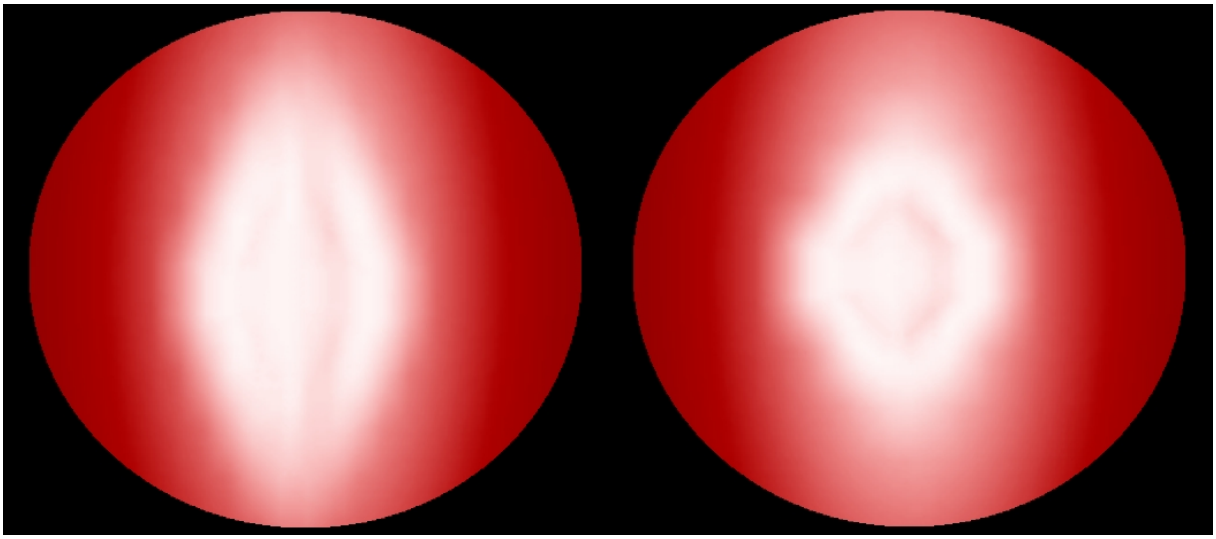


Figure 5.2: This figure demonstrates the effect of the z–scale adjustment.

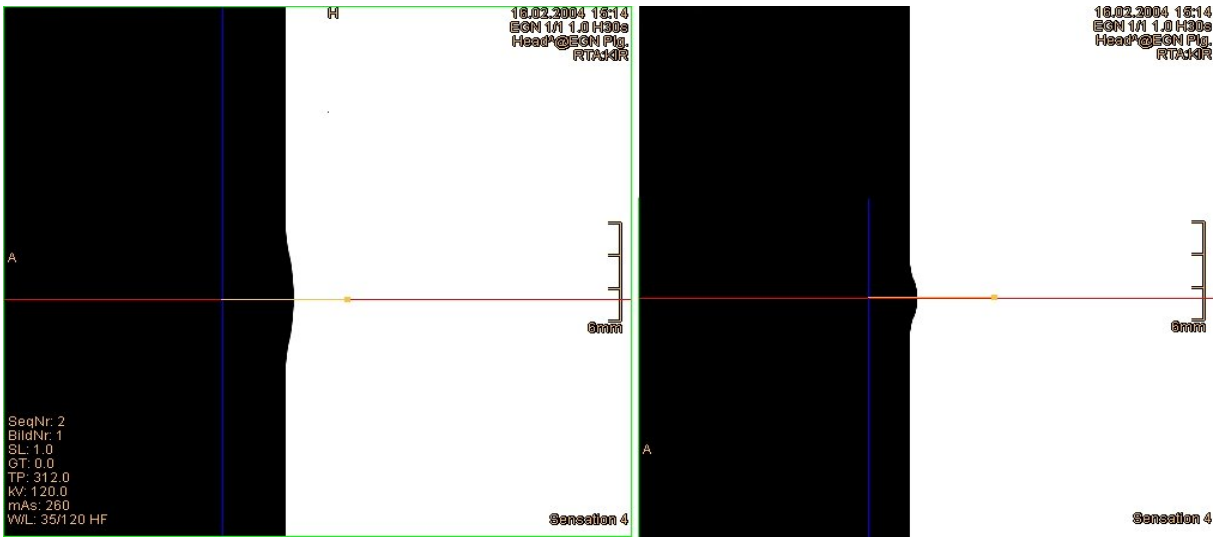


Figure 5.3: The stretch along the z–axis shown in the corresponding cutting plane views

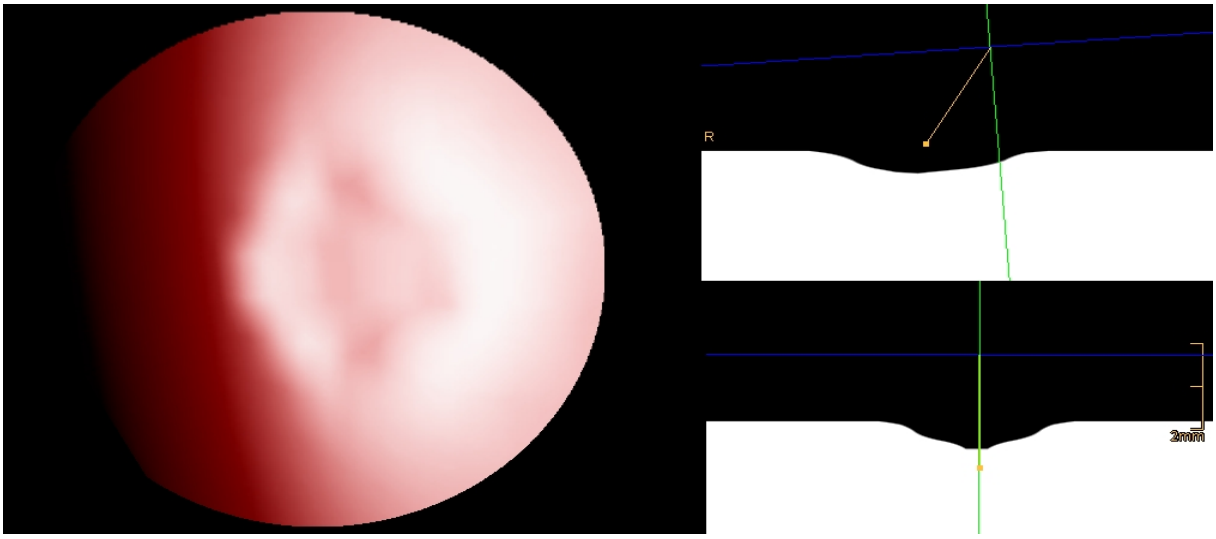


Figure 5.4: Picture of an angular deformation on the left. The top right side shows the corresponding cutting plane view. The picture on the bottom right shows the same deformation with a non angular force vector for reference.

Subsequent Deformation

The result of subsequent deformations is depicted in figure 5.5. Comparing the resulting image to figure 5.2 the huge difference caused by the offset change is visible. On one hand the deformation does not appear stretched any more. However, the shape of the deformation is rectangular rather than circular because of the cubic chain regions. Another effect caused by the cubic shape of the chain regions is presented in figure 5.10. The result of several continuous deformations is shown on the right side. First there middle area of the deformation was deformed further but because of the movement constraints, its size is smaller. Two other deformations show on the top left of the original deformation. Again, they are smaller because the movement constraint restricted the movement of the already deformed parts. However, they are bigger than the deformation in the middle since the outside elements were not moved as far as the inside elements in the initial deformation.

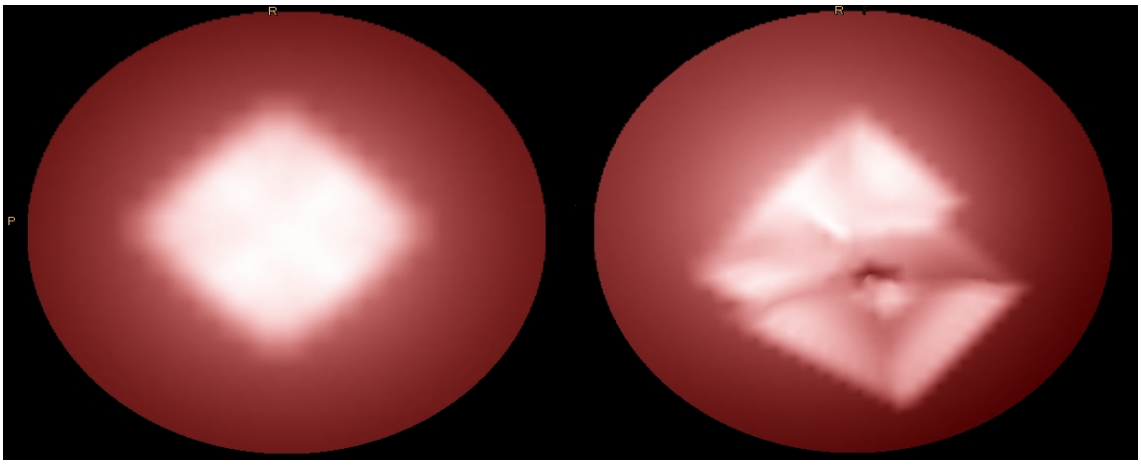


Figure 5.5: The left picture shows the initial deformation. The right picture shows subsequent deformations. The offset value used for the two images is 0.1.

Deformation of Edge

A special case of a deformation at the edge of the cube is presented in figure 5.6. It shows the depth of the deformation and the tissue appears to be drawn towards the point of collisions (left picture).

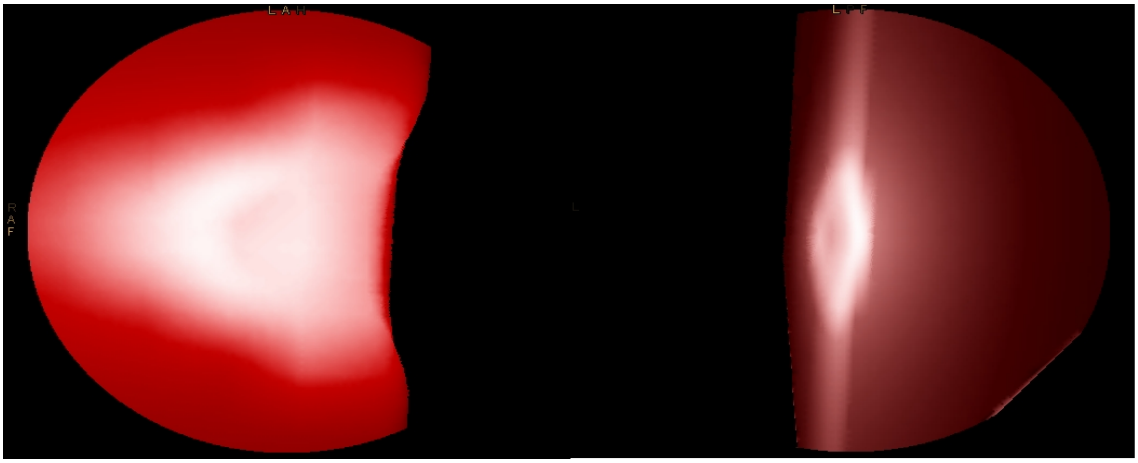


Figure 5.6: This figure shows the result of a deformation near the edge of the cube. The picture on the left represents the view in the direction of the deformation vector. The picture on the right shows a direct view of the deformed edge.

Inhomogeneous Data

The results of modeling inhomogeneous data are presented in figure 5.7 and figure 5.8. Figure5.7 depicts the deformation near the edge of two regions with different density values. The deformation in the upper side of the edge, where the data has higher density values, is smaller than the deformation in the lower part of the data with smaller density values.

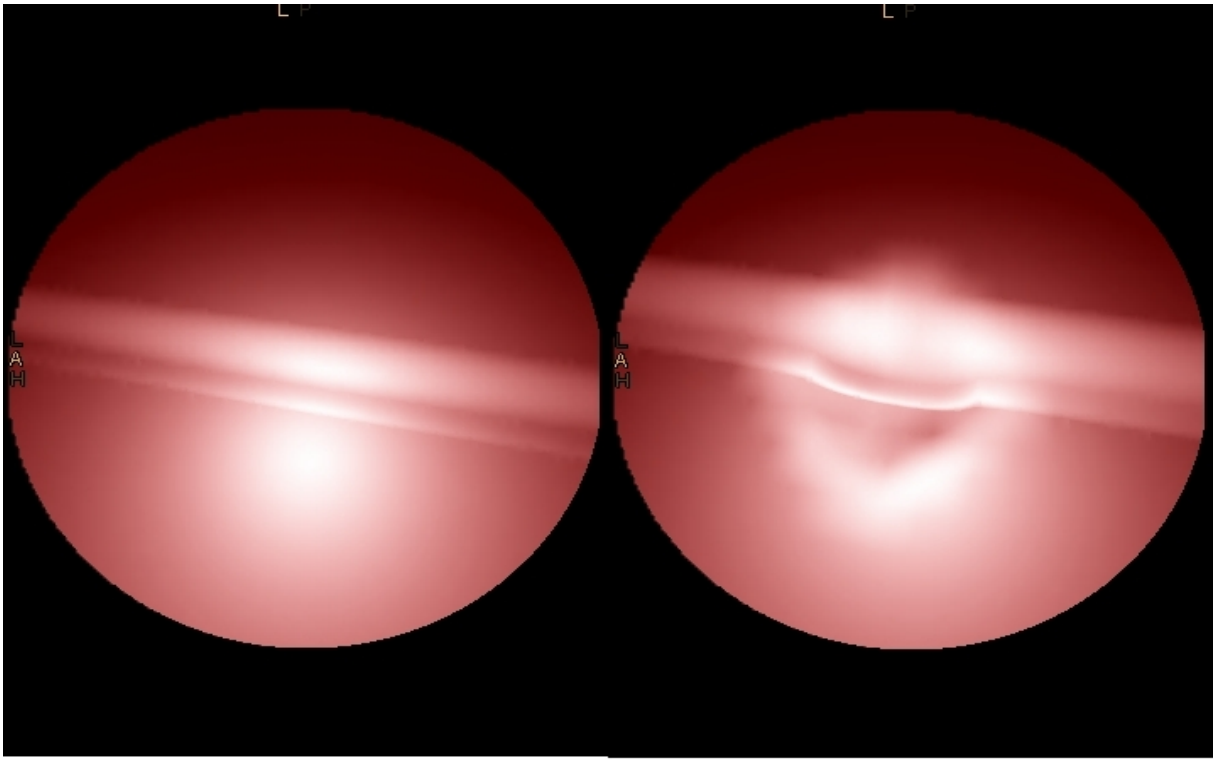


Figure 5.7: This figure shows the effect of the inhomogeneous chain region near the edge of the two different data regions.

In figure 5.8 The upper left side of the figure shows an inhomogeneous data set. The softer material on the upper side of the data set has a density value of 1000 and is marked with red frames. The stiffer material on the lower side of the data set has a density value of 2000 and is marked with blue frames. The pictures on the right are enlarged views of the deformations on the left. The top view shows the softer material, the bottom view the stiffer material. The figure on the left bottom is the cutting plane view of the deformed volume data. The pictures show, that the deformation of the softer material is steeper and slightly bigger. This is because the chain regions used for modeling the softer material are bigger. The cutting plane view also reflects the difference between the deformation. (In this figure the effect is exaggerated in the cutting plane view because of the different density regions.)

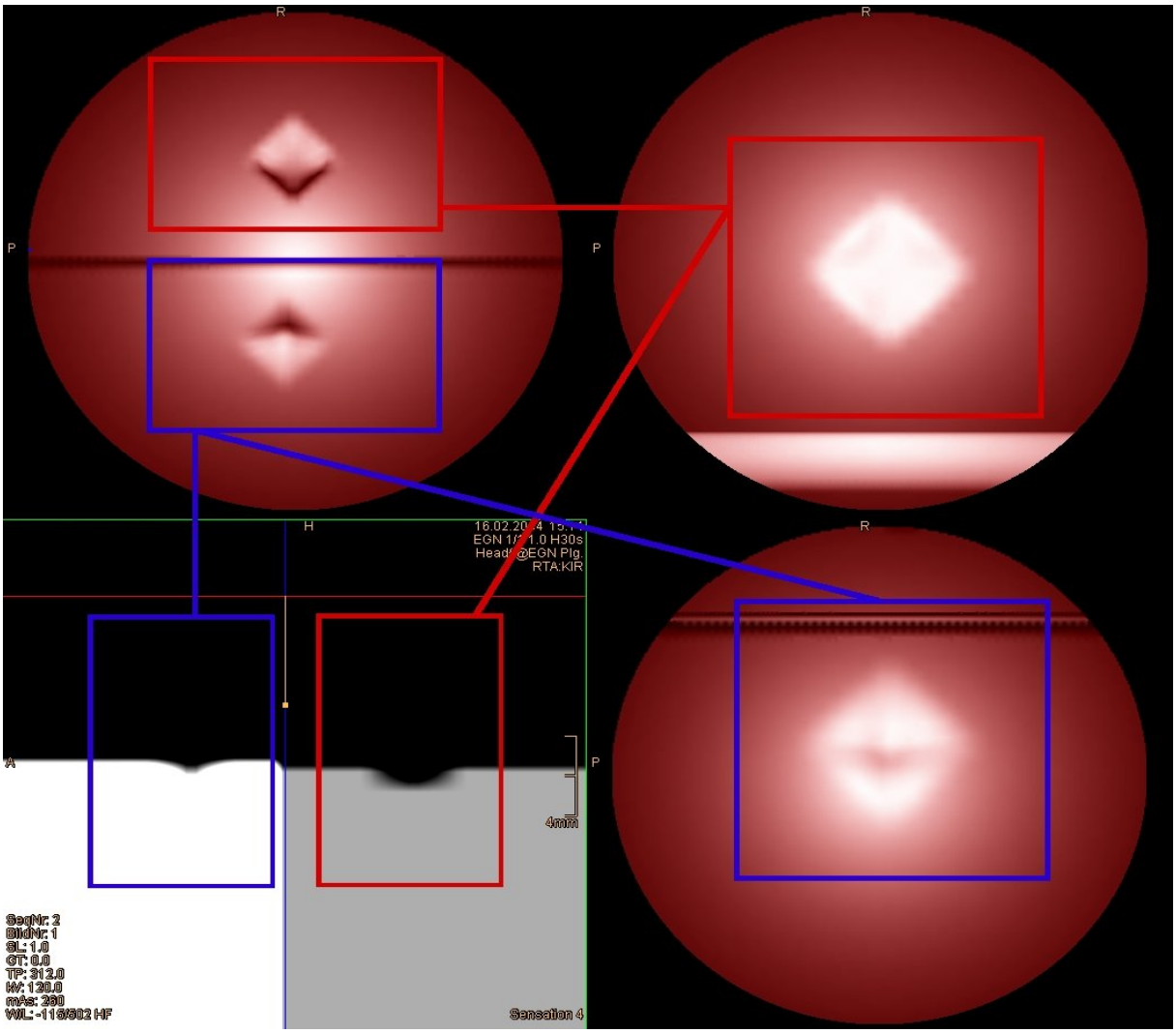


Figure 5.8: This picture shows the effect of the inhomogeneous chain regions.

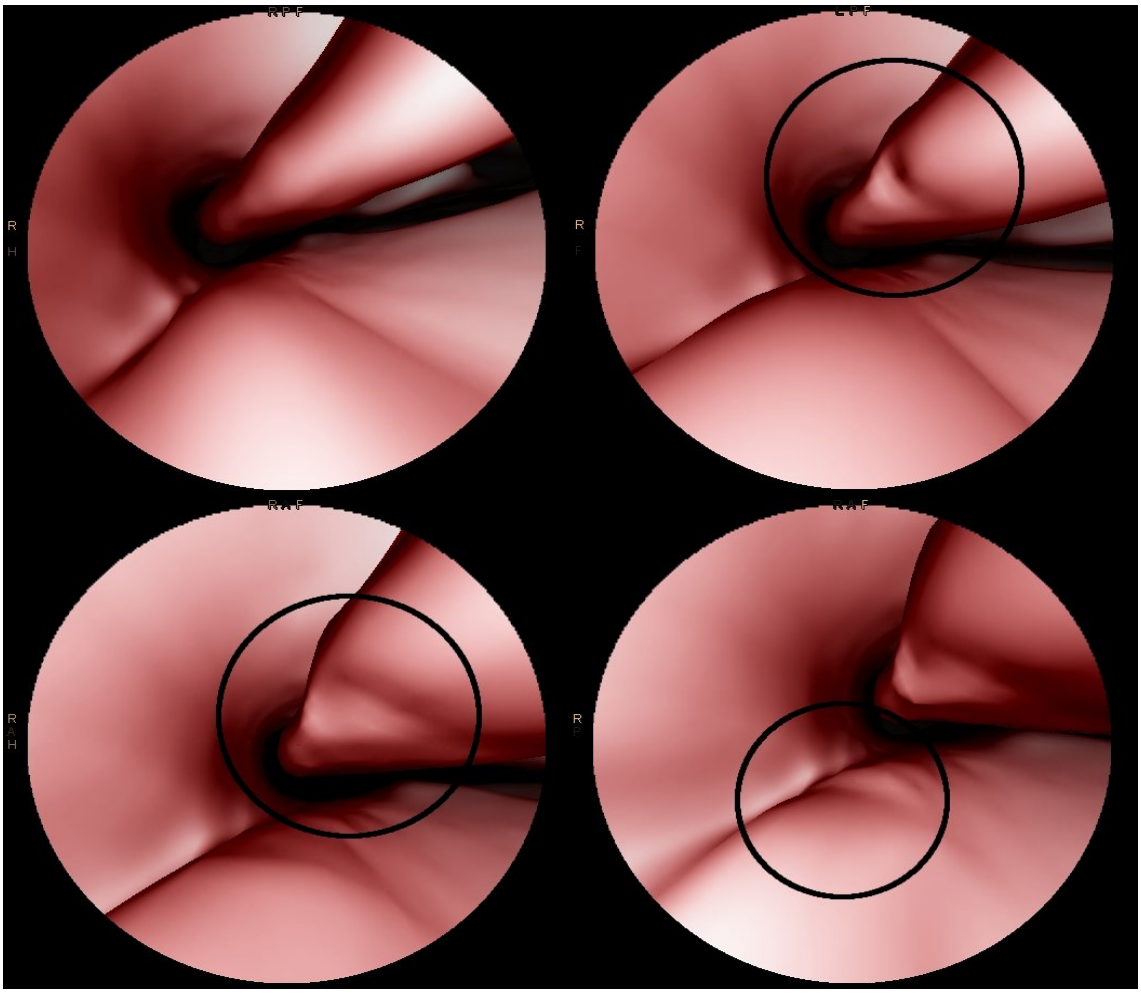


Figure 5.9: This sequence shows a typical use case. The offset is 0.2 and the scale factor is 1.2 in this image sequence.

Application

A typical use case is shown in figure 5.9. The endoscope is currently in the nose and pushed forward towards the sphenoid. In this case the space is narrow and collisions are likely to occur. The top left picture shows the initial data set without deformations. As the endoscope is pushed further it collides with the tissue. The resulting deformation encircled in the right top picture. The resulting image of a consecutive deformation is encircled in the right bottom picture. A new collision at the opposite side of the latter deformations deforms the tissue further.

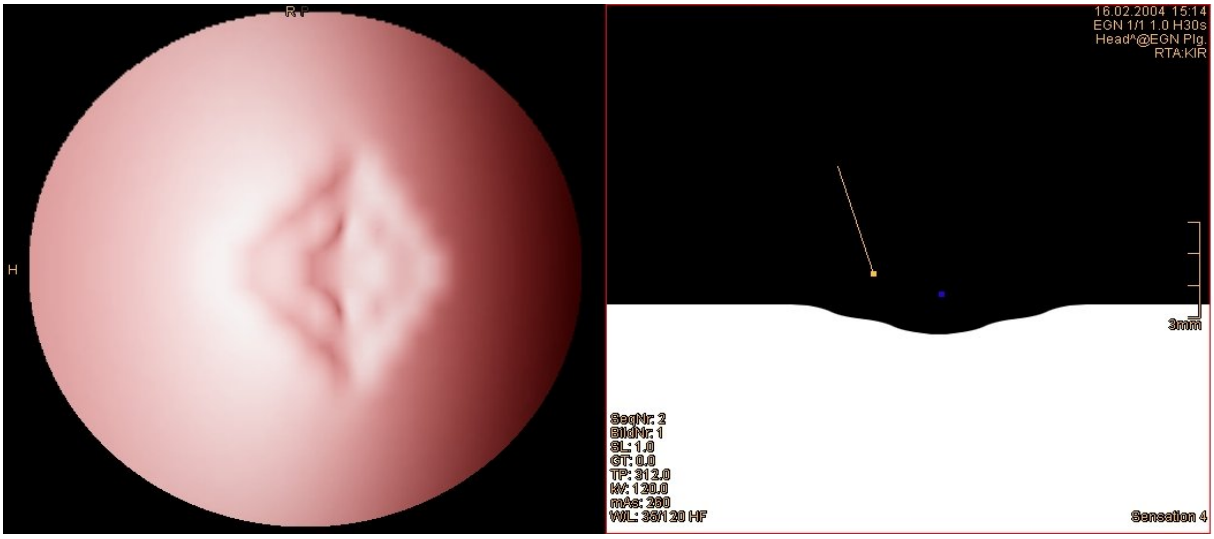


Figure 5.10: Stage Artefact.

Artefacts

Figure 5.10 shows an artefact that occurs for large deformations with large offset values. (The offset for this image is 0.25 and the scale factor is 2.5). For such a setup a stage effect occurs in the deformation. The reason for that is the cubic shape of the chain regions. The left image shows the virtual view. It clearly shows the stages between the moved elements. The right view presents the cutting plane view which also reflects the stage artefact for this setup. This effect can be soothed by assigning a smaller offset, however, the deformation would then become very large and the computational cost increases dramatically (please refer to the timing test section 6.1).

A similar situation as in figure 5.9 is shown in figure 5.11. Again the endoscope reaches a narrow passage through the nasal ways. The top left picture shows the undeformed data. In the top right picture, two deformations were conducted. One by shifting the endoscope to the left, the other by shifting the endoscope to the right. Then the endoscope was pushed forward and the tissue was deformed in the same manner as described above. The result is presented in the bottom left picture. The final picture on the bottom right was taken after deforming the tissue again but this time only the tissue on the right side of the endoscope was deformed. This creates an unwanted bump on the left side. The reason for this is, that the *ChainMail object* does not regard the shape of the modeled object. In this particular case the object is too large. It models both sides conjoined although they should be treated independently because they are separated. Hence, if one side is deformed it also affects the other side. The next image shows another artefact which is caused by the inability of the object to model the shape of the tissue.

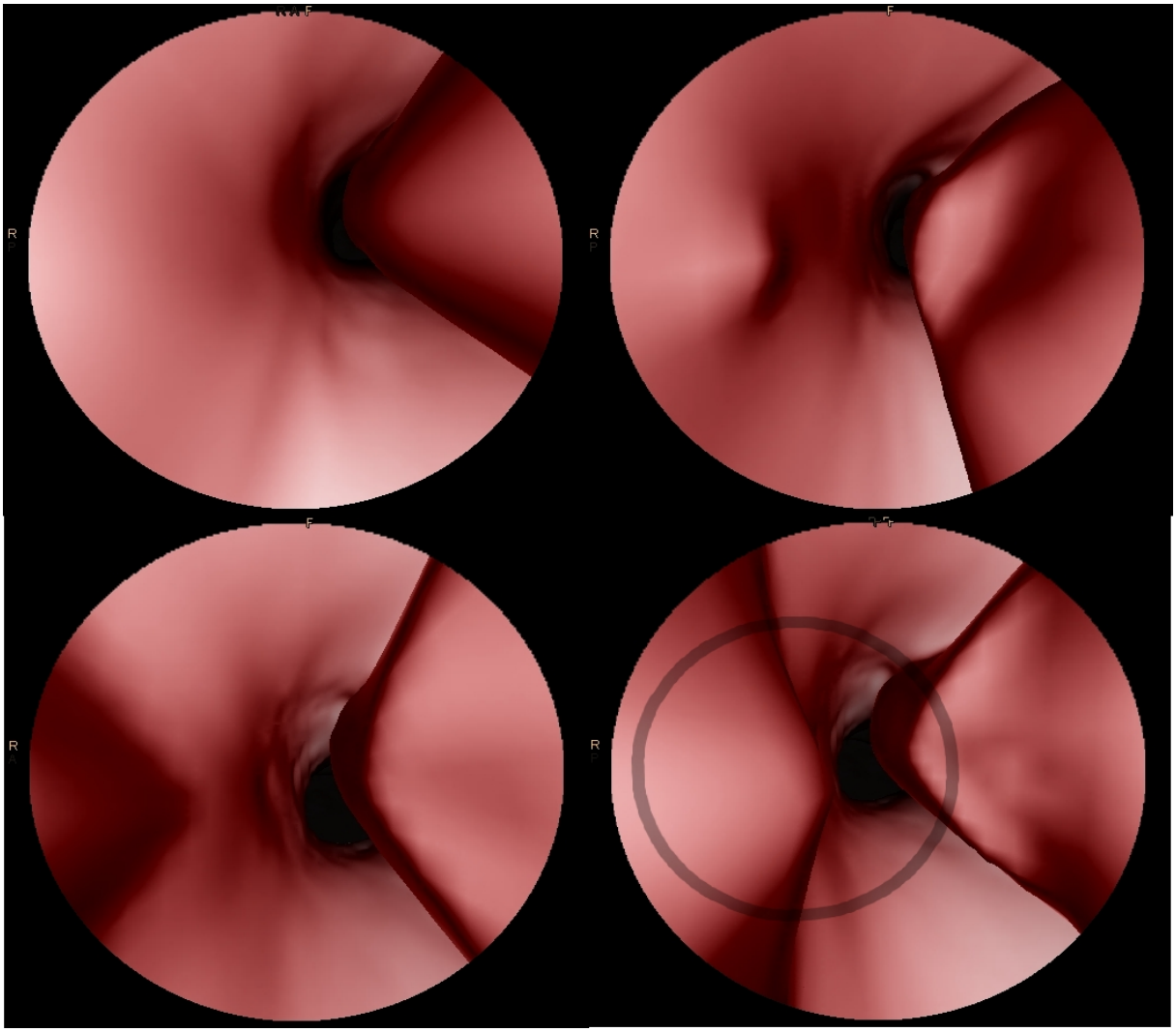


Figure 5.11: Artefact due to not modeling the object shape.

Another artefact caused by not modeling the shape of the object is shown in figure 5.12. The left picture shows the non deformed view. After the deformation the entire bump structure disappears as shown in the right picture. This behavior is not physically plausible. The structure would rather be pushed or bent to a side. The reason the bump disappears is that there is no tissue around the bump. Hence, the deformation shifts the bump deeper into the existing tissue but no surrounding tissue that would contribute to the position where the bump was exists and the bump disappears.

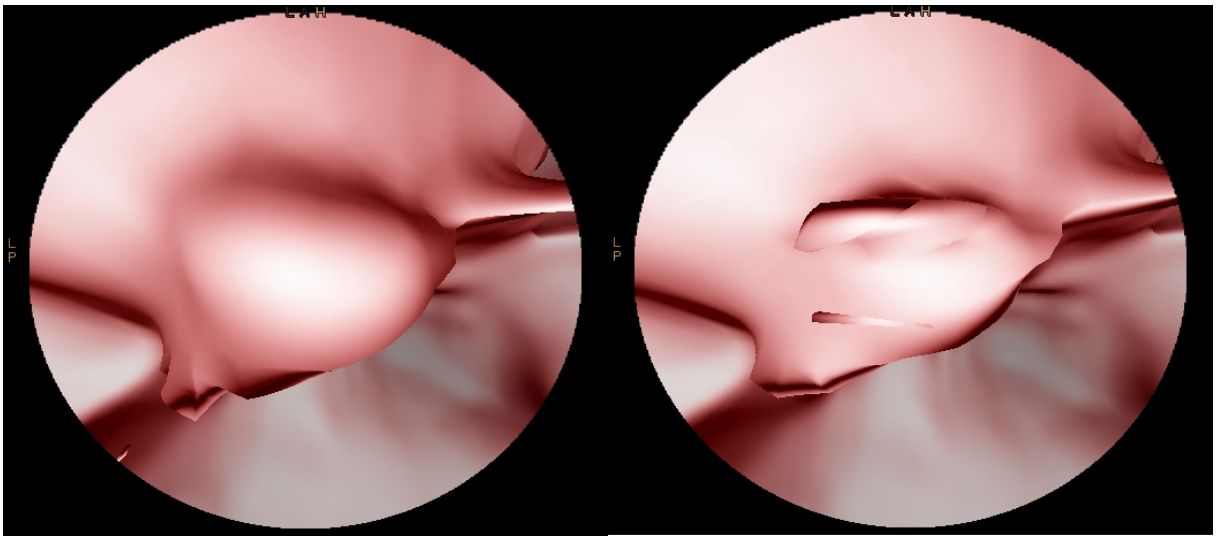


Figure 5.12: Disappearing of bump structure due to not modeling the object shape.

Chapter 6

Analysis

6.1 Timing Tests

The first observation presented in this section is the measurement of the memory management component. Hence, the time required to load a *cubicle* depending on its size. The effort for loading a cube is $O(n^3)$.

Cubicle size	time [ms]
3	1.062
4	4.657
5	12.698
6	31.755
7	66.176
8	132.35
9	248.29
10	446.81
11	740.36
12	1156

Figure 6.1: Time required to load a *cubicle* depending on the *cubicle* size.

Figure 6.1 shows that *cubicle* sizes bigger than 7 should be avoided because in this case loading of two *cubicles* already requires more than 0.25 seconds. It is very likely that at least two *cubicles* need to be loaded during a deformation step. Together with the time required for calculating the deformation the performance is too slow for interactivity. A smaller *cubicle* size requires less time for loading a *cubicle* and minimizes the overall memory usage. However, the overload for inserting increases and more load requests are necessary. We suggest *cubicle* sizes ranging from 3 to 6.

Figure 6.2 presents timing tests for the the deformation calculation which also includes the loading of missing *cubicles* and the mapping. It shows that the deforma-

chain region offset	Number of moved elements	Time for Deformation [ms]	Time for Mapping [ms]
0.40	56	110	100
0.20	224	160	270
0.15	392	170	441
0.10	1000	230	1171
0.08	1888	331	4757

Figure 6.2: The timing for the Deformation and Mapping components respectively. The scale factor for the tests is 1.0. The Deformation performs well even for a large amount of elements but the time needed to calculate the updated volume data increases drastically with the number of moved elements.

tion calculation and memory management (using a *cubicle* size of 3) perform well even for a large amount of points. However, it clearly identifies the mapping as the performance bottleneck of this implementation.

The reason for the bad performance of the mapping is the large amount of moved elements. Finding the enclosing tetrahedron for a given position is the operation that has the biggest impact on the performance of the mapping because it uses barycentric coordinates for the check. Each cube to check is split in 12 tetrahedrons thus, the method to calculate the barycentric coordinates of a point is called 69959 times with a given offset of 0.08. Using a non symmetrical split which only yields 5 tetrahedrons results in an overall speed up of 0.7 seconds. This is just a small adjustment but has a huge effect. Hence, a better way for identifying the eight enclosing elements needs to be implemented for mapping larger amounts of elements in an interactive fashion.

However, the amount of elements moved depends on two parameters: the scale factor for the initial deformation and the offset. The pictures in section 5.2 were created using an offset of 0.2 and a scale factor of 1.2. For this configuration the time needed for the deformation is $170ms$ and for the mapping of 328 moved elements is $351ms$. Hence, the overall time required is $521ms$ which is sufficient for interactivity.

6.2 Discussion

This work presented a flexible algorithm for fast direct volume deformation based on the works of *S. F. F. Gibson*, *M. A. Schill et al.* and *A. Neubauer*. It extended the STEPS system for transsphenoidal surgery simulation by adding deformations. It is able to calculate the deformation and directly manipulate the volume data without the need of some sort of intermediate step such as a surface extraction.

The algorithm used is fast enough for the amount of data to process to provide interactivity. The depth and size of a deformation can be configured by two parameters. However, although the algorithm supports modeling anisotropic data, the

simple configuration restricts the chain regions used to isotropic data but can easily be extended by adding parameters to specify individual offsets for each axis. The implemented algorithm also models inhomogeneous data.

The distorting effects caused by the z–scale factor was soothed using the analogy of a sound wave propagating through the object, by adjusting the shear constraints along the stretched axis and the z–component of the initial move vector.

To restrict the deformation and avoid unnaturally large deformations as well as the deformation of bone tissue a bone and a movement constraint were introduced.

This work also proposes an interface feasible for extending the STEPS application. It consists of three components responsible for the memory management, the calculation of the deformation and the mapping to and from the volume. This separation allows code reuse and increases the maintainability and easy integration into the STEPS system of the software. To allow a full separation of the mapping and the deformation calculation a simple DeformVoxel interface was introduced and the object ensures that all elements necessary for mapping are loaded. Hence, the mapper does not need to reload missing parts.

The deformations acquired with this implementation are reasonable although several problems were not solved.

Section 6.1 shows that the performance of the mapper is a bottleneck of the application for large amount of elements to be mapped. The limit is around $400 - 500$ elements. Another issue are the missing features such as relaxation and considering the shape of the modeled object during the calculation.

A detailed presentation of the current problems and the further work needed is presented in the next section.

6.3 Further Work

The further work needed can be split in two parts. The first part are improvements to the existing algorithm and the second part is adding new features to it.

Improvements

The improvements mainly concern the *performance of the mapping*. Currently, the way to find the eight enclosing elements for a given position is time consuming. A separation between the calculation of the barycentric coordinates and the check if a cube contains the requested position is required. Additionally, an equation solver with a better performance as the one currently implemented is desirable. The approach to find the nearest element for a given position, which is part of finding the enclosing eight elements, does not always return the correct element. This causes futile calculations and requires to find the nearest element using an

expensive search of the currently loaded elements. Another improvement required concerns the *multimove* which may lead to an *inconsistent ChainMail object*.

Desired Features

The new features focus on *enhancement of the realism*.

- **Relaxation**
  A relaxation is currently not implemented for performance reasons. One of the problems is that each relaxation requires a mapping step which is a bottle neck and that iso–values near the eye point of the endoscope need to be calculated to check if the eye point is still outside the tissue. An analysis of the problems can be found in section 3.4.6.
- **Avoid Clearing of Object**
  Currently, the object is cleared if a new deformation is started at a position outside of the currently loaded object. This allows the user to deform the volume without being restricted by the *movement constraint*. Therefore, another method to handle this case is required. The suggested way to do this is to keep the created objects in memory and merge them together as they expand. Here it must be ensured that the merged object satisfy the chain region constraints. This can be achieved if only non moved elements are allowed as surface elements which is the case in the current implementation.
- **Modeling the Object Shape**
  Regarding the *shape of the object during the calculation* is also a missing feature which creates artefacts such as the one shown in figure 5.11. The reason for this is that unconnected parts of the tissue are modeled as connected in the chain mail object. To handle this case separate *ChainMail objects* are required for each modeled unconnected tissue. Now the problem is that these objects may collide with each other during a deformation and hence a collision detection needs to be implemented. Modeling individual structures separately requires a segmentation step. The segmentation depends on the given threshold value for the iso surface. Hence, new way for adding missing parts of the volume to the existing object is required since a missing part might be the begin of an individual structure and a segmentation step has to be conducted each time the threshold values is changed because the topology might change too.

Despite the further work needed, this work produces reasonable results and shows that interactive direct volume deformation is possible in the given STEPS environment. This work is a starting point for further direct volume deformation extensions to the STEPS system.

Chapter 7

Summary

7.1 Introduction

Endoscopy

Minimally invasive procedures are medical applications of growing importance. The fields they are applied in include surgery, neurosurgery, gastroenterology, radiology and transsphenoidal tumor surgery. The benefits of this procedure are lower cost than traditional surgery and a less harming effect for the patient. Additionally, the risk of injuring the patient during a minimally invasive surgery is smaller than for a traditional surgery.

During a minimally invasive procedure a small endoscope is inserted into the region of interest along with a set of tools such as a rongeur for example. The little size of the endoscope allows the doctor to reach regions difficult to access, such as the sphenoid or the colon.

However, there are several drawbacks of endoscopy. The procedure is unpleasant for the patient and the risk of injuring the patient remains. Although the minimally invasive procedure is cheaper than the traditional surgery it is still expensive and due to the diameter of the endoscope some region of interest are inaccessible, such as small blood vessels.

Virtual Endoscopy

Virtual endoscopy was introduced to address these drawbacks. In a virtual endoscopy the surgeon relies on a 3D view of the patient's region of interest. The data for the visualization is typically acquired using tomography. The virtual reality endoscopy offers several benefits.

The doctor can reach every point of interest because the virtual endoscope is a point without dimensions. It is less harmful for the patient because it is a non invasive procedure. The patient does not need to be physically present for the diagnose

and the medial doctor can go through the data several times. This makes virtual endoscopy a tool for diagnosis, surgery planning and simulation, education, and inter–operative support.

The drawbacks of virtual endoscopy are first of all that a true realistic visualization of the data is not possible. Hence, the traditional endoscopy has to be used if the medical doctor is in doubt due to the inaccurate visualization. The physical behavior has to be modeled which is computational expensive. Hence, a tradeoff between interactivity, visual performance and physically plausible behavior has to be established. Finally, the patient's exposure to radiation during a CT[1] procedure presents a health risk.

Motivation

A. Neubauer et al. implemented a virtual endoscopy system for endonasal transsphenoidal pituitary surgery [Neubauer et al. '04]. They use a cell–based first–hit ray caster for direct volume based rendering of the data [Neubauer '01]. The ray caster supports interactive threshold changes which allows the user to interactively adjust the shape of the rendered surface to maximize the accuracy of the visualization.

The focus of our work is to extend the existing application called STEPS by *the integration of a deformation engine* to simulate the deformation of tissue. This was desired to enhance the realism of the application. The main requirements for the resulting deformation are that it is physically plausible and that the calculation is fast enough to allow interactive frame rates. The ray caster used allows interactive threshold changes. This feature must also be supported by the deformation engine.

Approach

Our *Divod ChainMail* algorithm, which is based on the ChainMail and Enhanced ChainMail algorithms, for direct volume deformation is presented in this work. Unlike other deformation algorithms we *directly model the volume* instead of objects. We use a Constrained Particle system to model the volume and calculate the deformation. For the *integration of our algorithm* into the STEPS application we developed a simple interface which enforces flexibility and code reusability.

The advantages of virtual endoscopy overweight the drawbacks and a lot of research has been conducted to resolve the remaining drawbacks. The results of this related research are presented in the next section.

7.2 Related Work

This section provides an overview of the research conducted in the fields of physical modeling and virtual endoscopy.

[1] Computer Tomography

Classification

Methods for physical modeling can be divided into two categories.

- physically based: the deformation and interaction of objects is calculated using laws of physics such as Newton's law of gravity.
- geometrically based: deform the object by moving it's defining control points, without the application of physical laws.

Physically based algorithms produce more realistic results than the geometrically base approaches. However, the high computational cost of physically based models often does not meet the requirement for interactivity. Thus, the faster geometrically based models are often applied. Another thing to consider is interaction with the object. For geometrically based models intuitive interaction is hard to archive because the shape of the object is indirectly defined by control points.

Mass Spring Model

One of the most popular methods for physical modeling is the *Mass Spring Model*, because of its speed and flexibility. In this model objects are defined through mass points which are connected by dampened springs. Hence, if a force is applied to a mass point it is propagated to the other mass points through the springs. The KISMET endoscopy simulation system uses this approach for physical modeling [Kühnapfel et al. '00]. A system similar to the Mass Spring Model is introduced by *M. Teschner et al.*. They derive three distinct forces from the potential energies of mass points which arranged in a tetrahedral mesh to calculate the deformation [Teschner et al. '04].

Finite Element Method

Another widely used approach is the so called *Finite Element Method* (FEM). It is a physically based and computational expensive method which yields very realistic results. To integrate it into interactive applications a lot of effort has been taken to optimize and decrease computation time. For example *Q. Zhu* uses the FEM to simulate macroscopic dynamics of muscle [Zhu '98]. FEM based models require a preprocessing step which depends on the topology of the object. This preprocessing step is also necessary for Mass Spring System based approaches if meshes are used as data structure. Hence, changes to the topology such as cutting have to be avoided. This problem is addressed by *H. W. Nienhuys* and *A. F. van der Strappen* [Nienhuys, Strappen '96].

Radial Basis Functions

The *Radial Basis Functions* (RBFs) are a geometrically based approach. It uses radial functions as interpolation functions for a set of control points. RBFs can be

divided into three classes [Kojekine et al. '02]

- Native Methods: these methods are computationally very expensive and not feasibly for interactive applications.
- Fast Methods: fulfill the requirements of interactive applications but are restricted to small problems.
- Compactly Supported Radial Basis Functions (CSRBFs): they combine the advantages of the two latter classes. Hence, they are fast enough for interactive applications but not restricted to a certain set of problems.

The CSRBFs are further optimized for speed and memory consumption by *N. Kojekine at al.*. *J. C. Carr et al.* use CSRBFs in medical imaging. They visualize human skull from depth maps even across defect areas [Carr et al. '97].

Free Form Deformation

Another fast geometrically base approach introduced by *T. Sederberg* and *S. Parry* is *Free Form Deformation* (FFD). *G. Hirota et al.* describe a way to add physically based extensions [Hirota et al. '99]. They add the rule of preservation of mass that governs the deformation. Models to ease direct intuitive interaction are presented by *P. Borell* and *D. Bechmann* [Borell, Bechmann '91] as well as *W. S. Hsu et al.* [Hsu et al. '92]. A FFD based model is applied in the laproscopic surgery simulation system implemented by *C. Basdogan et al.* [Basdogan et al. '98]. They decided to use FFDs due to their good performance. *G. Sela et al.* introduced a new type of FFDs called DFFDs which support discontinuities and allow topology changes due to cuts [Sela et al. '04].

ChainMail Algorithm

This work is further based on the ChainMail algorithm [Gibson '97] presented by *S. F. F. Gibson*. It is a fast and flexible algorithm for the deformation of volumetric objects that supports modeling different material properties and anisotropic material. *M. A. Schill et al.* extended this approach to inhomogeneous material with the Enhanced ChainMail algorithm [Schill et al. '98] and the restriction to rectilinear grids is overcome by the Generalised ChainMail algorithm[Y. Li '03] introduced by *Y. Li* and *K. Brodlie*.

The next section presents the *Divod ChainMail* algorithm as well as the ChainMail and Enhanced ChainMail algorithm this work is based on.

7.3 *Divod ChainMail* algorithm

The Direct Volume Deformation ChainMail algorithm (*Divod ChainMail* algorithm) consists of three parts.

- Deformation: This part is responsible for calculating the deformation. It is based on the original ChainMail and Enhanced ChainMail algorithms. However, some adjustments were applied to the original algorithms to integrate them into the existing application.
- Mapping: The mapping is the link between the object and the application's ray caster. The object needs to be mapped to the volume which is then visualized. Hence, the mapping needs to be fast and the quality of the visualization depends on the mapping algorithm used.
- Memory Management: Due to the large amount of data the *ChainMail object* can not model the entire volume because it would exceed available memory. Therefore, we implemented a mechanism for memory management.

Original ChainMail

In the original ChainMail algorithm an object is modeled through a set of elements which are connected by a rectilinear grid. Hence, each element has a left, right, top, bottom, front and back neighbor element. Each element may hold material properties such as color, temperature and density. Additionally, each element has a set of region constraints which define the valid region a neighbor element has to lie in. These constraints are $minDx, maxDxshearX, minDy, maxDy, shearY$ for the 2D case. Please see 3.1 for a detailed description. In the 3D case the constraints $minDz, maxDz$ and $shearZ$ are added.

Such an object can be compared to a chain mail in the 2D case. If a single chain element of a chain mail is moved only a short distance it does not influence the neighboring elements but if the movement length is big enough it forces the neighboring chain elements to move too. The same behavior applies to the elements of a *ChainMail object*. The chain regions define how far an element can be moved without forcing the neighbors to be moved too.

A deformation is initialized by moving a single element. After the first element is moved, its neighbors are added to candidates lists to check if they violate the chain region constraints and need to be moved too. There are 4 candidate lists in the 2D case: right, left, top, bottom. In the 3D case two additional lists for the front and back neighbors are needed.

Candidate Processing

If a candidate needs to be moved its neighbors are also added to the candidates list since they might violate the constraints too. In the 2D case, candidates from the left list add their top, bottom and left neighbors, candidates from the right list add their right top and bottom neighbors, candidates from the top list only add their top neighbor and candidates from the bottom list only add the bottom neighbor to the corresponding list. In the 3D case candidates from the left, right, top and bottom

list also add their front and back neighbors. The candidates from the front list only add their front neighbor. The candidates from the back list only add their back neighbor.

The algorithm continues by first processing the right, then the left, the top and finally the bottom candidate list until all lists are exhausted.

The speed of the original ChainMail algorithm, which is capable of modeling several thousand elements, roots in the fact that each element is processed at most once. However, the theorem this finding is based on, restricts the ChainMail algorithm to homogeneous data. Thus, all elements of a *ChainMail object* need to have the same chain regions.

Enhanced ChainMail

To overcome this restriction *M. A. Schill et al.* used the analogy of a sound wave propagating though material. They found that the sound wave propagates faster through stiffer material and used this observation to change the way the candidate elements are processed. To ensure that the deformation propagates through stiffer material fastest they replace the four candidate lists by a single ordered list. The ordering criterion is the amount of constraint violation, where the element with the biggest violation is processed first. This allows the modeling of inhomogeneous data through assigning each element an individual chain region.

Shortcomings

In the original ChainMail algorithm the deformation is initialized by moving a single element. Since the user interacts with the visualized surface and is unaware of the elements a different approach for initiating the movement is required. The approach implemented in this algorithm is moving the eight enclosing elements of the position where the endoscope intersects the surface. The eight elements are equally moved by the force vector which is the vector between the endoscope's eye point and the intersection point. With this procedure the eye point is likely to lie outside of the tissue after the deformation and the diameter of the endoscope is modeled.

Another problem arises from the z–scale in the volume data set. The distance between two elements along the x–axis and y–axis is one distance unit but along the z–axis it is z–scale times distance unit. This causes a distortion in the form of a stretch of the resulting deformation along the z–axis. To overcome this problem the z–component of the initial force vector is divided by z–scale and the chain region shear constraints for the x and y shear are multiplied by z–scale for the neighbors along the z–axis.

Finally, to avoid unnaturally large displacement of the tissue and the deformation of bone tissue we introduced two constraints. A *movement constraint* that restricts the length an element may be moved and a *bone constraint* that restricts the movement

of elements with density values that represent bone tissue. If one of these constraints is violated the movement is reverted and retried with a smaller initial step size.

Mapping

The mapping is the second integral part of the algorithm. Because the ray caster directly renders the volume it can be used to clearly separate the deformation engine from the STEPS application. Thus, it is necessary to update the volume data by mapping the deformed object. For the initial creation of the object it is also necessary to map the volume data to the object.

Each object element represents one volume voxel. The mapping to the object is done by assigning each new element the density value of the corresponding volume voxel.

The mapping to the volume uses barycentric coordinates. First, the eight enclosing elements of the volume position to be updated are acquired. Then a virtual middle element of these eight elements is calculated to allow a symmetrical split of the enclosing cube to twelve tetrahedrons. The symmetrical split is necessary to produce symmetrically visual results.

One of the twelve tetrahedrons is enclosing the point to be updated. This tetrahedron is used to calculate the barycentric coordinates of the corresponding point which are subsequently used for the calculation of the new density value for the given position.

Memory Management

A single object element requires approximately 250 bytes. Hence, due to the large amount of volume voxels ($512 \times 512 \times 128$) it is not possible to map every volume voxel to an object element. This would exceed the available memory.

Therefore, a memory management mechanism has been implemented. The memory management exploits the fact that the deformation propagates locally and outwards through the object. The volume grid is divided into a macro grid where a macro grid cell has the shape of a cube and consists of $size^3$ volume grid cells. A volume grid cell consists of eight elements which form a cube of the size 1.

The *ChainMail object* consists of cubes which are called *cubicles* . Each *cubicle* models one macro grid element. If the deformation exceeds the currently loaded object, the missing part is loaded in the form of a *cubicle* which elements are initialized using the mapper.

If a deformation propagates out of volume bounds, the memory manager produces virtual *cubicles* with the elements density value set to zero. This allows the algorithm to proceed unaware of the problem.

Initially the object is empty. Thus, the first *cubicle* is loaded with the start of the deformation and encloses the intersection point of the endoscope and the surface. If a subsequent deformation occurs, the object checks if it already holds the position

of the intersection. If so, the deformation is calculated using the existing object. If the position is not loaded, the current object is cleared and a new object is started by loading the first *cubicle* . This is done to avoid large objects and to avoid the problem of merging two objects which requires collision detection. However, this procedure allows the user to elude the movement constraint. If the user starts a new deformation at a different position the current object gets cleared. If the user then returns to the already deformed position a new object is created with reset movement constraints which allows a larger deformation than permitted.

Algorithm Outline

Initialization

The deformation is initialized by moving the eight enclosing elements for a given position, typically the point of collision. Due to initially moving eight elements, candidate elements would be added multiple times to the candidate list. To avoid duplicates each element holds a flag that stores the direction in which neighbors must not be added to the list. The neighbors that lie in these ignoring directions are added to the candidate list by one of the other seven initially moved elements.

This approach has a drawback because it allows inconsistencies within the *ChainMail object*. However, these inconsistencies have no impact on the visual outcome and are very unlikely to occur as laid out in section 3.5.3.

In case of bone or the movement constraint violations the current deformation is reverted. All elements in the moved list are reset to their last position. The deformation is then retried with a smaller step size.

Chain Regions

The chain regions used in this approach have a cubic shape which is governed by an offset parameter and the density value of the modeled element to model inhomogeneous data. The density value is used to scale the offset value: $offset_{scaled} = offset * (element_{value}/4096)^2$ The shear factors are set to the scaled offset value. $shear = offset_{scaled}$ The min and max constraints are given as $min = 1 - offset_{scaled}$ and $max = 1 + offset_{scaled}$. Hence, the offset has to be between 0 and 1 because the initial distance of an element to the neighbors is 1 and the initial object has to be in a valid state. With this configuration the stiffer material has smaller chain regions which results in smoother deformations.

Since stiff material does not deform as easily as soft material the movement constraint mc for a single element is also influenced by the density value of the element e given as:$mc = (1 - (e_{value}/4096)^2) * movement_{constraint}$ where $movement_{constraint}$ is the global movement constraint.

A configuration file allows to tune the parameters for the given data set.

Calculation of new Density Value

The first mapping algorithm implemented caused artefacts in the form of holes inside the volume. The currently implemented mapping uses barycentric coordinates to calculate the new density value of a voxel.

The first step in this procedure is to find the nearest element for the position to be updated. This is done because the nearest element must be one of the eight enclosing elements for the given position. Hence, all eight cubes that can be formed with the nearest element as corner element are checked if they enclose the position. The check is done using barycentric coordinates.

The cube is split into twelve tetrahedrons in a symmetrical manner. A symmetrical split yields symmetrical visual results. Then the barycentric coordinates for the point are calculated and checked if each component of the coordinates satisfies the condition $0 \leq b_{component} \leq 1$ which means that the tetrahedron encloses the element. The barycentric coordinates (b_x, b_y, b_z, b_w) and the density values of the enclosing tetrahedron points (V_1, V_2, V_3, V_4) are then applied to calculate the new density value given as $density = b_x * V_1 + b_y * V_2 + b_z * V_3 + b_w * V_4$.

To symmetrically split the cube a virtual middle element M is calculated. For example, the split of the bottom surface consisting of the points P_0, P_1, P_2, P_3 yields the 2 tetrahedrons (M, P_0, P_1, P_2) and (M, P_0, P_2, P_3).

Barycentric Coordinates

The barycentric coordinates are calculated by solving a set of four equations. Currently, a non optimized version of the Gaussian algorithm is implemented. However, the drawback of this approach is that the barycentric coordinates are also calculated for cubes that do not contain the voxel. This results in a large number of unnecessary calculations and drastically reduces performance.

A similar problem exists with the algorithm implemented to find the nearest element for a given position. It does not always return the correct element because the implemented algorithm may terminate prematurely. In this case an expensive second try which checks all loaded elements is necessary to find the correct nearest element.

However, the performance for the given problem is acceptable and allows interactivity.

Loading a new Cubicle

To create a new *cubicle* it is necessary to initialize the new elements. Hence, the memory manager requires the mapper to read the corresponding density values.

The memory manager also links the elements together. That is, it creates the appropriate neighborhood links for the elements within the new *cubicle* loaded.

Connecting a new Cubicle to the existing Object

The *ChainMail object* then connects the new *cubicle* to the already loaded parts. That requires establishing the neighborhood links for the opposing surface elements. For example, if a *cubicle* is added at the right side of the existing object the elements on the right surface of the object's corresponding *cubicle* have to be connected with the elements on the left side of the new *cubicle* . It is important to note that the other five possible neighbor *cubicles* have to be connected as well.

If a deformation propagates out of volume bounds the memory manager creates virtual chunks with elements initialized with zero so that the deformation calculation proceeds unaware of the problem.

7.4 Implementation

Interface

The purpose of this work is to integrate a deformation engine into the STEPS application. The main focus is on implementing a first prototype. Hence, lot of further work is expected to be done. Therefore, a basic interface for deformation engines to be integrated into the existing application was developed.

The interface models the three components of the algorithm.

- Deformation
- Mapping
- Memory Management

To fully separate the mapping from the memory management and the deformation a basic DeformVoxel interface is proposed. It holds the information about the position, value and the neighborhood of an element which is sufficient for a wide range of mapping algorithms.

Another thing that is emphasized on is an easy integration into the existing system. This is achieved by offering a single method which calculates the deformation and updates the volume date. Secondly, this method does not propagate errors. Hence, additional error handling is not required when integrating the deformation engine.

The data passed from the STEPS application to the deformation engine consists of a force vector and the intersection point between the endoscope and the iso surface. It is calculated in the isCollision() method of the STEPS implementation which checks if a collision between the endoscope and the iso surface occurred.

Integration

The deformation engine is integrated by creating the mapper, the memory manager and the deformation object. Then the method performDeformation() is called in the

isCollision() call passing the force vector and the intersection point to the deform engine.

The intersection point is calculated iteratively. First, the point is assumed at the eye point of the endoscope. Then it is repeatedly moved towards the last position of the eye point by a small step size until it lies outside the tissue. The middle of the last point inside the tissue and the first point outside the tissue is taken as the intersection point. The vector from the intersection point to the eye point is the force vector.

7.5 Results

The performance of the implementation is manly determined by the mapper. For 1000 moved elements the mapper needs $1,17$ seconds to map the deformed object to the volume. The time required to calculate the deformation is 230 milliseconds. Additionally, the time required for the calculation of the deformation does not increase drastically even for large numbers of moved elements. For 1888 moved elements it requires 331 milliseconds. The time required by the mapping drastically increases with the number of moved elements. It already takes $4,76$ seconds to update the volume for 1888 elements.

The reason for this decrease in performance is the way the enclosing cube for a given position is calculated. As shown in section 3.6.3 a lot of futile calculations are conducted to find the enclosing cube and tetrahedron. Hence, a faster way to solve this problem is required for larger amounts of moved elements.

The number of elements moved depends on the ratio between the offset and the scale factor for the initial movement. The offset governs how many elements are dragged where smaller offsets cause more dragging. The scale factor defines the size of the initial movement. A large initial movement also causes more elements to be dragged.

In the current implementation the offset is set to 0.2 and the scale factor is set to 1.2. The performance for these parameters is $170ms$ for the deformation and for the mapping $351ms$. This sums up to $521ms$ for a single deformation step and is sufficient to give an interactive impression.

7.6 Conclusion

The presented work shows an approach to integrate a deformation engine into the existing STEPS application. The results of this prototype are already promising although a lot of optimizations are still required the performance is sufficient for interactivity.

Large numbers of moved elements have to be avoided because the mapping which is the performance bottleneck requires over a second to map 1000 moved elements and the computational time increases drastically with the number of elements to map.

However, the current configuration does not require such large numbers of elements to be moved resulting in a sufficient performance.

The described interface allows code reuse and enhances the flexibility of the software. For example the mapping component is independent from the deformation and the memory management and can be interchanged. Easy integration is achieved by handling all errors internally and providing a single method which calculates the deformation and updates the volume data. Hence, the ray casting algorithm stays unaware of the new engine.

Further work includes optimizations of the the implementation and adding new features. Most of the optimizations focus on the mapper and the way the eight enclosing elements for a given position are found.

Desired new features include a better *multimove* strategy that avoids inconsistencies as well as a way for collision detection and merging *ChainMail objects* so that currently unused objects are not removed from memory which allows to ignore the movement constraint. The collision detection is also necessary if the different types of tissue are modeled individually through multiple *ChainMail objects*. That also requires a segmentation of the data. The artefacts that are created because the shape of the object is not modeled could then be avoided. For example two separate structures are currently modeled by a single *ChainMail object*. Hence, a deformation of one structure in the object also influences the other structure which is unrealistic behavior. Furthermore, a relaxation step would further increase the realism of the application.

This work presented our *Divod ChainMail* algorithm for fast direct volume deformation. It also described the *integration of the deformation engine* prototype into the existing STEPS application. Although there is still a lot of room for improvement concerning the performance and the features of the implementation the requirement of an interactive impression is met together with good visual results and the work presents a starting point for further direct volume deformation applications in the STEPS environment.

Chapter 8

Acknowledgements

I want to thank André Neubauer for his patience, his advise, and the time he spent helping me. Further I was to thank Katja Bühler for her patience, her advise and for inviting me to the CESGC in the year 2004 where I had a fabulous time (http://www.cg.tuwien.ac.at/studentwork/CESCG/). I also want to thank Eduard Gröller for his support as well as Erich Liebmann and Bernhard Schiemann who gave valuable hints. Further I want to thank the VRVis research center (http://www.VRVis.at), where parts of this work have been done as part of the basic research on visualization (http://www.VRVis.at/medvis/), for supporting my work, as well as Tiani Medgraph (http://www.tiani.com) for support and providing medical data sets. I would also like to give thanks to Sylvia Mittermann for support as well as Peter Pongratz and Mario Gatto for being there. Finally, I want to thank my parents for there enduring financial and emotional support throughout the years.

Bibliography

D. Baraff (2001). *SIGGRAPH Coursenotes - Physically Based Modeling, Collision and Contact (slides only).* Pixar Animation Studios.

A. V. Bartrolí, R. Wegenkittl, A. König, E. Gröller, E. Sorantin (2001). Virtual Colon Flattening. *Data Visualization, Proceedings of Symposium on Visualization*, S. 127 – 136.

D. Bartz (2003). Virtual Endoscopy in Research and Clinical Practice. *STAR - State of the Art Report, Proceedings of EUROGRAPHICS 2003.*

C. Basdogan, C. Ho, M. A. Srinivasan, S. D. Small, S. L. Dawson (1998). Force interactions in laparoscopic simulations: Haptic rendering of soft tissues. *Proceedings of MMVR 1998*, S. 385 – 391.

P. Borell, D. Bechmann (1991). Deformation of n-dimensional objects. *Proceedings of ACM Symposium on Solid Modeling*, S. 351 – 370.

M. Bro-Nielsen, S. Cotin (2001). Real-time Volumetric Deformable Models for Surgery Simulation using Finite Elements and Condensation. *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, Lecture Notes in Computer Sience 2208.

J. C. Carr, W. R. Fright, R. K. Beatson (1997). Surface Interpolation with Radial Basis Functions for Medical Imaging. *Proceedings of IEEE Transaction on Medical Imaging*, 16:96–107.

B. Eberhardt, O. Etzmuss, M. Hauth (2000). Implicit-Explicit Schemes for Fast Animation with Particle Systems. In *Eurographics Computer Animation and Simulation Workshop 2000*, 2000.

F. Ganovelli, P. Cignoniz, C. Montanix, R. Scopigno (2000). A Multiresolution Model for Soft Objects supporting interactive cuts and lacerations. *Proceedings of EUROGRAPHICS 2000*, 19(3).

S. F. F. Gibson (1997). 3D ChainMail: a Fast Algorithm for Deforming Volumetric Objects. *Proceedings of Symposium on interactive 3D Graphics, ACM SIGGRAPH*, S. 149–154.

D. Hearn, M. P. Baker (1997). *Computer Graphics C Version.* Prentice Hall.

G. Hirota, R. Maheshwari, M. C. Lin (1999). Fast Volume-Preserving Free From Deformation Using Multi-Level Optimization. *Proceedings of ACM Solid Modeling.*

W. S. Hsu, J. S. Hughes, H. Kaufmann (1992). Direct Manipulation of Free-Form Deformations. *Proceedings of Computer Graphics*, 26(2):177 – 184.

N. Kojekine, V. Savchenko, M. Senin, I. Hagiwara (2002). Real-time 3D Deformations by Means of Compactly Supported Radial Basis Functions. *Proceedings of EUROGRAPHICS 2002.*

U. G. Kühnapfel, H. K. Cakmak, H. Maaß (2000). Endoscopic Surgery Training using Virtual Reality and deformable Tissue Simulation. *Proceedings of Computers & Graphics*, 24:671–682.

A. Laghi, P. Pavone, V. Panebianco, I. Carbone, L. Francone (1999). Volume-rendered Virtual Colonoscopy: Preliminary Clinical Experience. *Proceedings of Computer Assisted Radiology and Surgery*, S. 171 – 275.

M. Nakao, T. Kuroda, H. Oyama, M. Komori, T. Matuda, T. Takahashi (2003). Physically-Based Fine and Interactive Soft Tissue Cutting. *IPSJ JOURNAL4*, 44(8).

L. P. Nedel, D. Thalmann (1998). Real Time Muscle Deformations Using Mass–Spring Systems. *Proceedings of CGI 1998*, S. 156–165.

A. Neubauer, S. Wolfsberger, M.-Thérèse Forster, L. Mroz, R. Wegenkittl, K. Bühler (2004). STEPS - an Application for Simulation of Transsphenoidal Endonasal Pituitary Surgery. *Proceedings of IEEE Visualization 2004*, S. 513–520.

A. Neubauer (2001). *Master Thesis, Cell-Based First-Hit Ray Casting.* Institute of Computer Graphics, Vienna University of Technology.

H. W. Nienhuys, A. F. van der Strappen (1996). A sugrey simulation supporting cuts and finiteelement deformation. *Proceedings of Computer Graphics Forum*, 15(4):C57 – C66.

J. O'Brien, A. Bargteil, J. Hodgins (2002). Graphical Modeling and Animation of Ductile Fracture. *Proceedings of SIGGRAPH 2002*, S. 291 – 294.

O.Ennemoser, H. Canaval, W. Ambach (1986). Computerised tomography (CT) in education: a demonstration experiment for students. *European Journal of Physics*, 7:88–90.

P. Reuter, I. Tobor, C. Schlick, S. Dedieu (2003). Point-based Modelling and Rendering using Radial Basis Functions. *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Austalasia and South East Asia (Graphite 2003)*, S. 111–118.

V. Savchenko, L. Schmitt (2001). Reconsturcting Occlusal Surfaces of Teeth Using a Genetic Algortihm with Simulated Annealing Type Selection. *Proceedings of 6th ACM Symposium on Solid Modeling and Application*, S. 39–46.

M. A. Schill, S. F. F. Gibson, H.-J. Bender, R. Männer (1998). Biomechanical Simulation of the Vitreous Humor in the Eye Using Enhanced ChainMail Algorithm. *Proceedings of Medical Image Computation and Computer Assisted Interventions (MICCAI) 1998*, S. 679–687.

T. Sederberg, S. Parry (1986). Free Form Deformation of Solid Geometric Models. *ACM Computer Graphics, Proceedings of SIGGRAPH 1986 Proceedings*, 20:151 – 160.

G. Sela, S. Schein, G. Elber (2004). Real-time Incision Simulation Using Discontinuous Free Form Deformation. *Proceedings of International Symposium on Mediacal Simulation 2004*.

D. Terzopoulos, K. Fleischer (1988). Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture. *Proceedings of SIGGRAPH 1988*, S. 269 – 278.

D. Terzopoulos, K. Waters (1990). Physically-Based Facial Modeling, Analysis, and Animation. *The Journal of Visualization and Computer Animation*, 1:73 – 80.

M. Teschner, B. Heidelberger, M. Müller, M. Gross (2004). A Versatile and Robust Model for Geometrically Complex Deformable Solids. *Proceedings of CGI 2004*, S. 312–319.

L. Verlet (1967). Computer Experiments on Classical Fluids. Ii. Equilibrium Correlation Functions. *Physical Review*, 165:201 – 204.

H. Wendland (1995). Piecewise polynomial, positive defined and compactly supported radial functions of minimal degree. *Proceedings of AICM*, 4:389–396.

A. Witkin, D. Baraff (2001). *SIGGRAPH Coursenotes - Physically Based Modeling, Differential Equatoin Basics*. Pixar Animation Studios.

A. Witkin (2001a). *SIGGRAPH Coursenotes - Physically Based Modeling, Constrained Dynamics*. Pixar Animation Studios.

A. Witkin (2001b). *SIGGRAPH Coursenotes - Physically Based Modeling, Particle System Dynamics*. Pixar Animation Studios.

K. Brodlie Y. Li (2003). Soft Object Modelling with Generalised ChainMail – Extending the Boundaries of Web-based Graphics. *Proceedings of Coputer Graphics Forum*, 22(4):717–727.

Q. Zhu (1998). *Masters Project Final Report, 3D Voxel-Based Muscle Volume Deformation by Finite Element Method*. Department of Computer Sience, SUNY at Stony Brook, New York.