NTNU
Innovation and Creativity

# Tetrahedral mesh for needle insertion

**Rolf Anders Syvertsen**

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Torbjørn Hallgren, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Making a tetrahedral mesh for use in a medical needle insertion simulator. The mesh is going to be used in a finite element method for soft tissue deformation.

Assignment given: 20. January 2007
Supervisor: Torbjørn Hallgren, IDI

# Tetrahedral Mesh For Needle Insertion.

Master's thesis

Rolf Anders Syvertsen

email:rolfans@gmail.com

---

June 18, 2007

Teacher: Torbjørn Hallgren

www.idi.ntnu.no


The Norwegian University of Science and Technology (NTNU)
The Department of Computer and Information Science (IDI)

## Abstract

This is a Master's thesis in how to make a *tetrahedral mesh* for use in a *needle insertion simulator*. It also describes how it is possible to make the simulator, and how to improve it to make it as realistic as possible. The medical simulator uses a haptic device, a haptic scene graph and a *FEM*[1] for realistic soft tissue deformation and interaction. In this project a tetrahedral mesh is created from a polygon model, and then the mesh has been loaded into the *HaptX*[1] haptic scene graph. The objects in the mesh have been made as different haptic objects, and then they have got a simple haptic surface to make it possible to touch them. There has not been implemented any code for the *Hybrid Condensed FEM* that has been described.


[1] **F**inite **E**lement **M**ethod

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

This report describes how to make a medical simulator for needle insertion using a haptic device and a *FEM*[1]. The *FEM* uses *tetrahedrons* to model the volume and the objects in the model. The goal is to make a simulator that is capable of teaching medical students how to insert a syringe into a human body. To do this the simulator needs to be as close to a real needle insertion as possible. For optimal results the model of the body needs to deform and act as it would in a real needle insertion.

The task of making a complete simulator is in it self to big too complete with this few resources. So in this project the goal has been reduced to make a *tetrahedral mesh* that is suitable to use with *FEM* in a medical simulator. When that mesh has been made, hopefully it is possible to load it into a haptic scene graph. It is unrealistic to set any bigger goals for this project, since there is just to much work that needs to be done to complete the simulator, and even more work to make it good enough for teaching purposes.

As it is today the students learn to insert a syringe by using parts of a dead person connected to a computer. This simulator has the drawback that the test specimen is stiff. And when the students uses it they leave marks and holes in it. So that the other students can see the holes and use them. This means that the model needs to be exchanged after a little while. To replace the model a new specimen is required, which is not that easy to come by. Before the specimen can be used in the simulator it needs to be prepared. This means that it needs to be scanned and segmented before it is possible to use it in the simulator. This because every human is different, there are not two persons that are alike, except for identical twins.

The other alternative for learning how to insert a needle is to use the other students as test subjects. By making a simulator that is capable of teaching them needle insertion in an augmented reality it could improve their skills. And they would be less likely to insert the syringe wrong and harming their fellow students and patients.

The data model for the simulator should be made from data from real persons to make them as realistic as possible. MRI[12][2] or CT[13][3] scanners can make data scans of persons that could

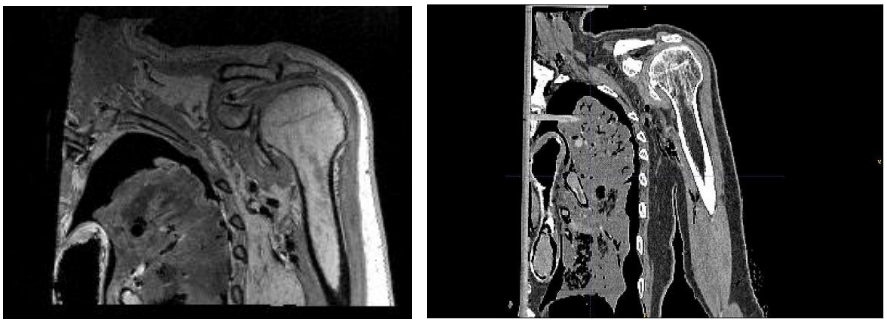
[1] Fenite Element Method

[2] Magnetic resonance imaging

[3] Computerised Tomography, also referred to as CAT (Computerised Axial Tomography)

be used as models in the simulator. This project is trying to make a simulator for insertion into the shoulder region. The goal is to hit the membrane between the Humerus[4] and Scapula[5] bones. It should be possible and easy to change the data model making it possible to simulate on a different body part like the knee.

Figure 1.1 shows a MRI and a CT image from a data set of the shoulder. The MRI data set has higher resolution and better contrasts then the CT scan, and is therefore the best option to use when making a model for a simulator.



(a) MRI (b) CT

Figure 1.1: MRI and CT image from a shoulder volume.

There is also the possibility of making the data model from a polygon model. The polygon model would then need to have all the parts that is wanted for that simulation. 3D models like this is possible to buy over the internet.

To make the simulator realistic the soft tissue should be able to be deformed when it is touched, and the forces from the tissue on the syringe should be as close to real as possible. To make this happen there has to be implemented a way to deform the model. There are many ways to do this. In this project the use of a *FEM* has been selected.

[4]Bone in the upper arm
[5]The shoulder blade

# Chapter 2

# About this project

This work is based on my project "Needle Insertion Simulator Applying A Haptic Device" [14] and the articles about "A Virtual Reality Training System for Knee Arthroscopic Surgery" [15] and "An improved scheme of an interactive finite element model for 3D soft-tissue cutting and deformation" [7].

In "Needle Insertion Simulator Applying A Haptic Device" [14] the different solutions of how to make a simulator has been described. The most realistic aproch is to make use of a FEM, the best one for this type of program seams to be the one described in [15] and improved in [7]. The *Hybrid Condensed FEM* described in Chapter 3.2.3.

Here is some information about things that is needed for this project. Both hardware, software and practical information.

## 2.1 Haptic Devices

A haptic device is a tool to give feedback to the user on the forces that interacts with the augmented reality on the computer. The device has multiple degrees of freedom that is controlled by motors. The motors set powers on the stylus to simulate different environments, like hitting a hard surface, spring forces, rough surfaces and more. It is possible to use two series of haptic devices for this program, it is the SensAble Phantom[16] series and the Novint Falcon[2]. Figure 2.1 shows the SensAble Phantom Desktop and Omni, and Figure 2.2 shows the Novint Falcon.

(a) Phantom Desktop (b) Phantom Omni

Figure 2.1: SensAble Phantom Haptic Devices. From www.sensable.com

Figure 2.2: Novint Falcon Haptic Device. From [2]

The haptic devices that are best fitted for use in a needle insertion simulator is the SensAble Phantom haptic devices. It is a SensAble Phantom that is going to be used in this project. But they are not perfect, they do not have a motor on the last link before the stylus, which means that there is nothing to stop the user from turning the stylus around in this link.

## 2.2 Haptic Scene Graphs

This project needs a haptic scene graph. There are a few haptic scene graphs to choose from, like *H3D*[3], *OSGHaptics*[4], *Reachin API*[17], *HaptX*[1] and more. Here are some information about some of the different haptic scene graphs.

### 2.2.1 H3D

The *H3D*[3] haptic scene graph is an open source scene graph. It is programed in C++, but it uses *X3D*[18] and *Python*[19] scripts to program it. Since C++ is wanted as the programming language for the scene graph H3D was not selected. Figure 2.3 show an image from H3D deform rectangle demo.

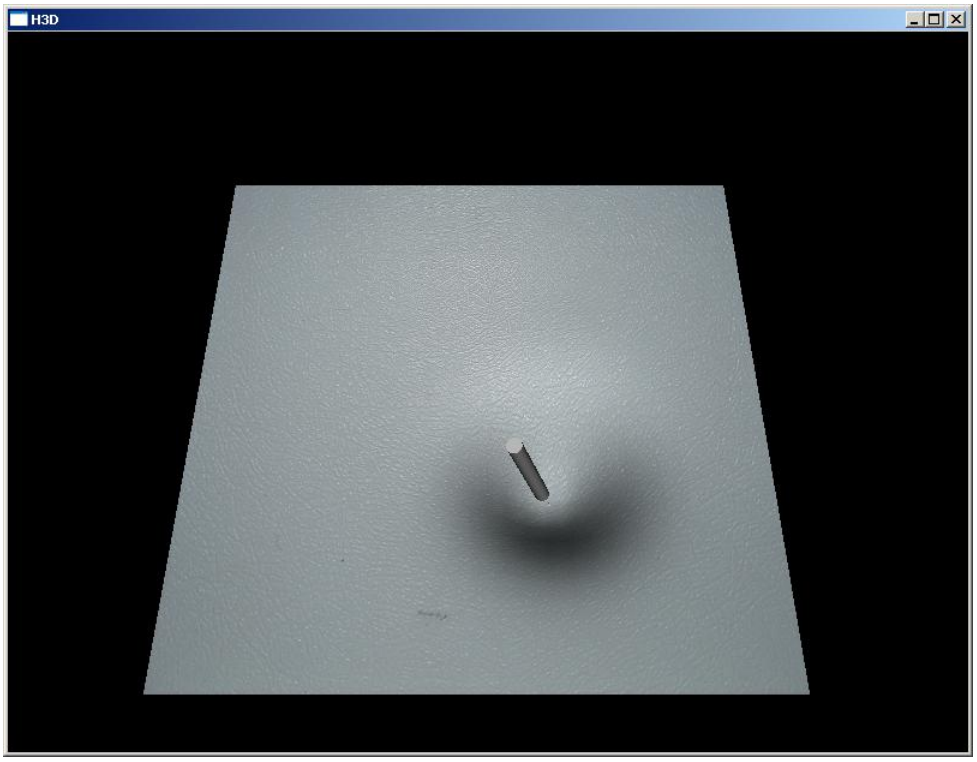



Figure 2.3: H3D deform demo. From [3]

### 2.2.2 OSGHaptics

*OSGHaptics*[4] is a haptic add on to the *Open Scene Graph (OSG)*[20] scene graph. OSG is an open source scene graph that is used in many different 3D applications. It is programed in C++, and the scene is programed in C++. It has built in a lot of cool 3D effects, like fire, explosions, shadows and more. The *OSGHaptics* addon adds the possibility to add a haptic scene to this scene graph. The main drawback here is that it is not implemented into the scene graph, but an add on. This means that not all the things in the scene graph works together with the haptic scene graph. And some things are a little messy. *OSG* has the advantage that it is extensive and well used as a scene graph. This means that it supports a lot of different 3D file formats, it supports and has a lot of addon libraries and that new functionalities is added all the time. Because *OSGHaptics* is not a part of the core library this powerful scene graph has not been selected as the haptic scene graph for this project. Figure 2.4 shows an image from the material demo.

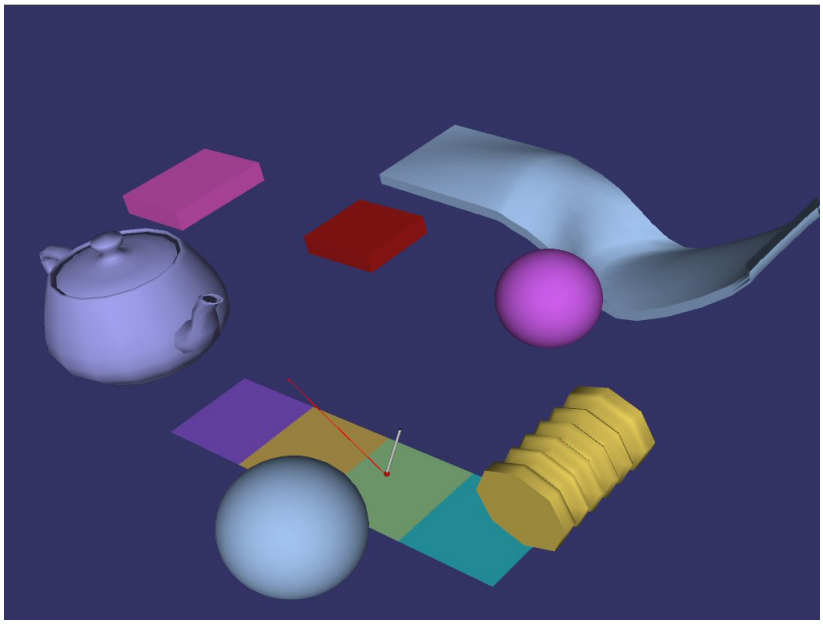
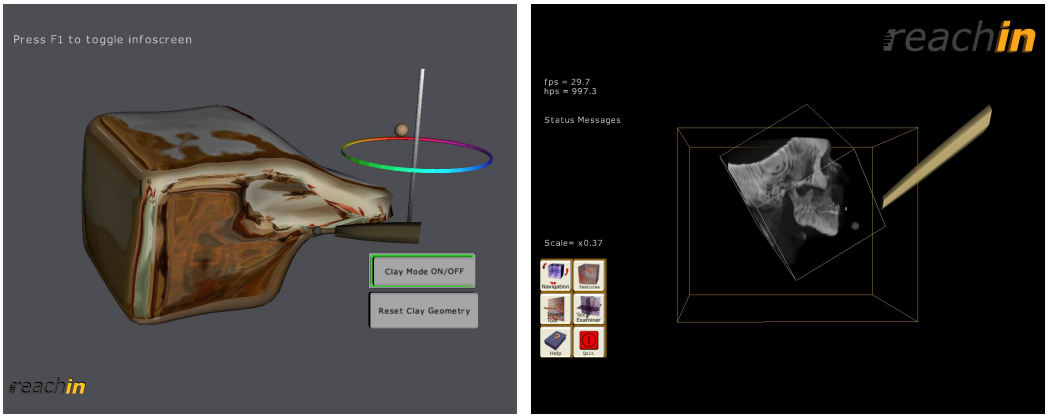


Figure 2.4: OSGHaptics material demo. From [4]

### 2.2.3 Reachin

At the beginning of this project the haptic scene graph that was intended to be used was the *Reachin API*[17] version 4.1. This scene graph is made up around *VRML*[1][21]. It is possible to define tetrahedral meshes in VRML, but then only as a series of faces and vertexes, not as a series of tetrahedrons. Reachin API is written in C++ and can be programed in *C++* or with *VRML* and *Python*[19] scripts. The Reachin API has included functions to support the Reachin Display and stereo goggles. Figure 2.6 shows the Reachin Display, stereo goggles and spacemouse. Figure 2.5 shows two images from two of the Reachin demos.

[1] **V**irtual **R**eality **M**odeling **L**anguage



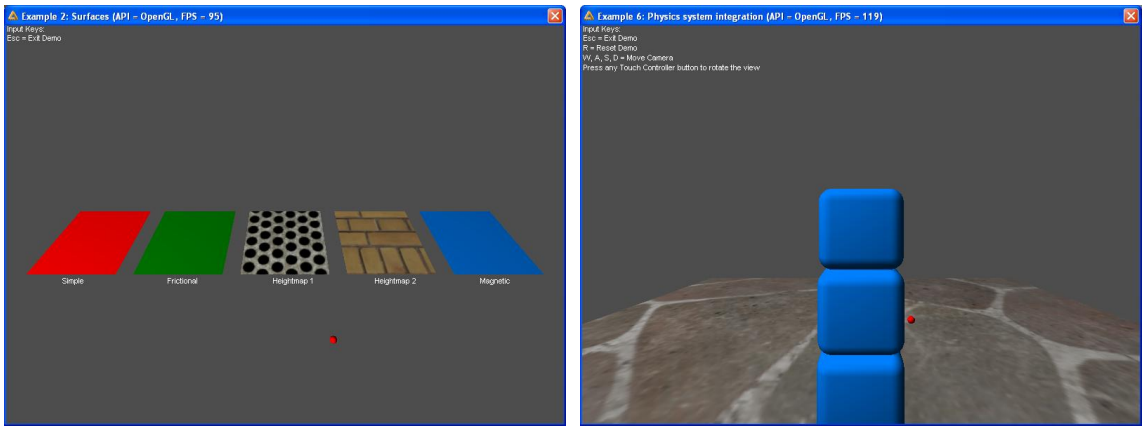

(a) Clay demo. (b) Volume demo.

Figure 2.5: Reachin API example programs. From [5]

Figure 2.6: Reachin Display. From [5]

### 2.2.4 HaptX

*HaptX*[1] is a new haptic scene graph made by Reachin. It was released in April 2007 and is written in C++. It does not have a graphical API included like the other scene graphs have. It supports both *OpenGL* and *DirectX* for 3D rendering, and it supports physics engines like *ODE*[2][22]. To program the scene it uses *C++*. It works with both SensAble Phantoms[16] and the Novint Falcon[2] haptic devices. Figure 2.7 shows two images from the HaptX example programs.




(a) Surface demo. (b) Physics demo.

Figure 2.7: HaptX example programs. From [1]

*HaptX* does not support stereo vision goggles like the Reachin API does. This can be added by using external APIs. How this can be added is described here *Writing stereoscopic software for OpenGL*[23], and a small *OpenGL* example can be found here OGLPlane[24] together with many more stereoscopic tools. There is also an alternative to use with *DirectX*, it is the StereoApi[25] from Nvidia.

To make HaptX work with a Reachin Display we need to use *DeviceSetCameraMatrix* method in the HaptX API to rotate the camera about 45 degrees. Making the scene visible in the mirror the right way. Figure 2.6 shows the Reachin Display with a Sensable Phantom Desktop, spacemouse and stereo goggles.

## 2.3 Anatomy

To make a medical simulator it is necessary to know a few things about the anatomy of the region in question. This project is going to make a simulator for syringe insertion into the shoulder of a human body. It is therefore good to know a little about what the shoulder looks like and what the simulator is going to hit.

[2] **O**pen **D**ynamics **E**ngine

In the shoulder region there are several bones that we need to know about. It is the *Scapula* the shoulder blade, the *Clavicle* the collar bone, the *Humerus* the bone in the upper arm and the ribbons. Figure 2.8 shows the bones in the shoulder region.

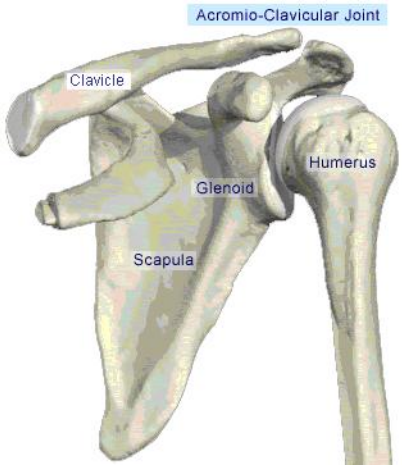



Figure 2.8: Bones in the shoulder. From [6]

There are a lot of muscles, veins, arteries, nerves, lymph and circulatory systems in the shoulder region. All these things should be defined in the model, but they are not that important. And has been left out in the test model made in this project. The main goal is to hit the membrane between the *humerus* and the *scapula*.

## 2.4 Properties of Soft Tissue

There is also a good idea to know a little about the properties of soft tissue. This section is based on the "Mechanics of soft tissue" chapter in "Cutting in deformable objects, The Thesis"[26].

All living tissue can be modeled by compressible hyperelastic aelotropic material models. This is a simplification for living tissue, it is not quite as simple as this. The mechanical characteristics of soft tissue are determined by connective tissue. The materials that contribute to the mechanics of the tissue includes the following:

- *Elastine* is a rubbery biological material. This material is almost perfectly elastic, and is found in the skin, artery walls and lungs.
- *Collagen* is a biological construction material. It forms the load bearing material in soft and hard tissue. It is a major component of tendons, bone, skin and blood vessels.

Elastine and Collagen put together can produce different fibers that is the building blocks of most parts of living cells.

The relationship between load and deformation (stress and strain) in soft tissue is divided into three trajectories:

- small load: the stress is exponential in the strain. (physiologic)
- medium load: the stress is linear in the strain. (overuse injury)
- large load: the tissue is almost stressed to failure, and reacts nonlinearly. (ligament rupture)

Figure 2.9 shows a graph for a ligaments stress and strain.

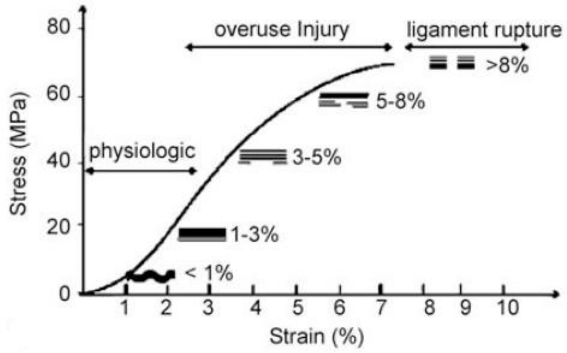



Figure 2.9: Tissue Stress and Strain relationship. From [7]

Normal tissue loads fall into the first category, small load. When you stretch soft tissue it offers more resistance than during a following unload. This phenomenon is called *hysteresis*, and it is an example of a viscoelastic effect: stresses in the material depend on the history of the deformation. When tissue is stressed with a constant load, then after the initial elastic response, the tissue will slowly distend further. This process is known as *creep*. A related phenomenon is *stress relaxation*: when a tissue specimen is loaded and then held at a constant elongation, stresses within the tissue decrease.

A description of 3D soft tissue elasticity is given by *quasi-linear visco-elasticity*. This model accounts for visco-elastic effects by modeling tissue as a superposition of materials with different relaxation times.

These properties need to be accounted for in the soft tissue in the simulator, to make it as realistic as possible.

# Chapter 3

# Proposed Solution

This chapter describes how it might be possible to make a medical needle insertion simulator.

## 3.1 Needle Insertion Simulator

The needle insertion simulator needs a way to deform the soft tissue, and a data model that can be deformed. This deformation needs to be realistic and fast.

The simulator is going to use a computer with Windows operating system, a haptic scene graph and a *SensAble Phantom*[16] haptic device for user interaction.

To start with, the simulator is going to simulate an insertion of a syringe in the shoulder region of the body. But it should be possible to load datasets from different parts of the body, and simulate on that dataset.

### 3.1.1 Soft Tissue Deformation

There are more than one way to deform the soft tissue of the model. It can be done with the use of a *mass spring system*, where every node in the model is connected with its neighbors with springs. *NURBS*[1], here the surface is modeled as splines, and the splines are deformed by changing the control points. *Bezier patches*, they are a lot like NURBS. Or by a *FEM*[2]. *FEM* is the most realistic approach, and is what has been chosen for this project. The mathematics of FEM is described in Chapter 3.2.

A *FEM* needs a data set with volume, and the best way to do this is by a *tetrahedral mesh*. Therefore it is needed to make a tetrahedral mesh from a MRI, a CT or a polygon dataset. The dataset needs to be segmented with the different tissue types defined.

Because of the size of the tetrahedral dataset and the complexity of the FEM calculations it

[1] Non-Uniform Rational B-Spline

[2] **F**inite **E**lement **M**ethod

is important to make sure that the mesh and the FEM method is optimized to be as fast and robust as possible. The mesh needs to be easy to divide into smaller tetrahedral elements when interacted with.

### 3.1.2 Tetrahedral mesh

The mesh in the *FEM* needs to be a mesh with volume, and the best solution for a mesh with volume is a *tetrahedral mesh*. There are other data structures with volume that could be used, like *voxels*. A tetrahedron is a data structure that consists of **four vertex**, **four faces** and **six edges**. As shown in Figure 3.1. By butting many tetrahedrons together to a mesh we get a model that has a volume and that it is easy to calculate forces on with a FEM.

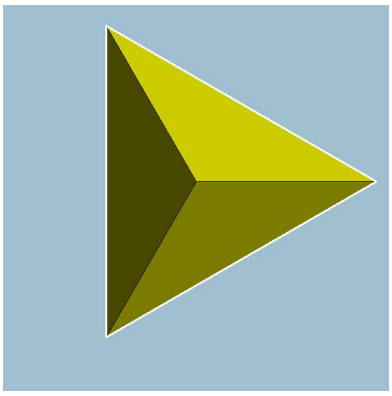


Figure 3.1: Figure of a Tetrahedron. From [8]

The simulation needs to be able to add points to the tetrahedral mesh and divide the mesh when the tip of the syringe hits the mesh. By doing so the start mesh does not need to be so fine, this saves memory and rendering time which makes the program go faster. The refinement of the mesh needs to be fast and effective so that it gets done in realtime.

### 3.1.3 Hybrid Tetrahedral Mesh

The mesh should be in a hybrid form. This means that the bones in the mesh does not need to be in a tetrahedral format. They only need to be in a polygon structure with variables on the nodes for the forces that effect the surrounding soft tissue. This can be done since there is no need to calculate and deform the bones. It is important that the vertexes in the tetrahedral mesh and the polygon model is the same. They can not be two separate models, since they need too interact. Figure 3.2 illustrates the hybrid tetrahedral mesh.

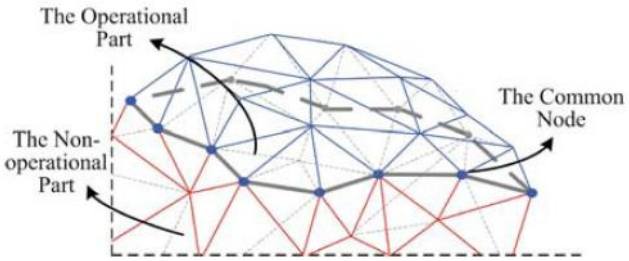



Figure 3.2: Hybrid Tetrahedral Mesh. From [7]

### 3.1.4 Tetrahedral Mesh and Haptic Scene Graphs

To load the tetrahedral mesh in a haptic scene graph we need to write a loader and make a data structure for the tetrahedral elements. This data structure might depend on the haptic scene graph used, since they usually have data structures for vertexes that could be used. It is possible to make a data structure from scratch that is independent of the haptic scene graph used.

The tetrahedrons need to know the volume they have. The volume should either be calculated fast in realtime or stored in the data structure. They also need to know their neighbors so that deformations can be calculated and distributed correctly.

#### Reachin API

The Reachin API[17] needs a data structure that is a subclass of the "Geometry" class of the Shape node and the tetrahedral elements need to be stored in a way that makes it easy to identify each of them, and splitting them when it is needed. They also need to store what kind of tissue the tetrahedron has.

#### HaptX

The newest scene graph from Reachin is the HaptX[1] scene graph. It is not made up around VRML like the Reachin API is, and does not have a class like the "Geometry" class of Reachin. The HaptX examples use win32 code and vertex arrays to display the scene. The examples that follows with the API use *ASE*[3] file format to load models. It might be a good idea to make the triangles that are going to be displayed be in a vertex array. By using vertex arrays more parts of the example code can be used. There are also a good idea to use vertex arrays because they are one of the fastest ways to render 3D graphics.

[3] ASCII Scene Exporter

### 3.1.5 Improvement

To improve the needle insertion, the program should be divided into two parts, one where the user uses a model of a finger to feel the contours of the surface. The user can only feel the surface without the possibility of penetrating it. This makes it possible to find the best point for the needle insertion. When this point is found, the finger should be switched with a syringe and the second part of the simulation can start. In the second part the needle insertion will be done at the selected point.

## 3.2 Finite Element Method

*Finite Element Method (FEM)* is a method used for finding an approximate solution of *partial differential equations* and of *integral equations*. For information about the different ways to implement *FEM* in medical simulators and their mathematics see the "Needle Insertion Simulator Applying A Haptic Device" [14] report. There a lot of different FEM versions are described and evaluated for use in a medical simulator.

The selected FEM for this simulator is the "Hybrid Condensed FEM"[15] & [7]. It divides the tetrahedral mesh into two parts, one that consists of the soft tissue and one that consists of the bones. The bones can not be deformed, and therefore does not need to be in a tetrahedral mesh format and the forces on the rest of the nodes in the mesh can be precomputed.

There are two ways to simulate the deformations in the soft tissue with *FEM*. By using *Linear* or *Non-Linear* formulas. The tissue itself is non-linear, but it is harder and requires more computations to emulate. The linear equations are less computational expensive and they still work quite well.

### 3.2.1 Basic

The basic formulas of a finite element analysis is:

$$Md'' + Dd' + Kd = f \tag{3.1}$$

Where $M$ is the mass matrix, $D$ is the dampening matrix, $K$ is the stiffness matrix, $d$ is the nodal displacement and $f$ is the force applied. The matrices is of size $n \times n$, where $n$ is the number of nodes in the model times the degrees of freedom. In a three dimensional space $n$ is three.

#### LU Decomposition

A way to reduce the computations needed is to use *LU Decomposition*. Here the FE Equation 3.1 is reduced by ignoring dynamic effects. Giving us this formula:

$$Kd = f \tag{3.2}$$

$K$ is the $n \times n$ stiffness matrix, $d$ and $f$ are the $n \times 1$ vectors of nodal displacement and applied force. To break down a 100% dense matrix[4] to L and U components $2/3n^3$ calculations are required for a $n \times n$ matrix.

#### Inverse of the Stiffness Matrix

Another way is to use the *Inverse of the Stiffness matrix*. The fact that only a small number of points might need to handle contact and force feedback, is exploited in this method. By using these fact the use of a precomputed inverse of the stiffness matrix at runtime is more effective than LU decomposition. This is only the case where the number of points is small enough. The inverse of the stiffness matrix can be computed in $2\alpha n^3$ operations, where $n$ is three times the number of nodes and $\alpha$ refers to the efficiency gains from sparse LU decomposition techniques. Under follows the equations for the *Inverse of the Stiffness matrix*.

$$\begin{pmatrix} d_{nocontact} \\ d_{contact} \end{pmatrix} = \begin{pmatrix} K_{i_{aa}} & K_{i_{ab}} \\ K_{i_{ab}}^T & K_{i_{bb}} \end{pmatrix} \begin{pmatrix} f_{nocontact} \\ f_{contact} \end{pmatrix} \tag{3.3}$$

In Equation 3.3 the known variables are $d_{contact}$, $f_{nocontact}$ and $K_i$ is the inverse of $K$. The size of the submatrix in three dimensions $b$ is three times the number of contact nodes and $a = n - b$. To obtain the unknowns at runtime we get the following formulas:

$$f_{contact} = (K_{i_{bb}}^{-1})(d_{contact} - K_{i_{ab}}^T f_{nocontact}) \tag{3.4}$$

$$d_{nocontact} = K_{i_{aa}}(f_{nocontact} + K_{i_{ab}} f_{contact}) \tag{3.5}$$

In these equations a sparse $f$ vector can significantly reduce computation.

It is possible to save a lot of matrix calculation by only calculating the *none zero* values in the matrix. This will speed up a large matrix multiplication quite a lot.

### 3.2.2 Conjugate Gradient FEM

One of the FEM versions that has been tested for needle insertion is the *Conjugate Gradient FEM* described in "Supporting cuts and finite element deformation in interactive surgery simulation"[27].

It uses tetrahedral elements and the linear elastic material method. It is based on the sparse property of the stiffness matrix. The complexity of the elevation function is almost linear in respect to the total number of nodes in the mesh. The performance of this method is subject to the number of iterations required for convergence. This method together with the structure of the tetrahedral mesh is made to make it easy to cut and make changes to the tetrahedral mesh. This means that the mesh does not need to have as high resolution as it would if it was

[4] dense matrix is a matrix with no zero values

impossible to divide the tetrahedrons into smaller tetrahedrons. For a needle insertion program to work without the possibility to divide the tetrahedrons the size of the tetrahedrons would have to be limited to the size of the needle. This means that the mesh would be to big, and it would be impossible to simulate anything realtime on it.

The *Conjugate Gradient FEM* has been tested for use in needle insertion in "Interactive needle insertions in 3D nonlinear material"[28]. The needle insertion program is made for Red Hat Linux. The program, images and a movie can be found here http://www.cs.uu.nl/groups/AA/virtual/surgery/needle3d/. Figure 3.3 shows two images from this test. A lot of the things implemented in this project also need to be implemented into a needle insertion simulator.

The mathematics for this method is described in [27], [28] and [14]. The basic idea is that the total deformation is determined by balancing all external forces with all elastic forces, or equivalently, when the energy of the system is minimal. This is described as in Equation 3.2:

$$f_{external} = -K \cdot d \tag{3.6}$$

Where $f_{external}$ is the total global external force on the tissue. Vector $d$ is the displacement of the tissue. Both of these are vectors of dimension $3n$, where $n$ is the number of nodes in the mesh. $K$ is the big global stiffness matrix of $3n \times 3n$. Every part of the vector $K \cdot d$ is an elastic force that can be computed locally. Therefore we do not need the matrix $K$ itself. This means that the memory requirement is kept low thanks to the sparseness of $K$.

### 3.2.3 Hybrid Condensed FEM

The *Hybrid Condensed FEM* uses the *Hybrid Tetrahedral mesh* described in Chapter 3.1.3. This method has good potential for use in a needle insertion simulator, and is based on the *Conjugate Gradient* method.

Here are the main aspects of the mathematics in the *Hybrid Condensed FEM*. The formulas for linear elasticity is as follows:

The second order tensor is:

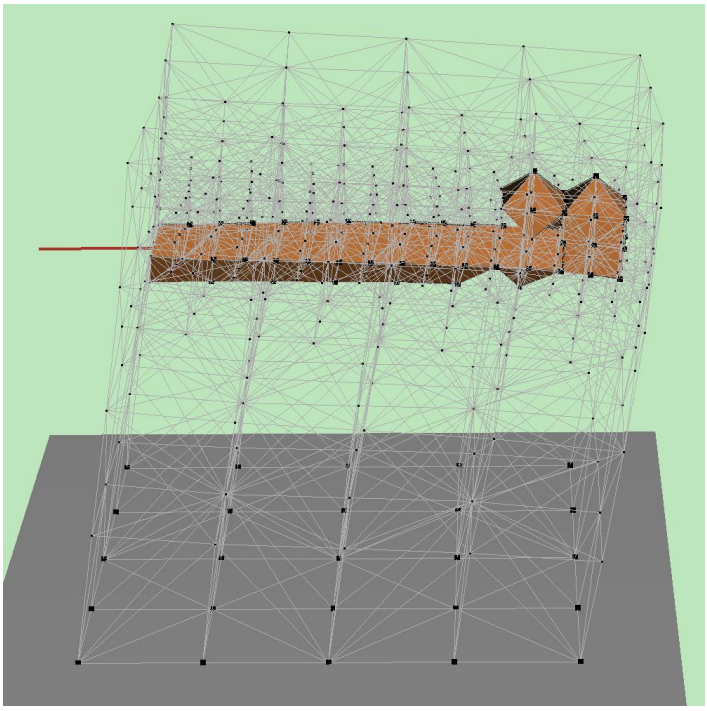
$$E = \frac{1}{2}(F^T \cdot F - I) = \frac{1}{2}(C - I) \tag{3.7}$$

Here $E$ is the *Green-Lagrange strain tensor*, $F$ is the deformation gradient and $C$ is the *Cauchy-Green deformation tensor*. In the Cartesian coordinate system the *Green-Lagrange strain tensor* components can be represented as:

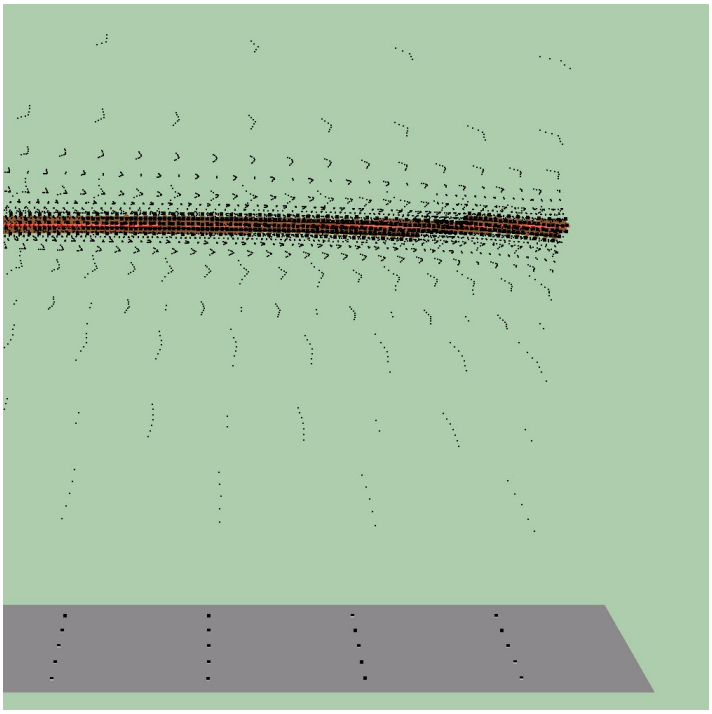
$$E_{xx} = \frac{\delta u}{\delta x} + \frac{1}{2}\left[\left(\frac{\delta u}{\delta x}\right)^2 + \left(\frac{\delta v}{\delta x}\right)^2 + \left(\frac{\delta w}{\delta x}\right)^2\right], \tag{3.8}$$

$$E_{xy} = \frac{1}{2}\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta y}\right) + \frac{1}{2}\left[\frac{\delta u}{\delta x}\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x}\frac{\delta v}{\delta y} + \frac{\delta w}{\delta x}\frac{\delta w}{\delta y}\right]. \tag{3.9}$$

The other four strain tensor components are represented similarly. When we are dealing with small deformations and rotations the second order portion can be neglected. From this we get



(a) linear material (refinement level 12)



(b) nonlinear material (neohookean material, refinement level 20)

Figure 3.3: Needle 3D Insertion, needle radius 1 mm. By Han-Wen Nienhuys.

the *Green-Langrange strain tensor* approximately equal to the strain $\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{yz}, \varepsilon_{zx}, \varepsilon_{xy}$ in a linear FE analysis.

$$\varepsilon_{xx} = \frac{\delta u}{\delta x},\ \varepsilon_{yz} = \frac{1}{2}\left(\frac{\delta v}{\delta z} + \frac{\delta w}{\delta y}\right),\ \varepsilon_{yy} = \frac{\delta v}{\delta y},\ \varepsilon_{zx} = \frac{1}{2}\left(\frac{\delta w}{\delta x} + \frac{\delta u}{\delta z}\right),\ \varepsilon_{zz} = \frac{\delta w}{\delta z},\ \varepsilon_{xy} = \frac{1}{2}\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x}\right). \tag{3.10}$$

The strain vector $\varepsilon$ is represented as a 6-component vector:

$$\varepsilon^T = [\varepsilon_{xx}\ \varepsilon_{yy}\ \varepsilon_{zz}\ \varepsilon_{yz}\ \varepsilon_{zx}\ \varepsilon_{xy}] \tag{3.11}$$

In the *tetrahedral element* the displacement variation $\mathbf{u} = (u,\ v,\ w)$ can be given as follows:

$$\mathbf{u} = N^e \cdot \mathbf{u}^e = \left[\mathbf{I} \cdot N_i\ \mathbf{I} \cdot N_j\ \mathbf{I} \cdot N_m\ \mathbf{I} \cdot N_p\right]\left[\mathbf{u}_i^e\ \mathbf{u}_j^e\ \mathbf{u}_m^e\ \mathbf{u}_p^e\right]^T \tag{3.12}$$

Where $\mathbf{I}$ is a $3 \times 3$ identity matrix and the shape function $N$ is a linear shape function and defined as:

$$N_r = \frac{a_r + b_r X + c_r y + d_r z}{6V},\ r = i,\ j,\ m,\ p \tag{3.13}$$

where

$$a_{i=1} = a_1 = (-1)^{i-1}\begin{pmatrix} x_j & y_j & z_j \\ x_m & y_m & z_m \\ x_p & y_p & z_p \end{pmatrix},\ b_{i=1} = b_1 = (-1)^{i-1}\begin{pmatrix} 1 & y_j & z_j \\ 1 & y_m & z_m \\ 1 & y_p & z_p \end{pmatrix},$$
$$c_{i=1} = c_1 = (-1)^{i-1}\begin{pmatrix} x_j & 1 & z_j \\ x_m & 1 & z_m \\ x_p & 1 & z_p \end{pmatrix},\ d_{i=1} = d_1 = (-1)^{i-1}\begin{pmatrix} x_j & y_j & 1 \\ x_m & y_m & 1 \\ x_p & y_p & 1 \end{pmatrix},$$

$V$ is the volume of the tetrahedral element.

The strain $\varepsilon$ at an arbitrary point in the element can be expressed as:

$$\varepsilon = \mathbf{BU} \tag{3.14}$$

Where $\mathbf{B}$ is the displacement differentiation matrix, $6 \times 3_{n_e}$ matrix and $\mathbf{U}$ is a $3_{n_e} \times 1$ vector of nodal displacement. $n_e$ is the number of nodes in an element.

With a linear elastic material the stress $\sigma$ is linearly related to the strain $\varepsilon$ via *Young's Modulus*, and we can obtain the strain energy $E$ with:

$$E = \frac{1}{2}\int_V \sigma^T \varepsilon dV \tag{3.15}$$
$$= \frac{1}{2}\int_V epsilon^T \mathbf{D}\varepsilon dV \tag{3.16}$$
$$= \frac{1}{2}\int_V (\mathbf{BU})^T (\mathbf{BU}) dV, \tag{3.17}$$

Where $\mathbf{D}$ is the symmetric elastic matrix that is related to the material physical properties. The potential energy of a body is the sum of the total strain energy $E$. The work $W$ performed on the body by external forces can be written as:

$$W = \int_V \mathbf{u} \cdot \mathbf{f}_b dV + \int_\Gamma \mathbf{u} \cdot \mathbf{f}_s dS + \sum_i \mathbf{u}_i \cdot \mathbf{p}_i \tag{3.18}$$

In this equation $\mathbf{u}$ is the displacement vector, $\mathbf{f}_b$ is the body forces applied to the volume $dV$, $\mathbf{f}_s$ is the surface forces applied to the surface, $dS$ and $\mathbf{p}_i$ is the concentrated load acting at point $(x_i,\ y_i,\ z_i)$. From this a set of linear equations can be derived: $\mathbf{K}^e\mathbf{u}^e = \mathbf{f}^e$, where $\mathbf{K}^e$ is the stiffness matrix of a single element. These elements are put together into a global stiffness matrix that determines the global elastic properties of the model.

### Nonoperational Region

The processing of the nonoperational region can be done by using these formulas. The equations for the operational and nonoperational region:

$$\begin{pmatrix} \mathbf{K}_{11} & \mathbf{K}_{1I} \\ \mathbf{K}_{I1} & \mathbf{K}_{II} \end{pmatrix} \begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_I \end{pmatrix} = \begin{pmatrix} \mathbf{P}_1 \\ -\mathbf{P}_I \end{pmatrix}, \tag{3.19}$$

$$\begin{pmatrix} \mathbf{K}^2_{II} & \mathbf{K}_{I2} \\ \mathbf{K}_{2I} & \mathbf{K}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_I \\ \mathbf{P}_2 \end{pmatrix}, \tag{3.20}$$

Where superscript 1 represent the operational and 2 represent the nonoperational regions, subscript $I$ represents the common nodes shared with these two regions. $\mathbf{P}_I$ and $-\mathbf{P}_I$ are the force and counterforce applied to the common nodes when we analyze these two parts separately. From this equation we can obtain a new matrix system:

$$\begin{pmatrix} \mathbf{K}_{11} & \mathbf{K}_{1I} \\ \mathbf{K}_{I1} & \mathbf{K}^1_{II} + K' \end{pmatrix} \begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_I \end{pmatrix} = \begin{pmatrix} \mathbf{P}_1 \\ -P'_I \end{pmatrix} \tag{3.21}$$

where

$$K' = \mathbf{K}^2_{II} - \mathbf{K}_{I2} \cdot \mathbf{K}^{-1}_{22} \cdot \mathbf{K}_{2I}, \tag{3.22}$$

$$P' = \mathbf{K}_{I2} \cdot \mathbf{K}^{-1}_{22} \cdot \mathbf{P}_2 \tag{3.23}$$

Which we can rewrite into the condensed from:

$$\begin{pmatrix} \mathbf{K}^2_{II} & \mathbf{K}_{Is} & \mathbf{K}_{Ii} \\ \mathbf{K}_{sI} & \mathbf{K}_{ss} & \mathbf{K}_{si} \\ \mathbf{K}_{iI} & \mathbf{K}_{is} & \mathbf{K}_{ii} \end{pmatrix} \begin{pmatrix} \mathbf{A}_I \\ \mathbf{A}_s \\ \mathbf{A}_i \end{pmatrix} = \begin{pmatrix} \mathbf{P}_I \\ \mathbf{P}_s \\ \mathbf{P}_i \end{pmatrix} \tag{3.24}$$

Subscript $i$ represents the interior nodes that can be condensed out and $s$ represents the surface nodes to be retained. From this we then get:

$$(\mathbf{K}_{ss} - \mathbf{K}_{si} \cdot \mathbf{K}^{-1}_{ii} \cdot \mathbf{K}_{is}) \cdot \mathbf{A}_s = \mathbf{P}_s - \mathbf{K}_{si} \cdot \mathbf{K}^{-1}_{ii} \cdot \mathbf{P}_i + (\mathbf{K}_{si} \cdot \mathbf{K}^{-1}_{ii} \cdot \mathbf{K}_{iI} - \mathbf{K}_{sI})\mathbf{A}_I \tag{3.25}$$

Since the external forces do not apply to the interior nodes all terms related to $\mathbf{P}_i$ can be ignored. Equation 3.25 can be written as: $K^* \cdot \mathbf{A}_s = \mathbf{P}_s + P^* - \mathbf{A}_I$ where

$$K^* = \mathbf{K}_{ss} - \mathbf{K}_{si} \cdot \mathbf{K}^{-1}_{ii} \cdot \mathbf{K}_{is}, \tag{3.26}$$

$$P^* = \mathbf{K}_{si} \cdot \mathbf{K}^{-1}_{ii} \cdot \mathbf{K}_{iI} - \mathbf{K}_{sI} \tag{3.27}$$

The following variable remains unchanged during the simulation: $K', P', K^*$ and $P^*$, and are therefore calculated in the preprocessing stage. To remove this restriction, a new procedure has

to be made for the preprocessing data. Since $\mathbf{P}_2$ in Equation 3.19 and $\mathbf{P}_s$ in Equation 3.24 no longer are zero when users interact with the tissue model, they need to be recalculated.

The interactive force vector $\mathbf{P}'_2$ represent the nonoperational region produced by user interaction, and is changed on the fly. We rewrite Equation 3.23 to $P' = \mathbf{K}_{I2} \cdot \mathbf{K}_{22}^{-1} \cdot \mathbf{P}'_2$, from Equations 3.25 to 3.27 we get:

$$\mathbf{A}_s = K^{*-1} \cdot \mathbf{P}_s + K^{*-1} \cdot P^* \cdot \mathbf{A}_I \tag{3.28}$$

Since the interaction takes place only on the surface, the nodes on which interactive forces act should be among the surface nodes of the nonoperational region $\mathbf{P}_s$. The Equation 3.28 can be rewritten to:

$$\mathbf{A}_s = K^{*-1} \cdot \mathbf{P}_{s0} + K^{*-1} \cdot \mathbf{P}_{s\_interact} + K^{*-1} \cdot P^* \cdot \mathbf{A}_I \tag{3.29}$$

Where $\mathbf{P}_{s0}$ is the external force vector of the initial condition when the simulation starts and $\mathbf{P}_{s\_interact}$ is the vector of applied force that is updated during the simulation.

### Preprocessing

Here is a list of the things that needs to be precomputed for the *Hybrid Condensed FEM*.

Table 3.1: Hybrid Condensed FEM preprocessing variables

| | |
|---|---|
| Preprocessing data | $K'$ |
| | $\mathbf{K}_{I2} \cdot \mathbf{K}_{22}^{-1}$ |
| | $K^{*-1}$ |
| | $K^{*-1} \cdot \mathbf{P}_{s0}$ |
| | $K^{*-1} \cdot P^*$ |

These variables can then be stored and does not need to be changed during the simulation.

### Run Time Calculations

The *Hybrid Condensed* method has the following computational stages:

1. Calculate the corresponding preprocessing data listed in Table 3.1
2. Use Equation 3.21 to compute displacement of nodes $\mathbf{A}_1$ and $\mathbf{A}_I$ in the operational region.
3. Determine the interactive nodes of the nonoperational region and update the applied force vector $\mathbf{P}_{s\_interact}$.
4. Compute the new values of $K^{*-1} \cdot \mathbf{P}_{s\_interact}$
5. Use Equation 3.29 to compute the displacement of the surface nodes $\mathbf{A}_s$ in the nonoperational region.

By using the fact that $\mathbf{P}_{s\_interact}$ is made up of a lot of zeros and only calculating the values of the matrix multiplication, where there are nonzero entries, the update can be done easily and a lot faster.

**Improvement**

To improve the speed of the **Hybrid Condensed** method the *GPU's*[5] *fragment shader* has been taken into use in the "An improved scheme of an interactive finite element model for 3D soft-tissue cutting and deformation"[7] article. When there are made changes to the tetrahedral mesh, the mesh and the coefficient matrix is reconstructed. Therefore the new equations for the mesh needs to be recomputed. For these elements the *Conjugate Gradient* method is used. The matrix vector multiplication in the *Conjugate Gradient* algorithm requires $O(m)$ operations, where $m$ is the number of nonzero entries in the matrix. Therefore a faster matrix vector multiplication implies a faster computation of the mesh, and a faster simulation. By using the *fragment processor* of the GPU, and its efficient manipulation of the local texture memory, it is possible to speed up the calculations. To make use of the GPU the matrices and vectors are loaded as textures into the GPU and then rasterized to a proper quad of pixels for invoking the fragment program. The result from the GPU can be obtained as a color value and transferred back to the CPU.

An alternative to use the GPU is to use a *PPU*[6], like **PhysX** from Ageia [29]. PPU's are specially designed to calculate physics fast, like explosions, cloth behavior, water, smoke and more. This might be a good way to speed up the matrix calculations.

There is also a third alternative and that is to take advantage of the multicore CPU that is coming these days. The new CPU's are equipped with four or more cores, making it possible to have two or more cores working on the calculations of the matrix multiplications in parallel.

There might also be possible to use all tree of these at the same time. Parallelizing the different cores of the CPU, the GPU and a PPU could speed up the program heavily. This would make it easier to get a more realistic simulation.

### 3.2.4 Tetrahedral Volume

The volume of a *tetrahedron* needs to be calculated for the *FEM*. This can be done by using cross and dot products. Given a tetrahedron like the one in Figure 3.4 with the vertexes *a, b, c, d* the formula for the volume is:

[5] **G**raphics **P**rocessing **U**nit
[6] **P**hysics **P**rocessing **U**nit

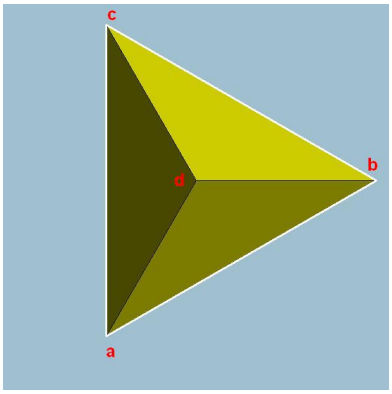



Figure 3.4: Tetrahedron with vertexes marked.

$$\mathbf{V} = \frac{|(d-a)\cdot((d-b)\times(d-c))|}{6} \tag{3.30}$$

The volume should be stored in the tetrahedron so that there is no need to recalculate this each time the volume is needed. There is only one more value to be stored in the data structure, and that should not be a problem.

## 3.3 Mesh generation

The mesh is going to be made from a polygon model. The polygon model may be made from a CT or MRI scanned volume that has been segmented. The mesh need to be of a tetrahedral structure and needs to be divided into a part that can be deformed with the use of a FEM (soft tissue), and a part that can't be deformed (bones).

### 3.3.1 TetGen

The mesh generation will use algorithms based on "Delaunay triangulation"[30]. *TetGen* [31] is an open source program for Delaunay triangulation of a polygon mesh to a tetrahedral mesh. TetGen is written in ANSI C++ and runs on Linux, Unix, Windows and Mac. It uses the command line for file input and input switches. It can do a lot of the things that is needed in the tetrahedral mesh for the simulator, but not all. It can not divide the input mesh into two parts, an operational and a nonoperational as would be the best solution for the mesh. But it can divide the mesh up into different regions. The tetrahedral meshes generated by TetGen can be viewed using *TetView*. Figure 3.5 is an image of a tetrahedral mesh generated by TetGen, viewed in TetView.

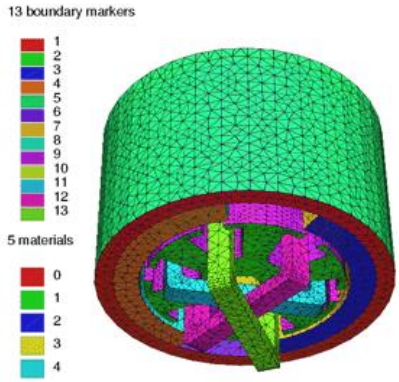



Figure 3.5: TetView image of tetrahedral mesh from TetGen. From [9]

### 3.3.2 Tetrahedral Mesh Generation

The *FEM* of the proposed needle insertion simulation needs a tetrahedral mesh that has the different tissue types defined. *TetGen* is an open source program that can make a tetrahedral mesh with regions marked, but it can not make regions from other formates than *.poly*[32] and *.smesh*, which both are *piecewise linear complex* formats.

*TetGen* can be edited to load other formates and to load regions from some of the other formates. In the end tetrahedral mesh the bones does not need to be in tetrahedral format, they could be represented as polygons. This would save a lot of memory and make the program faster. The regions are only defined by an integer, so it is needed to specify numbers as different tissue types.

Therefore the proposed solution is to edit TetGen's code so that it makes a tetrahedral mesh from for example "stl"[33] files with regions. The ideal solution would be to make a program that made and segmented a MR Image[7][12] directly into a Tetrahedral data structure.

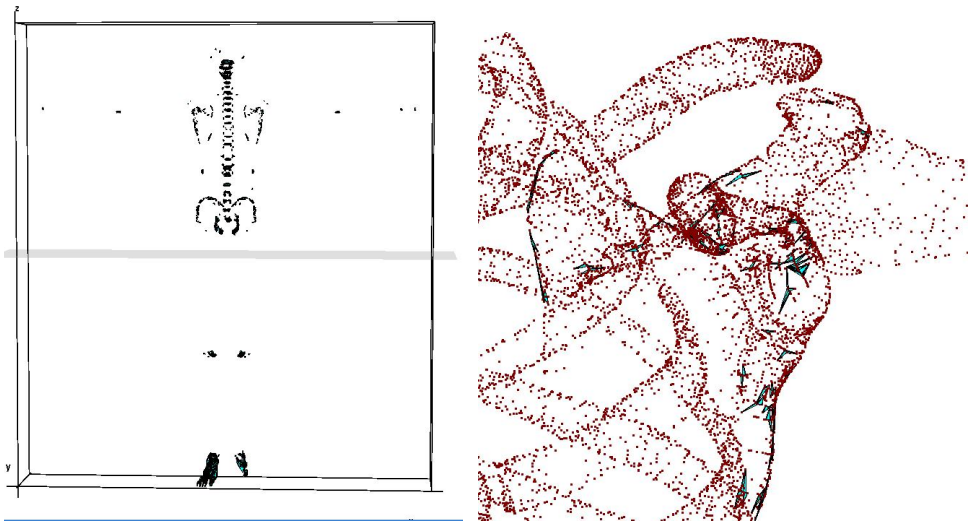
### 3.3.3 Triangle Intersection

*TetGen* does not make a mesh from a model with triangle intersections. Therefore we need to find a way to remove intersecting triangles from the model, since TetGen only finds the intersections and then report them without making a tetrahedral mesh.

When TetGen runs with the "-d" switch it only outputs a ".node", a ". face" and a ".smesh" file with the faces that are intersecting. It is necessary to find a way to fix the triangle intersection so that the mesh is created like it should, because it only becomes a mess when triangles intersect. Figure 3.6 shows the triangle intersections in the bones in the man polygon model.

The possibility to use *3D Studio Max*[34] or *Blender*[35] to clean up the mesh before TetGen

[7] **M**agnetic **R**esonance **I**mage



(a) Intersections. (b) Shoulder area with vertexes.

Figure 3.6: Polygon Intersection in the bones in the man polygon model.

gets a hold of it has been considered, but it does not seem like any of these 3D editors have the functions necessary to do this. They make polygon meshes that are full of errors. In most cases where the models from these programs are used it does not mater as long as it looks right. It might be possible to make a script to *3D Studio Max* that can fix the triangle intersections, but then it might just be an equally good idea to write the code anywhere else.

### 3.3.4 Open Faces

*TetGen* does not make a mesh from models which have open faces. An open face means that there is a hole in the model. The volume is not closed, resulting in a leak when the volume is trying to be created. When there is a hole in the model there is not a complete volume there, it is like trying to fill a bucket with a hole in the bottom with water, it just flows out. That is what happens when the algorithm comes across a model with one or more open faces in it. So there are no volumes to be made and it only reports the coordinates of the first open face it finds in that object and moves on without making a mesh of that object's volume. If the object is inside another object, that space will be filled with the other object's mesh, because the other object flows into that space.

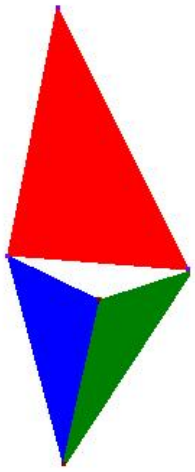


Figure 3.7: Open Face illustration.

The best solution to this problem is to make a program that finds the open faces, and then reconnects the polygons surrounding the hole. Because in most, if not all, cases the hole is there because the polygons are connected wrong. A polygon is connected to the wrong vertex making a small hole in the mesh. Figure 3.7 shows an illustration that has been made to show an open face error in a polygon mesh. The red triangle should have been divided into two but it is not, making a small hole in the mesh. These holes is usually so small that they are very hard to see with the naked eye.

### 3.3.5 Tissue Types

Since the *tetrahedral mesh* in the *.ele* file format only has an *integer* variable for material type, it is needed to define values that says what kind of tissue it is. This can be done by defining integer values for the different tissue types. This would make the simulator know that a tetrahedron, which for instance has a material value of 100 is of the type skin. By doing this the simulator automatically sets the right tissue type, colors, textures and forces to the model when it is loaded. The different tissue types are defined in Table 3.2.

It might be a good idea to define all the bones and muscles that could be in the simulator separately, making it possible to give the user output on what bone or muscle that has been hit. This would improve the users experience when using the simulator, and might be very useful for them.

| **Value** | **Tissue Type** |
|---|---|
| 0 - 99 | Not Defined |
| 100 | Skin |
| 101 | Soft tissue |
| 102 | Blood veins |
| 103 | Blood arteries |
| 104 | Nerves |
| 105 | Lung |
| 106 | Hart |
| 107 | Membrane |
| 200 - 299 | Bones |
| 300 - 399 | Muscles |

Table 3.2: Tissue Type Definement

# Chapter 4

# Implementation

This chapter describes what has been done to create a *tetrahedral mesh* and a *needle insertion simulator*.

## 4.1 TetGen

In this section a description of what has been attempted to accomplished with *TetGen*'s[31] code to create a *tetrahedral mesh* for a needle insertion simulator is described. It is TetGen version 1.4.1 that has been edited here. TetGen has been released in version 1.4.2 after it was edited for use in this project. It should be no problem to implement these changes into the new version of the program.

### 4.1.1 STL With Regions

The *load_stl* method in the *tetgenio* class has been edited to load region information from a text file. The added code to the *load_stl* method can be found in Chapter 9.1. The text file is named *.regions* and holds information about regions in the *stl* file in the same format as they are defined in the " piecewise linear complex"(.poly)[32] files. The program finds the file automatically if it is named correctly, if it is not it will assume that there is no regions file and create a mesh without region information. In the file the first number is the number of regions, the next lines consist of the *index of the region*, the *x, y and z* coordinates which is a point inside the region, a *region identifier* and a *region attribute*.

This solution has been chosen because it is a lot simpler to implement than to write an algorithm to find a point automatically inside all the *solid* blocks of the *stl* file. This also makes it possible to have the whole model as just one solid in the stl file. The drawback is that a coordinate from the different regions need to be found manually in a 3d editor like *3D studio Max*[34] or *Blender*[35]. The code for adding the regions is almost identical to the region code in the *load_poly* method.

Figure 4.1 shows the tetrahedral mesh result from a stl test file with three regions defined. The model consists of a cube with a cylinder and a sphere inside it. The cube is in red with the region identifier 100, a cylinder is in green with the identifier 200 and a sphere is in blue with the identifier 201. A very simple model made only to test the improvements done to *TetGen*. The model is made this simple to make sure that it does not have any intersecting triangles.

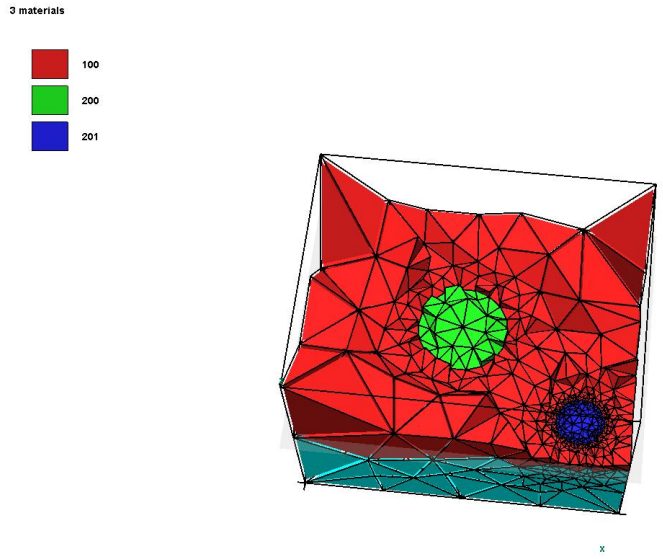


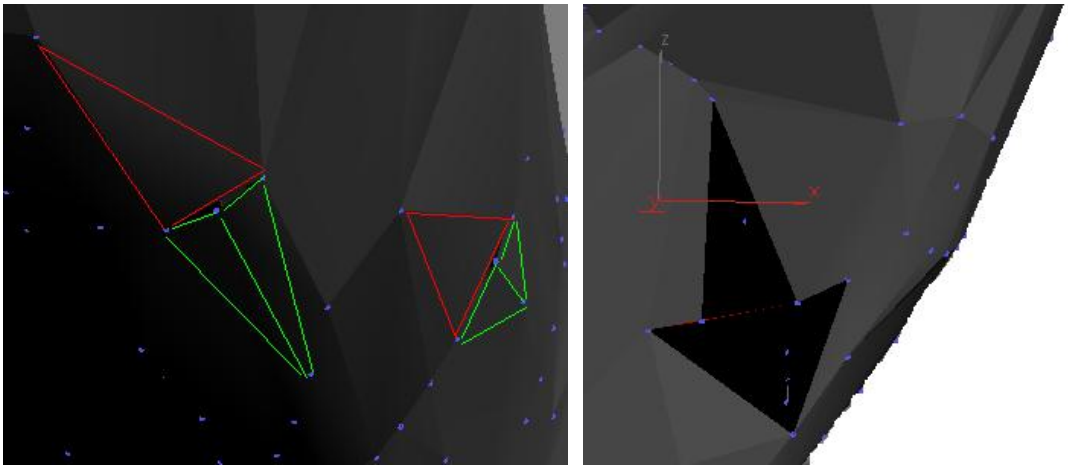

Figure 4.1: Tetrahedral mesh form a stl test file.

### 4.1.2 Triangle Intersection & Open Faces

Files with triangle intersections and open faces need to be fixed before a tetrahedral mesh can be made. A couple of hacks in *TetGen*'s code has been attempted to come around the problem of *triangle intersection*. It has been attempted to let TetGen ignore the fault in the mesh, this only results in a forever loop somewhere in the *tetrahedralize* method. Deleting the triangles that intersects has also been tested, but this only produces *open faces*, which does not help since it can not create a mesh from a model that has open faces. These errors makes TetGen exits without creating a mesh from that region.

Have not tried to make *TetGen* insert faces where it detects an open face. It could be done, but it is not as easy as just adding a face to the mesh, because in most cases the reason that it is an open face is that the mesh is not connected correctly. The hole is very small and most of the time almost invisible to the eye, and the problem is there because a vertex has been skipped when the polygons has been made. To fix this at least one polygon needs to be deleted. It is hard and time consuming to write an algorithm to do this correctly. This has therefore not been done.

What have been done is to edit the triangles of the files manually in *3D Studio Max*. This way manually finding and removing the errors. This is a very hard and time consuming task since the errors are small and very hard to see. In Figure 4.2 there are two images of the most common errors in the polygon model that results in *triangle intersections* and *open faces*.



(a) Open Face. Two wrong polygons

(b) Triangle Intersection. Wrong polygons that make an intersection, that is hard to find.

Figure 4.2: Triangle errors in the Scapula polygon model.

The figure is from the left scapula of the shoulder model that is described in Chapter 4.2.2. It shows an image of a part with two wrongly connected triangles making two open faces. The wrong triangles are marked in red, and an image of a triangle intersection where the intersecting triangle is marked in red. The intersecting triangle is very small and hard to see. To find it the surrounding triangles had to be deleted.

## 4.2 Tetrahedral Mesh

There has been acquired a *MRI* and a *CT* data set of the shoulder. From the MRI data set the skin and the bones has been segmented out and stored in the *.stl* file format. This has been done because the scan is from one of the specimens that has been used in the existing simulator. These files are quite large and they are not completely free of errors, there are triangle intersections in the files. Because of the size of these files and the format they are stored in *3D studio Max* and *Blender* can not effectively open these files. It is too hard and time consuming to connect all the different vertexes and faces. Since it is so hard to open the files, it is almost impossible to edit the files and remove the triangle intersections. And besides, there is only the skin and the bones that has been segmented out. Because of this the files can not be used in this project. Figure 4.3 shows the *MRI* volume and Figure 4.4 shows the *.stl* segmented bones and skin.

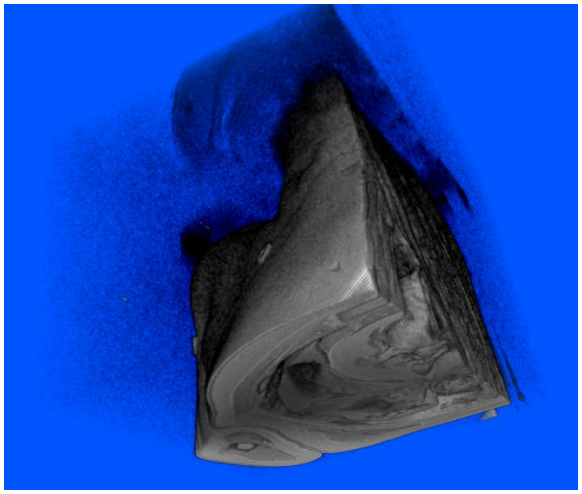
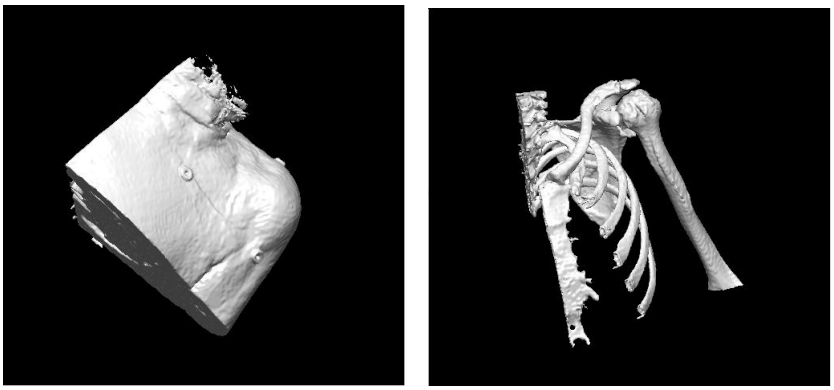


Figure 4.3: MRI Shoulder volume

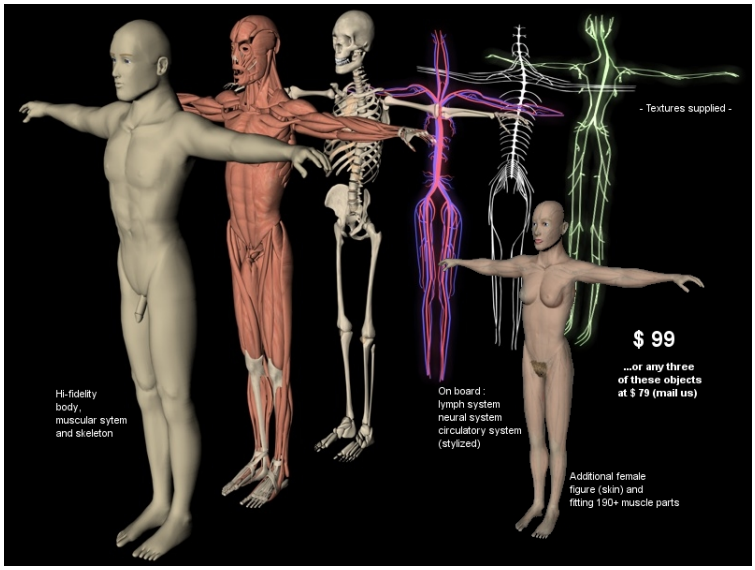


(a) Skin. (b) Bones.

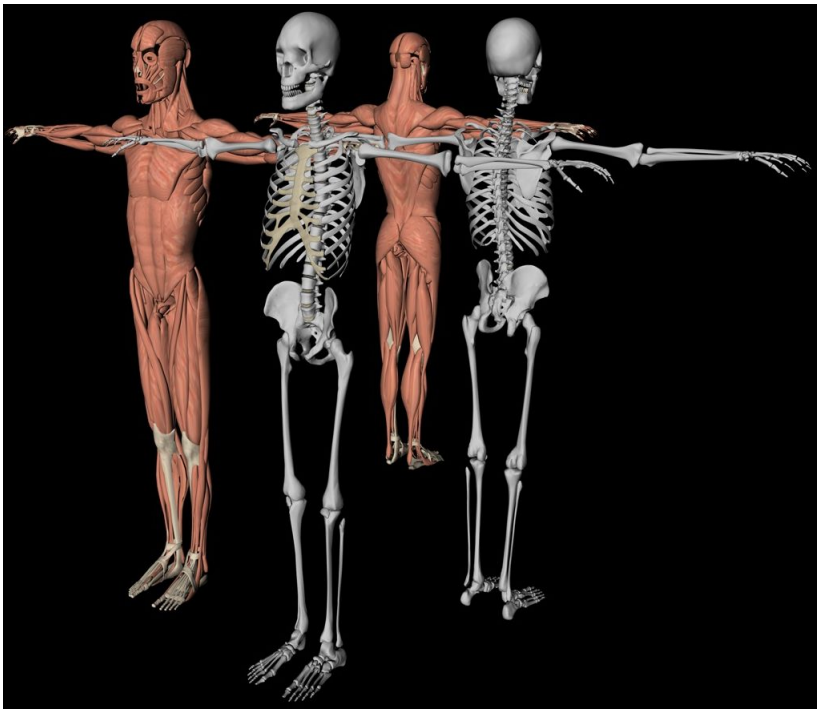
Figure 4.4: Stl Shoulder skin and bones.

There has been bought a polygon model[10] of a human body with skin both for a man and a woman. The model consists of *bones, muscles, nerves, lymph and circulatory system*. There are 219 parts in the muscle system, brain and eyes has 56 parts, the skeleton has 114 parts. The nerve, lymph and cardiovascular system consists of 1 to 3 parts each. The model has textures for all the parts making it look quite good. Figure 4.5 shows the model and its parts.

The polygon model has been edited in *3D Studio Max*[34] to include only the part of the body that is needed for this simulation. The objects in the model has been exported to the *.stl* file format so that it can be loaded into *TetGen* and made a *tetrahedral mesh* from it. The objects have then been put together in a *.stl* file to make a complete model of the region. *3D Studio Max* has then been used to get out a set of coordinates from the different regions. These coordinates have been stored in a *.regions* file. The *.stl* and the *.regions* file have been loaded into the edited version of *TetGen*, described in Chapter 4.1. *TetGen* has then been used to make a tetrahedral mesh with regions from the model. The program has been run with the *-d* switch to make sure that the model is without errors. When all the errors in the model have been fixed the program has been run with the *-qA* switch to make a *tetrahedral mesh* with regions.




(a) All parts.

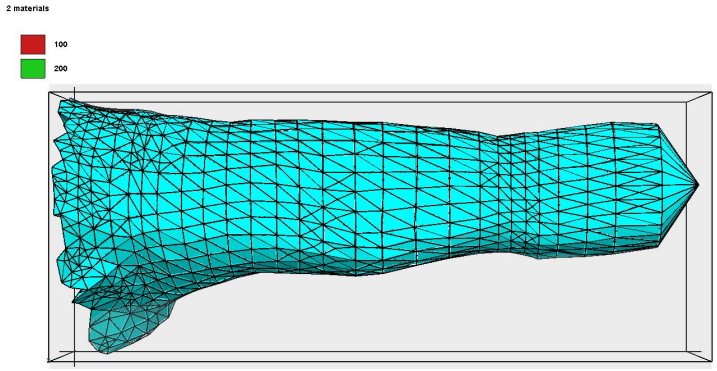


(b) Muscles and bones.

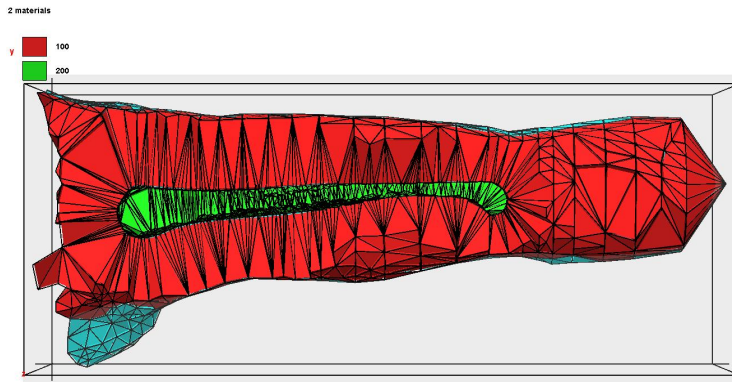
Figure 4.5: Polygon model of human man and woman. From [10]

### 4.2.1 Tetrahedral Arm

The first model that was created is a model of the arm, with only skin and the humerus bone from the upper arm. This was made to see how easy it is to create a tetrahedral mesh from the polygon model. The model does not have any nerves, muscles or veins in it, only the skin and the humerus bone. The skin of the arm and the humerus did not have many triangle intersections or open faces, making them easy to fix. The model has 8733 points, 108430 faces and 53563 tetrahedrons. Figure 4.6 shows two images from the tetrahedral mesh of the upper arm.




(a) The surface.




(b) With cut plane to see the bone.

Figure 4.6: Tetrahedral mesh of the upper arm with skin and bone.

### 4.2.2 Tetrahedral Shoulder

The next model that was created was the shoulder model, with the skin of the left upper arm and the left chest, the humerus [1], the clavicle [2], some of the ribs and the scapula[3]. The skin of the chest, the clavicle and the humerus did not have a lot of errors, and was not that hard to

[1] Bone in the upper arm
[2] Collarbone
[3] Shoulder blade

create a tetrahedral mesh from. The scapula had a lot of errors and the ribs had some. The left scapula in the model was all wrong, it was tuned inside out, making all the faces point into the model instead of out. The model has therefore been deleted and replaced with a mirror of the right scapula. The mirrored scapula was reported to have 279 intersections when *TetGen* was run with the "-d" switch. While repairing these a lot of other errors where found and fixed. But still after all the triangle intersections where repaired there where reports of open faces, how many open faces the model had and that have been fixed is not known, since a lot where fixed under the triangle intersection repair, and since TetGen only report the first one it finds. A good guess is that the number was over a 1000 in this model. Which is a lot, and that is way it took a very long time to find and fix all of them manually. The soft tissue is only defined as skin in this model. The model has 65632 points, 823696 faces [4] and 404957 tetrahedrons. Figure 4.7 shows the polygon model that the tetrahedral mesh is made from. The model is of the bones and the skin in the *.3DS* file format. Figure 4.8 shows two images from the tetrahedral mesh of the shoulder model.

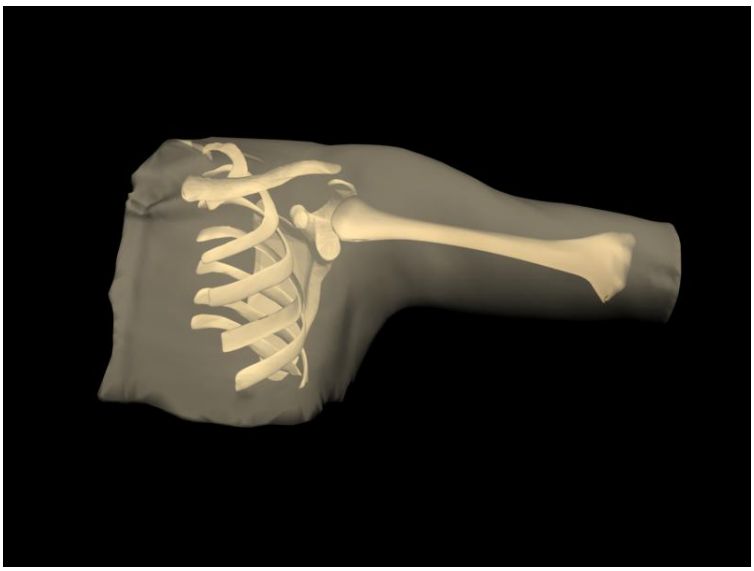


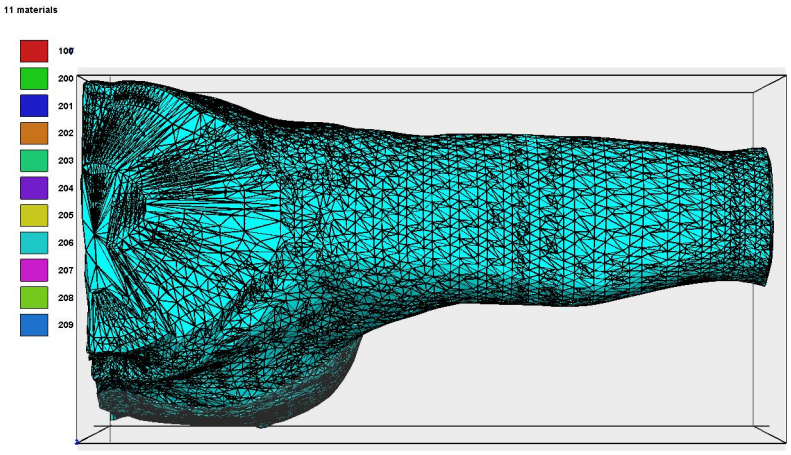
Figure 4.7: Shoulder polygon model (.3DS)

The model does not have the membrane between the humerus, clavicle and the scapula that is the goal of the simulation. This is a object that is required in the simulator, but it needs to be connected to the humerus without any open faces or triangle intersections, so it requires a lot of work to create this object. The spine was so full of errors and on the other side of the goal so it has been left out.

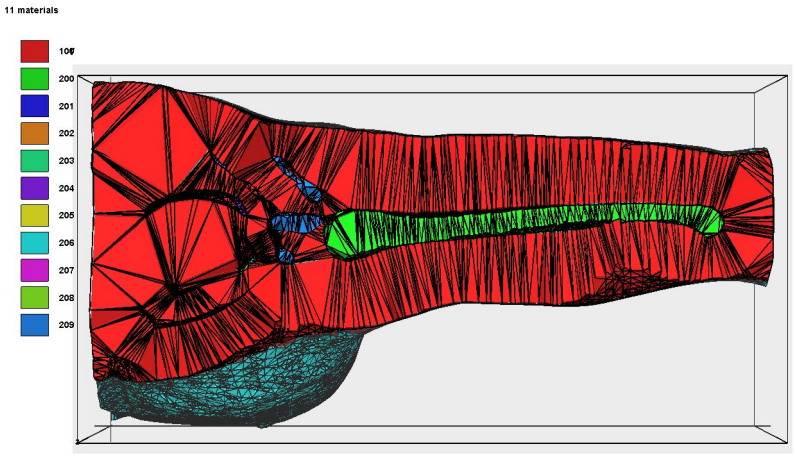
## 4.3 Needle Insertion

At the beginning of the project the plan was to let the simulator use the Reachin API[17] haptic scene graph to interact with a Sensable Phantom Desktop haptic device[16]. In April

[4] Does not know why TetGen reports only 823696 faces, $4 * 404957 = 1619828$ faces, and that is what is drawn in the scene graph.




(a) The surface.




(b) With cut plane to see the bones.

Figure 4.8: Tetrahedral mesh of the shoulder.

2007 Reachin released a new API called *HaptX*[1]. This project has been switched over to use *HaptX* since it is newer and not integrated with VRML. The *HaptX* haptic scene graph handles the haptic scene graph, but it does not have a 3D display API included. Code to display the scene needs to be written for either *OpenGL*[36] or *DirectX*[37]. There are example code for this in the *HaptX* examples.

The data model for the simulator is the shoulder tetrahedral mesh described in Chapter 4.2.2. The model is too detailed to work in a haptic scene graph on a normal computer. It needs some sort of *level of detail* (LOD) reduction for it to work in the haptic scene graph in realtime. There are too many faces in the model to be rendered 25 + times a second. This can be fixed since we do not need to see all the faces that is inside the volume of the model, we only need to see the edges of the different tissue types.

### 4.3.1 Syringe Model

A polygon model of a syringe has been found and downloaded from www.the3dstudio.com[11], it is a free model made in the *.max and .3ds* file format. Figure 4.9 shows an image of the syringe model. This is not a model of a syringe that could be used in the finished version of the simulator, it is not the right type of syringe. But it could be a start to test the idea of a needle insertion.

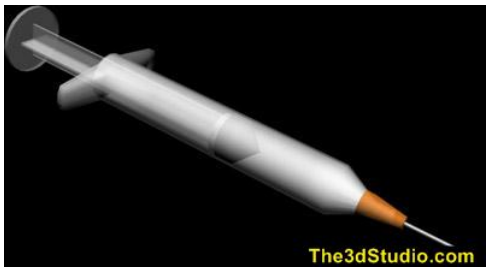



Figure 4.9: Syringe 3d model. From [11]

The HaptX example code has an "ASE" file loader. Therefore the model has been converted to the "ASE" file format, so that it can be loaded as the model for the needle in the simulator, but there is an error in the file and it will not be displayed. Whether it is the model or the converter in *3D Studio Max* that has an error is unknown. Converting any other files to the "ASE" file format has not been tested.

### 4.3.2 Haptic Scene Graph

The haptic scene graph needs to have the tetrahedral mesh of the model to calculate the forces and the deformations. When interaction with the tetrahedral mesh occurs, the visual model needs to be updated according to the forces that is applied to it. These calculations needs to be made in realtime. To do this some smart tricks are required.

### 4.3.3 Reachin API

In the start of the project the haptic scene graph used was the Reachin API[17]. It is not the perfect choice so when Reachin released the new haptic scene graph *HaptX*, it was decided that the project should switch over to this scene graph instead. It was possible to do this since the program was not that closely linked to the Reachin API. One of the main reasons for the switch is because Reachin API is so closely linked to the *VRML*[21] 3D language, which has not been used in this project.

### 4.3.4 HaptXTest

HaptX[1] is built up in a different way than the Reachin API. It is not linked to any 3D scene language like *VRML*, it does not have 3D rendering built into it and it is built so that it is easy to implement a physics engine like *ODE*[22]. The examples that follows with the API uses win32 code to load the program and a 3D rendering environment. It uses vertex arrays to draw the 3D objects. Because it uses vertex arrays it was a lot of work to rewrite the code that had been written to work with *HaptX* instead of Reachin API. Some of the source code for the *HaptXTest* program is listed in Chapter 9.2, but not all.

#### Tetrahedral Mesh

The *tetrahedral mesh* is loaded into the program in the *TetMesh* class. It holds a list of all the tetrahedrons and a list of all the vertexes, a list of the connections between the vertexes and the tetrahedrons, the number of tetrahedrons and the number of vertexes. It is this class that loads the tetrahedral mesh from the *.ele* and *.node* files. It has functions to connect the tetrahedrons to reduce the number of faces that needs to be rendered. It has a function to count the number of faces and vertexes in the different tissue types and then store these for later use, and a function to calculate the face normals of the tetrahedrons and a function to calculate the vertex normals from the sides that is going to be rendered. The *TISSUETYPE* enumerator is defined here, so that we do not need to remember what integer that represents the different tissue types.

In the *Vertex* class there are three floats that holds the *x, y, z* coordinates of the vertex and three floats that holds the vertex normal. It has methods to calculate the cross product and the length of a 3D vector. This is used to calculate the normals. These methods could have been somewhere else, and we could use the vertex struct in the HaptX example programs.

The *Tetrahedron* class holds a list with the indexes of the *four* vertexes in the vertex list, four face normals, four tetrahedron pointers to the neighboring tetrahedrons, four boolean values that keeps track of the sides that is going to be rendered and a tissue type variable.

There are two more data structures in the tetrahedal mesh part of the program, it is the *TissueType* which holds the number of vertexes and faces in the different tissue types, and the *Vertet* that stores a vector of the tetrahedral indexes for each vertex. The Vertet is used when connecting the tetrahedrons and when calculating the vertex normals. This is what makes it possible to calculate all the normals in seconds in stead of hours.

All these classes need some more error handling. There are very few methods that has anything to handle calls outside the legal range of the variables. A call like this will make the program crash, without any good error message output.

### Scene Graph

To make the scene graph the *Navigation* example from the HaptXCore was used as a starting point. This was chosen to make it load a tetrahedral mesh and display it. The reason to start with this demo was that it was necessary to turn the camera around to find the tetrahedral mesh and then transform it in front of the starting point of the camera. The demo uses vertex arrays to display the model, so the polygons needed to be in a vertex array, with the different tissue types defined as different models.

The program gets loaded using win32 code in the *main.cpp* file. This file is made by Reachin for the example code to the HaptX scene graph. It loads a scene from the *NeedleInsertion* class, and uses the *IGraphicsFramework*, the *IDemoFramework* and the *GraphicsOpenGL* classes to set up the HaptX scene graph and the *OpenGL* render.

The *InitDemo* method in the *NeedleInsertion* class is the one that loads the tetrahedral mesh and makes sure that the vertex array and the haptic scene is set up. In the *makeDispObjects* method in the *TetMesh* class the tissue types and the number of vertexes and faces in the different types are counted and stored in the *TissueType* data structure. This data structure is then used in the *LoadRenderModel(TetMesh *tetmesh)* method in the *IGraphicsFramework* class to make the renderer objects and the vertex arrays from the tetrahedral mesh for the rendering. This method should be changed, as it is now it loads all the vertexes as many times as they are used, this should be fixed so that it only has copies of the vertexes when the vertex normals is pointing in completely different directions. It is necessary to get the right normals for the front and back sides where two different tissue types meets. This also means that the *makeDispObjects* method in the *TetMesh* class needs to be changed, to count the right number of vertexes. The model is setup with textures, the texture coordinates are only sat to $0.0f$, this is not right and should be fixed. The best result would be to make the texture wrap around the whole object. This means that the texture coordinates needs to be calculated as to where the vertex is in the object. This is a hard task and would require a lot of work.

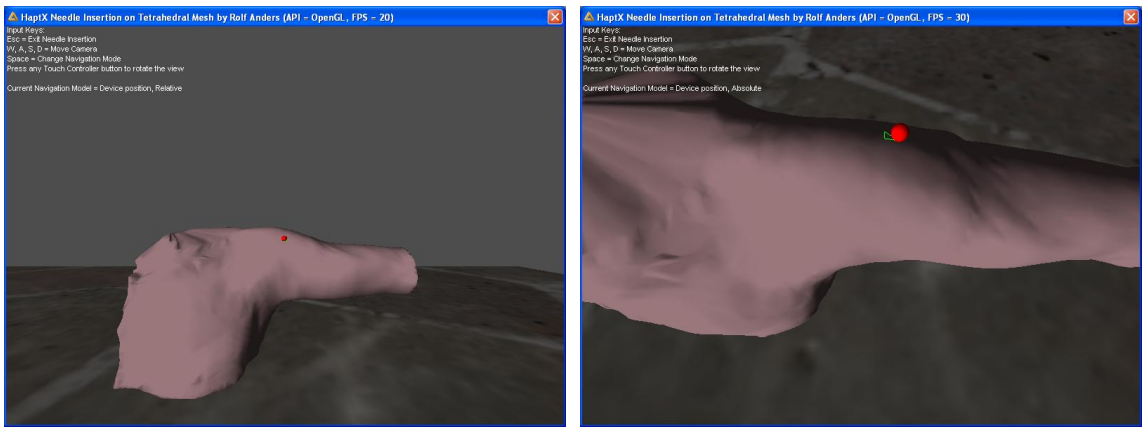
The vertex array is also used to make the haptic surfaces in the *CreateHapticsShape* method in the *NeedleInsertion* class. This method sets up a haptic surface for each of the objects. None of the surfaces has anything other than a simple surface with the default values. This makes the surfaces hard and impossible to penetrate with the haptic device. It is therefore not possible to touch the bones of the model.

There have been created a method called *LoadingFrame(std::string message)* in the *NeedleInsertion* class to load the screen with a text message on how far the loading of the tetrahedral mesh, the setup of the haptic scene and the visual scene has gotten. The view will only be updated when the *InitDemo* method sends a new message. This should be changed so that the frame is updated with more information on how far it has gotten. Not only with information from the *initDemo* method but also with information from the *TetMesh* class. This method

should be made so that it gets drawn more than once a second.

In the *UpdateDemo* method in the *NeedleInsertion* class the scene gets updated, the camera moved and the polygons that is hit with the haptic device gets found and marked with green lines. There needs to be a connection between the vertex arrays and the tetrahedral mesh, making it easy to find and edit the tetrahedral mesh and the polygons in the vertex arrays.

Figure 4.10 shows the outside of the shoulder in the HaptX scene graph, Figure 4.11 and Figure 4.12 shows the bones inside the shoulder tetrahedral mesh. The model has been textured, and you can see the polygons which is hit with the haptics device, as they are marked in green. All the different tissue types have been divided up into different objects and there have been added different surface materials to them. All the haptic materials are only defined as standard simple surfaces, making the surfaces hard, so the haptics device can not penetrate it. The model is rendered with more than 30 FPS[5], witch is enough to make a simulator work.




(a) Shoulder outside front.

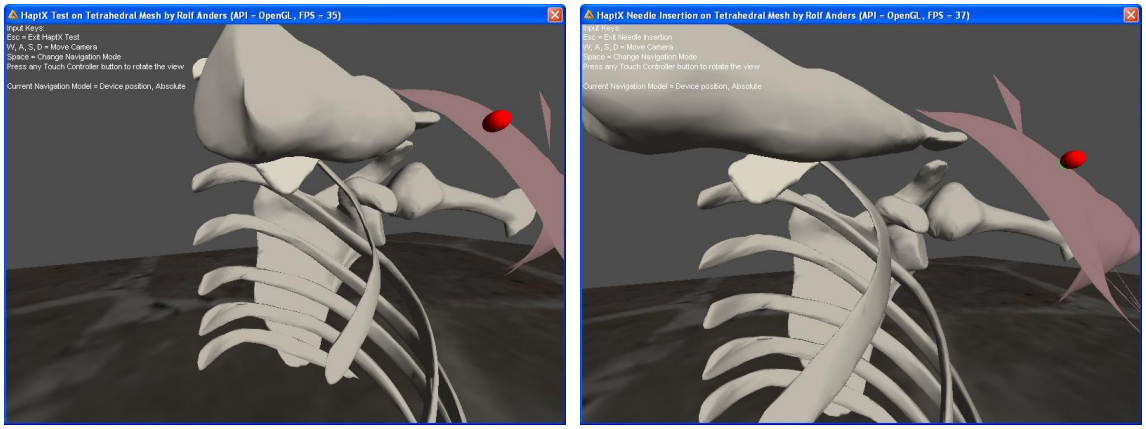
(b) Shoulder outside top.

Figure 4.10: HaptXTest program and the outside of the shoulder tetrahedral mesh.

In the HaptX example class *GraphicsRenderObject* the method *ComputeNormals* uses a nested for loop to calculate the vertex normals. It uses too much time to be used on the tetrahedral mesh and has been exchanged with the *calculateVertexNorm* method in the *TetMesh* class. It has been improved and now only uses a few seconds to calculate the vertex normals to the vertexes that is going to be rendered. It skips every vertex and face that is not going to be rendered. This method has an error, it calculates the vertex normal by adding the face normal to all the sides that is going to be rendered, and therefore the normals will be wrong inside the volume. This will happen where the vertexes has faces facing each other. This problem needs to be fixed by making it possible to add more normals to a vertex. The easy way to fix this is to duplicate the vertexes, but this might not be a good idea. The problem is not visible in the shoulder tetrahedral mesh, since there are no other parts in the model than the skin and the bones.

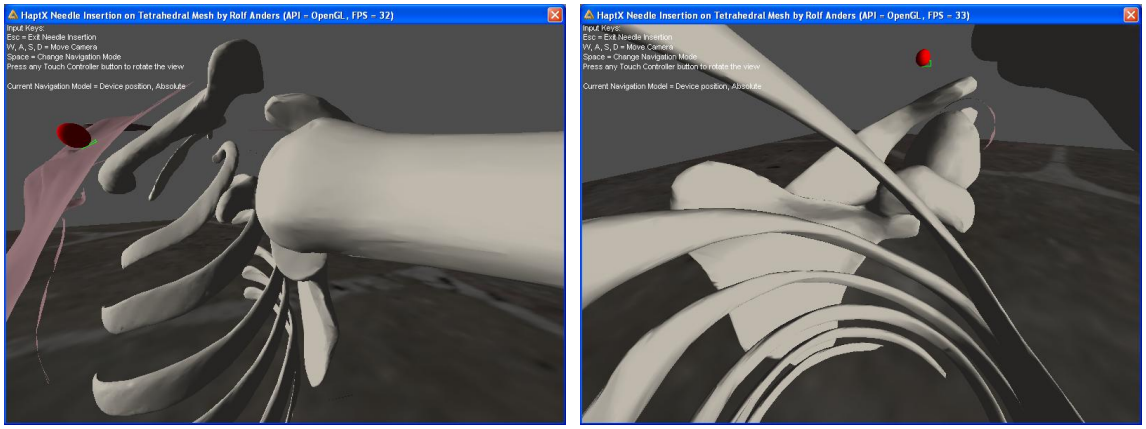
The code has been cleaned up so that it no longer uses more than 650 MB of RAM as it did, it now only uses 170 MB of RAM at the most when running the shoulder tetrahedral mesh.

[5] **F**rames **P**er **S**econd




(a) Shoulder inside. (b) Shoulder inside.

Figure 4.11: HaptXTest program and the inside of the shoulder tetrahedral mesh.




(a) Shoulder inside. (b) Shoulder inside.

Figure 4.12: HaptXTest program more shoulder tetrahedral mesh pictures.

**Running the Program**

HaptXTest starts to set up the graphical scene graph and it tries to load the *shoulder.1.ele* and *shoulder.1.node* tetrahedral mesh files. While the files are loading the tetrahedrons are connected, the normals are calculated, the display lists and the haptic surfaces are created, the screen displays a message about how far it has gotten. When everything is loaded the tetrahedral model is displayed and it is possible to interact with it with the haptics device. By using the *W, S, A, D and space* keys it is possible to move around in the scene and change the camera navigation model, by using the button on the haptics device is it possible to rotate the camera around.

It is possible to feel the surface of the skin, but it only has a hard surface, that it is impossible to penetrate. Therefore it is not possible to feel the surface of the bones, but they are there and can be touched if you could come into them. The triangles that is touched with the haptic device gets marked with green.

### 4.3.5 Triangle Reduction

To reduce the number of triangles that needs to be rendered a preprocessing stage to connect the tetrahedron has been implemented. This algorithm marks the end faces of the tissue type, so that only it gets rendered. This reduces the number of triangles that needs to be rendered in the shoulder model from 1.6$million$ triangles to 234836. This is done in the *connectTet* method in the *TetMesh* class. The first implementation of this was a $n^2$ algorithm, on the shoulder model this took hours to complete. It has now been fixed so that it only takes a little more than a minute. This is done by the *vertTet* data structure that holds a list over which tetrahedrons the vertexes are in, and then it only checks the tetrahedrons that has this vertex, not all the tetrahedrons.

The code has been improved even more and removes even more faces, so now it renders only 131200 faces. This is done by making the faces that is facing the bones disappear. These sides will never be visible and there are therefore no reason to render them. This also makes the error of the wrong vertex normals disappear in the *shoulder* model.

# Chapter 5

# Results

This chapter describes the results that have been obtained from the different parts of this project. It also contains a few thoughts about the different problems that has been encountered.

## 5.1 TetGen

*TetGen* has been edited to make a tetrahedral mesh from a *.stl* file with a *.regions* file defined. This makes it possible to create a mesh with different tissue types defined from a polygon model. The tetrahedral mesh that is created can be used in a medical simulator. *TetGen* does not make a *Hybrid Tetrahedral Mesh* as desired. This is not necessarily a big problem since the simulator has implemented a triangle reduction algorithm that removes all the inside of the volume. Therefore it might not be that important to get the mesh in a hybrid form.

## 5.2 Tetrahedral Mesh

There has been made a tetrahedral mesh of the shoulder region. The mesh does not have all the parts that it should have for a simulation in the shoulder. It does not have any more parts than the bones and the skin. The model has enough parts to make it possible to use as a demonstration model. It is possible to test and see if the other elements of the simulator could work with this mesh. It will work fine as a starting point for a FEM implementation.

## 5.3 FEM

As far as the *FEM* goes there have not been any implementation of it. Therefore there are no test or results on how well it would work. The only thing that has been done in this report is to describe the mathematics behind it, which is in Chapter 3.2.3.

## 5.4 Haptic Scene Graph

The *HaptX* scene graph is a brand new scene graph, and therefore there is not that much information about it available. It has no problems rendering the shoulder model haptically and graphically fast enough to work as the scene graph in the simulator, at least not when it uses vertex arrays. It renders the 131200 triangles in the shoulder model with more than 30 FPS, with all the objects in the mesh defined as haptical surfaces. This scene graph should work as the scene graph for the simulator.

## 5.5 Thoughts about problems

This section lists some thoughts about the problems that have been encountered while trying to make the simulator.

### 5.5.1 TetGen

The fact that *TetGen* is unable to create a mesh from a model that is not completely free of errors is a big problem. This problem needs to be solved with a program that has an automatic procedure to correct the triangle intersections and the open faces in the data model. To manually remove all these errors takes way to much time.

### 5.5.2 Data Model

The model used in this project only has bones and skin defined. For a optimal solution it needs to have more parts defined. It needs to have more than just skin and bones, at least it should have soft tissue under the skin and a membrane between the humerus and the scapula, which is the goal of the needle insertion.

The bones does not need a tetrahedral data structure, since they are not going to be deformed. Therefore the bones should only be defined by the surface of the objects as polygons. This would save a lot of data and could make everything go faster. As it is now the bones and all the internal faces of the model have been removed. Therefore there might not be any reason to make the bones as only a surface mesh, as it might work fine as it is now.

There needs to be a connection between the polygons in the vertex array of the haptic scene graph and the tetrahedrons, and back from the tetrahedrons to the polygons. This would make it possible to find the tetrahedrons that are touched with the haptic device and then deform the polygons that are affected. It is necessary to be able to only update the affected polygons. It would require too much work to update the whole model each time something is changed. These connections also need to be updated to the haptics vertex array, not only the visual vertex array.

### 5.5.3 Mesh Generation

The tetrahedral mesh should be generated from a MRI[12] volume. For an optimal solution the mesh should have bones, muscles, veins, arteries, nerves and skin, and not only skin and bones as the test shoulder model has. The mesh should be divided up into an operable and a non operable region. It might not be necessary to make a *Hybrid Tetrahedral Mesh* out of it, since there has been implemented ways to reduce the number of polygons and the tissue types are defined, making it easy to find the non operable parts. The FEM properties to the non operable parts can be identified and stored as they should in a preprocessing stage of the *FEM*.

### 5.5.4 FEM

There is still a big problem that needs to be solved and that is how to implement the *FEM*. The first problem is to find out how to get a *FEM* to work. Then there is the problem of making it work fast enough to work in realtime. There are so many variables and formulas that needs to be calculated and stored as they should for the *FEM* to work.

There is also the material properties of the tissue. They need to have properties close to that of the tissue in question. To make the FEM work fast enough it is necessary to make it run on multiple threads on multiple calculation devices, like multicore *CPU's*, *GPU's* or *PPU's*.

### 5.5.5 Simulation Files

It might be a good idea to make a file format for the simulator where all the precomputed data is stored. This would make it possible to load the simulator much faster than if it has to precompute all the normals, connection between the tetrahedrons, the *FEM* precomputations and the simulation specific data like transform and scale of the objects in the scene.

If all this and more had been stored in a file, the simulator would start a lot faster and it would be a lot easier to make different models and areas to simulate on. Then all the user would have to do to change where and what he wants to simulate on is to load a new simulator file. The file should then have a list of all the files that is needed for that simulation and load all of them automatically. When the file is selected the user could just sit back and wait for it to load. The set up of the scene and all the required elements should then be loaded automatically.

### 5.5.6 Needle Insertion

The SensAble Phantoms have a weakness for use in a needle insertion simulator: The last joint before the stylus is without a motor to stop or set forces on that joint. This means that the angle of the syringe can be changed without any feedback. This problem can be temporary fixed by the forces in the model, meaning that the model does not change even though the stylus does. It is also possible to implement a force on the other motors to make the entire haptic device shake

and then give a warning that you can not do this. The best solution is to get a haptic device that has motors in all the joints.

### 5.5.7 Haptic Device

For the optimal simulator there should be made a new haptics device with motors to put forces on the last link before the stylus. This link does not have any motors on any of the SensAble Phantoms. If it had a motor there it would improve the user's experience and make it work a lot better. There are no haptic devices like this out on the marked, which means that someone needs to design and produce it. This is not a requirement for the simulator to work, it would just make the simulator a lot better.

# Chapter 6

# Further Work

This chapter describes the tings that needs to be done to make this a complete simulator. It also describes the things that should be done to make the simulator as good as possible.

## 6.1 Mesh Generation

The *tetrahedral mesh* that has been used in this project has been made from a polygon model by using a edited version of *TetGen*. The mesh goes not have anything more than skin and bones.

The mesh should be made form a *MRI* data set to get an optimal model. A program that automatically segments and creates a tetrahedral mesh from the segmented MRI data set should be written. The mesh does not need to be as fine as possible, but we need to be able to divide the tetrahedrons up into smaller ones as they get punctured by the syringe, inserting new vertexes and using them to create new tetrahedrons.

The segmentation should be able to segment out more parts of the tissue than only skin and bone. This is a big challenge. The model should have veins, nerves, muscles and bone defined. This is a extremely hard task that would require a lot of work. The program would need some variables to control how the loaded volume should be segmented. What sort of file format the segmented files should have is open, since the only purpose of it is to load it into a tetrahedralization algorithm. The only requirement for the file format is that it needs to be in a format that can easily be loaded into the next step, to create a tetrahedral mesh. The model needs to be free of errors, there can not be any triangle intersections or open faces in the model.

When the volume has been segmented it should be tetrahedralized using *Delaunay tetrahedralization*. This is also a task that requires a lot of work, but it is possible to use existing programs like *TetGen*. The best solution though would be to write or edit a program that makes a mesh that is perfect for the use in a medical simulator.

## 6.2 FEM

There have been made no attempts at implementing a *FEM*[1]. The mathematics for the selected FEM are described in Chapter 3.2.3. The different tissue types need to have material properties that are close to what the properties of that tissue type is. The tetrahedrons need to have the volume in them calculated and stored.

There is a lot of work that needs to be done before a *FEM* is up and running. This is a real challenge but it is vital to make a simulator that is as good as possible.

The *FEM* needs to be as optimized as possible. This can be done with the use of threads and the new multi core CPU's, or by GPU's[2] or PPU's[3]. What the best solution would be is hard to guess, that needs to be tested.

## 6.3 Simulator

As it is now the simulator is only a haptic scene which displays a tetrahedral mesh, making it possible to touch the objects in the mesh with the haptics device. There is still a lot of work that needs to be done before it is a simulator. When the data model is as it should be and the *FEM* is implemented, there are still a lot of tings that needs to be fixed so that the simulator gets as close to a real needle insertion as possible.

The vertexes in the vertex array has at this point a vertex for each point in every face. This should be fixed so that every vertex is in the vertex array only once. It is necessary to check if the vertex is in the vertex array already. If it is we only need to add the index of it to the face list. It might be an idea to store in the vertex class the index of the vertex in the vertex array. Then we would have a connection between the tetrahedrons and the vertex array. This connection would not be a two way connection as we should have, but we have a connection at least.

It is necessary to fix the calculation of the vertex normals. As it is now it makes a normal from all the faces that is going to be rendered. This is correct as long as all the faces points one way, but when the faces are face to face this is not correct. This happens when the faces from two different tissue types are facing each other. Therefore it might be necessary to double the vertexes in the vertex arrays where this is an issue, making the normals point the right way.

It is also necessary to be able to change the soft tissue's alpha value, making it transparent. This would make it possible to see through the soft tissue and down to the bone and the goal of the simulation. To accomplish this, blending has to be turned on and the objects need to be drawn in the correct order, starting with the bone and working our way out to the skin. This means that the objects need to be sorted out from their position in the model. This should not be too hard to implement. It should not be too hard to implement something to change the alpha value

[1] **F**inite **E**lement **M**ethod
[2] **G**raphics **P**rocessing **U**nite
[3] **P**hysics **P**rocessing **U**nite

of the soft tissue objects either. The best solution would be to make it possible to change the different objects alpha values independently. This is possible, but to do this a GUI[4] window with a list of the objects and a slide bar or some other form to change the value is required.

The simulator needs a model of a syringe. A free syringe model has been found and downloaded from the internet, but it is not of the right type, and when converted to the *.ASE* file format it would not be loaded into the program. The model needs to be of the correct type of syringe, and it needs to be scaled to fit the model. The syringe model might need to be in a tetrahedral mesh to interact with the simulation model as it should.

### 6.3.1 Reachin Display

The simulator should also implement the possibility of using a *Reachin Display* and *stereo goggles*. This would improve the user's experience from the simulator since the user would get a better feeling for the depth of the objects in the scene.

*HaptX* does not support Reachin Display or stereo goggles. Stereo goggles need to be implemented by using an external API in *HaptX*. How it is possible to get the stereo goggles to work in *HaptX* is described in Chapter 2.2.4. To make the Reachin Display work the scene needs only to be rotated, so there is no need for an external API for this.

### 6.3.2 Dividing The Simulator In Two

A good idea is to divide the simulation up into two parts. One where the syringe is exchanged with a *hand with a finger or just a finger*. The finger should be used in the first part of the simulation to only feel the surface of the skin and the bones beneath it. When pushing down, the skin should deform making you feel the contours of the bones and the muscles like when you push a finger on the surface of a real person's skin. This should make it possible to find the perfect spot to do the syringe insertion. When the right spot is found, the hand should be exchanged with the syringe and the second part of the simulation should start with the needle insertion.

To make this work a model of a hand or a finger is also needed and the program needs to make it impossible for the finger to go through the skin, it should only be able to deform the skin as long as forces are put on it. This means that the simulator needs to operate with different physics for the finger and the syringe model.

### 6.3.3 Demonstration

The simulator should have a way to demonstrate how the simulator works and what the goal of the simulation should be. One way to do this is by running a prerecorded video sequence of the simulation where a person explains the user what is done and how to do it. It is also possible

[4] **G**raphics **U**ser **I**nterface

to do this by giving hints to the user during the simulation by drawing on the model, or a voice that tells the user what to do. This would make it feel like there is a mentor standing there to assist the user.

The files for the demonstration needs to be stored and loaded automatically when the simulation file is loaded. It is necessary to be able to turn the demonstration function on or off if it is done interactively.

### 6.3.4 Addons

To improve the simulation even further it is possible to add animations that *animate the injection* when the user thinks the right spot has been found. This means that the syringe is filled with liquid, and then when the user pushes a button the liquid flows out of the syringe.

There is also the possibility of adding animations, forces and other effects when the user for example hits an artery or a vein. The needle hole could be made to bleed when this happens, making blood flow out from the puncture wound. This would be a really cool feature, but then the model needs to have the arteries and veins defined, which adds to the complexity of the model and the simulation.

It would also be cool if the program had implemented sound, that could make the program scream if the user did something that would hurt in a real needle insertion, like hitting a nerve or by using too much force and hitting the bone too hard.

# Chapter 7

# Conclusion

The task of creating a medical simulator that can be used to train medical personnel is a big task. It requires a lot of resources to make it good enough for this purpose. Which means that the simulator has to mimic the real situation and do this as close to the real needle insertion as possible. This is why the *FEM* solution with the use of *tetrahedrons* has been chosen. It is possible to emulate a *FEM* much closer to reality, than with any of the other deformation possibilities.

The haptic scene graph has been selected out from the criteria that it has to be made as a haptic scene graph, and it has to be programmable with *C++*. This means that scene graphs like *H3D*[3] and *Open Scene Graph* with *OSGHaptics*[4] was not chosen. The choice fell on *Reachin API*, even though it was not a perfect choice, since it is too closely connected with *VRML*. When Reachin released its new scene graph, the *HaptX* in April, it seemed to be a good idea to switch over to this instead. The final version of the program uses this scene graph. The *HaptX* scene graph seems to have all the desired functions, and it works as it should for the things that have been tested.

The data model that has been created, has been created from a polygon model. The polygon model is full of errors and therefore the shoulder model does not have all the parts that is desired, it only has the skin and the bones. The skin and the bones is all that is required to implement and test a *FEM* on. When the *FEM* is up and running the model can be exchanged with one that has more parts, like muscles, veins and so on.

It has been a real challenge to find information about how to use a *FEM* in a simulator and how to create a *tetrahedral mesh* for this purpose. This task would have been a lot easier if there was anybody with knowledge of this to talk to.

The main task of this project was to make a *tetrahedral mesh* that can be used in a *FEM* in a needle insertion simulator. This task has been solved with the *shoulder tetrahedral mesh*, even though the mesh does not have all the wanted parts. Though it has the parts that is important for further development of the simulator. The mesh generation procedure should be improved, in order to correct the errors in the polygon mesh automatically. The best solution would be to create a mesh directly from a *MRI* volume, by first automatically segmenting the volume and then creating a tetrahedral mesh from the segmented volume.

So far there has only been created a tetrahedral mesh of the shoulder which only has the bones and the skin, and a program to load this model into a haptic scene graph. In the scene graph program, the mesh has been divided into different objects. The objects have been made haptical by adding simple haptical surfaces to them, making it possible to feel the surface of the objects. The goal of this project was to create a tetrahedral mesh that it is possible to test a *FEM* on. In that respect the goal of the project has been met.

The task of making a complete simulator is too big to complete within such a short time span, and with so few resources. Therefore not even all the parts of the simulator has been started on, and no parts are complete. There is still a lot of work that needs to be done on the medical simulator before it is close to something that can be called a simulator. The tetrahedral mesh should be improved by adding more tissue types to it. The *FEM* needs to be implemented, and there are a lot of small things that need to be fixed or changed to make everything work as it should.

In this report the *tetrahedral mesh* creation and the setup of the haptic scene graph has been described. There has also been discussed a lot of possible solutions, problems, ideas and several other things that would have been nice to have in the simulator. All this to create a simulator that is as realistic as possible and educational and interesting to use.

# Chapter 8

# Acknowledgment

This project is given by Assistant Professor Torbjørn Hallgren at IDI, NTNU[1][38], IDI[2] [39] in cooperation with Doctor Jostein Halgunset at the St. Olavs Hospital[40] in Trondheim.

*MRI* and *CT* data has been given by Geir Arne Tangen Research Scientist (MSc) SINTEF Health Reserch[41], Medical Technology.

The tetrahedral mesh is generated with the use of TetGen[31] and could not have been done without the program made by *Hang Si* at the Research Group: Numerical Mathematics and Scientific Computing at Weierstrass Institute for Applied Analysis and Stochastics.

[1]The Norwegian University of Science and Technology
[2]The Department of Computer and Information Science www.idi.ntnu.no

# Bibliography


[1] Reachin Technologies AB. Haptx - haptic scene graph. http://www.haptx.com.

[2] Novint. Novint falcon haptic device. http://www.novint.com/falcon.htm.

[3] SenseGraphics AB. H3d - open source haptics. http://www.h3d.org/.

[4] Umeå University VRlab. Openscenegraph haptic library (osghaptics). http://www.vrlab.umu.se/research/osgHaptics/.

[5] Reachin Technologies AB. Reachin technologies ab. http://www.reachin.se.

[6] Francone Jacob and Lossow. *Anatomi og fysiologi*. Universitetsforlaget, 1997.

[7] Wen Wu, Pheng Ann Heng. The Chinese University of Hong Kong. Department of Computer Science, and Engineering. An improved scheme of an interactive finite element model for 3d soft-tissue cutting and deformation. http://appsrv.cse.cuhk.edu.hk/~vrcentre/pub/download/visualcomputer.pdf, 2005.

[8] Wikimedia Foundation Inc. Tetrahedron. http://en.wikipedia.org/wiki/Tetrahedron.

[9] Hang Si Research Group of Numerical Mathematics, Scientific Computing Weierstrass Institute for Applied Analysis, and Stochastics. Tetview - a tetrahedral mesh and piecewise linear complex viewer. http://tetgen.berlios.de/tetview.html.

[10] 3D Special. Human anatomy model, anatomy value set 1, a high detail set of human body systems - skeleton, muscles, cardiovascular, nerve and lymph. www.3-d-models.com/3d-model_files/anatomyM1.htm.

[11] James Murray The 3D studio. Syringe 3d model. http://www.the3dstudio.com/product_details.aspx?id_product=3056.

[12] Wikipedia. Mri (magnetic resonance imaging). http://en.wikipedia.org/wiki/Magnetic_resonance_imaging.

[13] NetDoctor.co.uk. Ct(computerised tomography). http://www.netdoctor.co.uk/health_advice/examinations/ctgeneral.htm.

[14] Rolf Anders Syvertsen. Needle insertion simulator applying a haptic device. http://folk.ntnu.no/rolfans/Needle_Insertion.pdf, 2006.

[15] Tien-Tsin Wong Yangsheng Xu Yim-Pan Chui Kai-Ming Chan Pheng-Ann Heng, Chun-Yiu Cheng, Shiu-Kit Tso. The Chinese University of Hong Kong. Department of Computer Science, and Engineering. A virtual reality training system for knee arthroscopic surgery. http://www.cse.cuhk.edu.hk/~ttwong/papers/arthro/arthro2.pdf, 2004.

[16] Sensable Technologies. Phantom desktop haptic device. http://www.sensable.com/products/phantom_ghost/phantom-desktop.asp.

[17] Reachin Technologies AB. Reachin api 4.1. http://www.reachin.se/products/ReachinAPI/.

[18] ISO/IEC. X3d - the successor to the virtual reality modeling language (vrml). http://www.web3d.org/x3d/.

[19] Python Software Foundation. Python programming language. http://www.python.org/.

[20] OSG Community. Open scene graph. http://www.openscenegraph.com.

[21] ISO/IEC. Virtual reality modeling language. http://www.web3d.org/x3d/specifications/vrml/.

[22] Russell Smith. Open dynamics engine. http://www.ode.org.

[23] StereoGraphics Corporation. Writing stereoscopic software for opengl. http://www.stereographics.com/support/developers/pcsdk.htm.

[24] Real D. Oglplane - stereoscopic example program. http://www.reald-corporate.com/scientific/developer_tools.asp.

[25] nVidia. Stereoapi - directx stereoscopic api. http://download.developer.nvidia.com/developer/SDK/Individual_Samples/3dgraphics_samples.html.

[26] A. Frank van der Stappen Han-Wen Nienhuys. Cutting in deformable objects, the thesis. http://www.cs.uu.nl/groups/AA/virtual/surgery/thesis/, 2003.

[27] A. F. van der Stappen H. W. Nienhuys. Supporting cuts and finite element deformation in interactive surgery simulation. Technical Report UU-CS-2001-16, Institute of Information and Computing Sciences, Utrecht University, 2001.

[28] A. Frank van der Stappen Han-Wen Nienhuys. Interactive needle insertions in 3d nonlinear material. Technical Report UU-CS-2003-019, Institute of Information and Computing Sciences, Utrecht University, 2003.

[29] Inc. AGEIA Technologies. Physx physics processing unit and sdk. http://www.ageia.com.

[30] Wikipedia. Delaunay triangulation. http://en.wikipedia.org/wiki/Delaunay_triangulation.

[31] Hang Si Research Group of Numerical Mathematics, Scientific Computing Weierstrass Institute for Applied Analysis, and Stochastics. Tetgen - a quality tetrahedral mesh generator and three-dimensional delaunay triangulator. http://tetgen.berlios.de/index.html.

[32] Hang Si Research Group of Numerical Mathematics, Scientific Computing Weierstrass Institute for Applied Analysis, and Stochastics. poly file format. http://tetgen.berlios.de/fformats.poly.html.

[33] Wikipedia. stl file format. http://en.wikipedia.org/wiki/STL_(file_format).

[34] Autodesk. 3d studio max. http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112.

[35] Blender Foundation. Blender. http://www.blender.org/.

[36] SGI. Opengl. http://www.opengl.org/.

[37] Microsoft. Directx. http://www.microsoft.com/directx/.

[38] Ntnu (the norwegian university of science and technology). www.ntnu.no.

[39] NTNU. Idi (the department of computer and information science). www.idi.ntnu.no.

[40] St. olavs hospital. http://www.stolav.no.

[41] Sintef health research. http://www.sintef.no/content/page2____333.aspx.

# Chapter 9

# Appendix

## 9.1 TetGen

Here is the part of TetGen that as been the most edited. The file loading of the *stl* files.

Listing 9.1: TetGen - tetgen.cxx

```
bool tetgenio::load_stl(char* filename)
{
        // from line 1815 this has been added
  //-------------------------------------------------------

  // Adding regions from regions file.
  // Added by Rolf Anders Syvertsen 01.03.2007

  // rolfans@stud.ntnu.no

    FILE *regionFile;
    char inregionsfilename[FILENAMESIZE];

    strcpy(inregionsfilename, filename);

    strcat(inregionsfilename, ".regions");

    char *stringptr;
    int index;

    char inputline[INPUTLINESIZE];

    regionFile = fopen(inregionsfilename, "r");



    if (regionFile != (FILE *) NULL) {
```

```
    printf("Found regions file. Loading regions.\n");
    stringptr = readnumberline(inputline,regionFile, NULL);
    if (stringptr != (char *) NULL && *stringptr != '\0') {
      numberofregions = (int) strtol (stringptr, &stringptr, 0);
    }
    else {

     numberofregions = 0;

  }

  if (numberofregions > 0)
  {

    // Initialize 'regionlist'.

    regionlist = new REAL[numberofregions * 5];
    index = 0;
   for (i = 0; i < numberofregions; i++) {
      stringptr = readnumberline(inputline, regionFile,
         inregionsfilename);
      stringptr = findnextnumber(stringptr);
      if (*stringptr == '\0') {
        printf("Error:  Region %d has no x coordinate.\n",
           firstnumber + i);
        break;
      }
      else {
        regionlist[index++] = (REAL) strtod(stringptr, &stringptr
           );
      }
      stringptr = findnextnumber(stringptr);
      if (*stringptr == '\0') {
        printf("Error:  Region %d has no y coordinate.\n",
           firstnumber + i);
        break;
      }
      else {
        regionlist[index++] = (REAL) strtod(stringptr, &stringptr
           );
      }

      stringptr = findnextnumber(stringptr);
      if (*stringptr == '\0') {
        printf("Error:  Region %d has no z coordinate.\n",
           firstnumber + i);
        break;
```

```
        } else {
          regionlist[index++] = (REAL) strtod(stringptr, &stringptr
              );
        }
        stringptr = findnextnumber(stringptr);
        if (*stringptr == '\0') {
          printf("Error:  Region %d has no region attrib.\n",
              firstnumber + i);
          break;
        } else {

          regionlist[index++] = (REAL) strtod(stringptr, &stringptr
              );
        }
        stringptr = findnextnumber(stringptr);
        if (*stringptr == '\0') {
          regionlist[index] = regionlist[index - 1];
        } else {
          regionlist[index] = (REAL) strtod(stringptr, &stringptr);
        }
        index++;
      }
      if (i < numberofregions) {
        // This must be caused by an error.
        fclose(regionFile);
        return false;
      }
    }
  }
       else {
              numberofregions=0;
       }

}
```

## 9.2 HaptXTest

Here is some of the code from the HaptX test program.

### 9.2.1 H Files

Listing 9.2: HaptXTest - needleinsertion.h

```
/** This file is based on the HaptX example kode
```

```
        Tetrahedral mesh viewer for HaptX
        using the HaptX demo framework
        Edited by Rolf Anders Syvertsen
        04.05.2007
        rolfans@stud.ntnu.no
        rolfans@gmail.com
*/

#ifndef NEEDLEINSERTION_H
#define NEEDLEINSERTION_H

#include "HaptX/graphics_framework.h"
#include "HaptX/demo_framework.h"

#include "tetmesh.h"

class NeedleInsertion : public IDemoFramework
{
private:
        IGraphicsFramework*             m_graphics;
        inputState_t*                   m_input;
        IGraphicsRenderModel*   m_model;
        IGraphicsRenderModel*   m_proxyModel;
        IGraphicsRenderModel*   m_floor;
        IGraphicsRenderModel*   m_syringe;

        float                                   cx;
        float                                   cy;

        float                                   m_camSensitivity;
        bool                                    m_prevButton1;
        bool                                    m_prevButton2;

        int
           m_activeNavigationModel;

        // the tetrahedral mesh
        TetMesh *m_ptetmesh;


private:
        void HandleMoveCamera(float deltaTime);
        void HandleCameraRotation(float x, float y);

public:
        NeedleInsertion();
        virtual ~NeedleInsertion();
```

```
        const char*                 GetDemoName();
        // load the model
        bool                        InitDemo(IGraphicsFramework* gfx,
           inputState_t* inputState);
        //! displaying the scene and a message
        void                        LoadingFrame(std::string message);
        //! create the haptics shapes
        void                        CreateHapticsShape();
        void                        ShutDownDemo();
        void                        UpdateDemo(float deltaTime, int
           width, int height);
};

#endif
```

Listing 9.3: HaptXTest - tetmesh.h

```
/**     tetrahedral mesh for the HaptX scene graph
        made by Rolf Anders Syvertsen
        2007-04-13
        rolfans@stud.ntnu.no
        rolfans@gmail.com
*/
#ifndef TETMESH_H
#define TETMESH_H

#include <stdio.h>
#include <iostream>
#include <windows.h>
#include<gl/gl.h>
#include<gl/glu.h>

#include <math.h>
#include <vector>

#include "tetrahedron.h"
#include "vertex.h"
#include "vertet.h"
#include "tissuetype.h"

using namespace std;

/** The loading of the tetrahedral mesh is based on the loader in
*       TetGen.
*       Loads meshes from .ele and .node files
*/
class TetMesh
{
public:
```

```
        //! Constructor
        TetMesh();
        //! Destructor
        ~TetMesh();

        //! load tetrahedral mesh from a .ele and a .node file
        bool loadTetMesh(char *filename);
        //! load the nodes from the .node file
        bool loadNodes(FILE* infile, int markers, char* infilename);

        //! calculate face normals for tetrahedron at
        void calcFaceNorm(int tetind);

        //! the number of tissue types
        int numTissueTypes();
        //! the tissueTypes
        TissueType *getTissueData(){return m_ptissue;};

        //! the number of tets
        int numTets(){return m_numTet;}
        //! the tet at
        Tetrahedron *getTetAt(int index){return &m_ptetList[index];}
        //! the vertex at
        Vertex getVertexAt(int index);

        //! types of tissue
        enum TISSUETYPE{SKIN=100, SOFTTISSUE=101, BLOODVEINS=102,
                BLOODARTERIES=103, NERVES=104, LUNG=105, HART=106,
                MEMBRANE=107,
                BONE=200, // should be from 200 -> 299
                MUSCLES=300 // should be from 300 -> 399

        };

protected:
        //! connect the tetrahedrons
        void connectTet();

        //! calculate the vertex normals of the
        //! vertexes that will be rendered
        void calculateVertexNorm();

        //! makes lists with the vertexes and faces in the
        //  polygon model for displaying
        void makeDispObjects();

private:
```

```
        //! the number of tissue types
        int m_numTissueTypes;

        //! the number of faces in the differen tissue types
        TissueType *m_ptissue;

        //! Maximum number of characters in a file name (including
           the null).
    enum {FILENAMESIZE = 1024};

    // Maxi. numbers of chars in a line read from a file (incl. the
       null).
    enum {INPUTLINESIZE = 1024};

        //! read a number line from a infile
    char *readnumberline(char* string, FILE* infile, char*
       infilename);
        //! find the next number, skipping spaces and comments
    char *findnextnumber(char* string);

        //! tetrahedron list
        Tetrahedron *m_ptetList;
        //! number of tetrahedrons
        int m_numTet;
        //! number of tetrahedron attributes
        int m_numTetAttrib;

        //! vertex list
        Vertex *m_pvertexList;
        //! number of vertexes
        int m_numVertexes;

        //! The connection between the Vertexes and
        //! the tetrahedrons
        Vertet *m_pvertexTet;

        //! the dimention of the mesh
        int m_meshdim;
        //! the first vertex index in the file
        int m_firstnum;
};
#endif
```

Listing 9.4: HaptXTest - tetrahedron.h

```
/**     tetrahedron data structure
        made by Rolf Anders Syvertsen
        2007-04-13
        rolfans@stud.ntnu.no
```

```
*/

#ifndef TETRATEDRON_H
#define TETRAHEDRON_H

#include "vertex.h"
/** The neighboring tetrahedrons:
        m_pfaceNeighb1= vertex0 + vertex1 + vertex2
        m_pfaceNeighb2= vertex0 + vertex2 + vertex3
        m_pfaceNeighb3= vertex0 + vertex3 + vertex1
        m_pfaceNeighb4= vertex1 + vertex3 + vertex2

*/
class Tetrahedron
{
public:
        //! constructor
        Tetrahedron();
        //! destructor
        ~Tetrahedron();
        //! add the corners
        void addCorners(int verti1,int verti2, int verti3, int
           verti4);
        //! set the tissue type
        void setType(int type);
        //! get the tissue type
        int getType();
        //! get the vertex indexes
        int *getVertex();
        //! set face normals
        void setFaceNorm(int ind,Vertex norm);
        //! get Face Normals at
        Vertex getFaceNormAt(int index);

        //! set neighbor tet at ind 0-3
        //! sets the faces that should be rendered
        void setNeighbor(int ind, Tetrahedron *tet);
        //! how many neighbors are sett
        int numNeighbors();
        //! is the face at ind going to be rendered
        bool renderFaceAt(int ind);

private:
        //! Vertex indexes for the four vertexes
        int m_vertex[4];
        //! Face normals
        Vertex m_faceNormal[4];
        //! tetrahedron type
```

```
        int m_type;

        //! neighbor tetrahedrons
        Tetrahedron *m_pfaceNeighb1,*m_pfaceNeighb2,*m_pfaceNeighb3
            ,*m_pfaceNeighb4;
        //! is the faces going to be rendered
        bool m_renderface[4];
        //! number of tet neighbors
        int m_neighbors;

};
#endif
```

Listing 9.5: HaptXTest - tissuetype.h

```
/**     number of faces and vertexes in this tissue type
        made by Rolf Anders Syvertsen
        2007-05-08
        rolfans@stud.ntnu.no
*/
#ifndef TISSUETYPE_H
#define TISSUETYPE_H


class TissueType
{
public:
        //! constructor
        TissueType(){m_type=-9999;m_numVertexes=0;m_numFaces=0;}
        //! the type
        int m_type;
        //! number of vertexes
        int m_numVertexes;
        //! number of faces
        int m_numFaces;

};
#endif
```

Listing 9.6: HaptXTest - vertet.h

```
/**     Vertex tetrahedron vector
        used to connect the vertexes and tetrahedrons
        made by Rolf Anders Syvertsen
        2007-04-13
        rolfans@stud.ntnu.no
*/
#ifndef VERTET_H
#define VERTET_H
```

```
#include <vector>

using namespace std;

class Vertet
{
public:
        Vertet(){m_tetid.clear();}

        vector<int> m_tetid;
};
#endif
```

Listing 9.7: HaptXTest - vertex.h

```
/**     Vertex data structure
        made by Rolf Anders Syvertsen
        2007-04-13
        rolfans@stud.ntnu.no
*/
#ifndef VERTEX_H
#define VERTEX_H

#include<math.h>
/** Also defines a 3d vector
*/
class Vertex
{
public:
        //! constructor with the dim.
        Vertex();
        //! set the point
        void setPoint(float x,float y, float z);
        //! get the x coord
        float getX();
        //! get the y coord
        float getY();
        //! get the z coord
        float getZ();

        //! set the normal
        void setNormal(float x,float y, float z);
        //! get the x Normal coord
        float getNormalX();
        //! get the y Normal coord
        float getNormalY();
        //! get the z Normal coord
        float getNormalZ();
```

```
        //! cross product
        Vertex cross(Vertex v2);
        //! the lengt of the vector
        float length();

        //! * operator overloader
        Vertex operator *(float scalar);

private:
        //! The vertex
        float m_vertex[3];
        //! The vertex normal
        float m_normal[3];

};
#endif
```

### 9.2.2 Cpp Files

Listing 9.8: HaptXTest - graphicsframework.cpp

```
// from line 303
IGraphicsRenderModel* IGraphicsFramework::LoadRenderModel(TetMesh *
    tetmesh)
{
        IGraphicsRenderModel* m = CreateRenderModel();

        if (m)
        {
                TissueType *tissue=tetmesh->getTissueData();
                for (int i = 0; i < tetmesh->numTissueTypes(); i++)
                {
                        GraphicsRenderObject obj;
                        m->m_objectArray.push_back(obj);
                        size_t id = m->m_objectArray.size() - 1;
                        int vertid=0; int faceid=0;

                        // texture
                        m->m_objectArray[id].m_tissueType=tissue[i].
                            m_type;
                        if (tissue[i].m_type==TetMesh::SKIN)
                        {
                                m->m_objectArray[id].m_colormap =
                                    LoadTexture("tissuext.bmp");
                        }
```

```
else if (tissue[i].m_type>=TetMesh::BONE&&
    tissue[i].m_type<TetMesh::MUSCLES)
{
        m->m_objectArray[id].m_colormap =
            LoadTexture("bonetile.bmp");
}
else if (tissue[i].m_type>=TetMesh::MUSCLES
   &&tissue[i].m_type<400)
{
        m->m_objectArray[id].m_colormap =
            LoadTexture("musclet.bmp");
}

int vertex_count = tissue[i].m_numVertexes;
m->m_objectArray[id].m_numVertices =
    vertex_count;
m->m_objectArray[id].m_vertices = new
    vertex_t[vertex_count];
memset(m->m_objectArray[id].m_vertices,0,
    sizeof(vertex_t) * vertex_count);

int face_count = tissue[i].m_numFaces;
m->m_objectArray[id].m_numFaces = face_count
    ;
m->m_objectArray[id].m_faces = new unsigned
    int[face_count * 3];

// set the faces and vertexes
for(int j=0;j<tetmesh->numTets();j++)
{
        Tetrahedron *tetrahe=tetmesh->
            getTetAt(j);
        if(tetrahe->getType()==tissue[i].
            m_type)
        {
                if(tetrahe->renderFaceAt(0))
                {
                        int *vertindex=
                            tetrahe->
                            getVertex();
                        // vertex 0
                        m->m_objectArray[id
                            ].m_vertices[
                            vertid].vertex
                            [0]=tetmesh->
                            getVertexAt(
                            vertindex[0]).
                            getX();
```

```
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getNormalY();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [2]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getNormalZ();
m->m_objectArray[id
    ].m_faces[faceid
    ]=vertid;

m->m_objectArray[id
    ].m_vertices[
    vertid].texture
    [0] =
          m->
              m_objectArray
              [id].
              m_vertices
              [vertid].
```

```
                texture
                [1] = 0.0
                f;
 vertid++;   faceid++;
// vertex 2
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getX();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getNormalY();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [2]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getNormalZ();
```

```
m->m_objectArray[id
    ].m_faces[faceid
    ]=vertid;

m->m_objectArray[id
    ].m_vertices[
    vertid].texture
    [0] =
          m->
              m_objectArray
              [id].
              m_vertices
              [vertid].
              texture
              [1] = 0.0
              f;
vertid++; faceid++;
// vertex 1
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
    vertindex[1]).
    getX();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[1]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[1]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[1]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
```

```
                vertid].normal
                [1]=tetmesh->
                getVertexAt(
                vertindex[1]).
                getNormalY();
            m->m_objectArray[id
                ].m_vertices[
                vertid].normal
                [2]=tetmesh->
                getVertexAt(
                vertindex[1]).
                getNormalZ();
            m->m_objectArray[id
                ].m_faces[faceid
                ]=vertid;

            m->m_objectArray[id
                ].m_vertices[
                vertid].texture
                [0] =
                    m->
                        m_objectArray
                        [id].
                        m_vertices
                        [vertid].
                        texture
                        [1] = 0.0
                        f;
            vertid++; faceid++;
    }
    if(tetrahe->renderFaceAt(1))
    {
            int *vertindex=
                tetrahe->
                getVertex();
            // vertex 0
            m->m_objectArray[id
                ].m_vertices[
                vertid].vertex
                [0]=tetmesh->
                getVertexAt(
                vertindex[0]).
                getX();
            m->m_objectArray[id
                ].m_vertices[
                vertid].vertex
                [1]=tetmesh->
                getVertexAt(
```

```
        vertindex[0]).
        getY();
    m->m_objectArray[id
        ].m_vertices[
        vertid].vertex
        [2]=tetmesh->
        getVertexAt(
        vertindex[0]).
        getZ();
    m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [0]=tetmesh->
        getVertexAt(
        vertindex[0]).
        getNormalX();
    m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [1]=tetmesh->
        getVertexAt(
        vertindex[0]).
        getNormalY();
    m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [2]=tetmesh->
        getVertexAt(
        vertindex[0]).
        getNormalZ();
    m->m_objectArray[id
        ].m_faces[faceid
        ]=vertid;

    m->m_objectArray[id
        ].m_vertices[
        vertid].texture
        [0] =
              m->
                  m_objectArray
                  [id].
                  m_vertices
                  [vertid].
                  texture
                  [1] = 0.0
                  f;
    vertid++;   faceid++;
    // vertex 3
```

```
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getX();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalY();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [2]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalZ();
m->m_objectArray[id
    ].m_faces[faceid
    ]=vertid;

m->m_objectArray[id
    ].m_vertices[
```

```
    vertid].texture
    [0] =
          m->
              m_objectArray
              [id].
              m_vertices
              [vertid].
              texture
              [1] = 0.0
              f;
vertid++;   faceid++;
// vertex 2
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getX();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[2]).
    getNormalY();
```

```
                        m->m_objectArray[id
                            ].m_vertices[
                            vertid].normal
                            [2]=tetmesh->
                            getVertexAt(
                            vertindex[2]).
                            getNormalZ();
                        m->m_objectArray[id
                            ].m_faces[faceid
                            ]=vertid;

                        m->m_objectArray[id
                            ].m_vertices[
                            vertid].texture
                            [0] =
                                m->
                                    m_objectArray
                                    [id].
                                    m_vertices
                                    [vertid].
                                    texture
                                    [1] = 0.0
                                    f;
                    vertid++; faceid++;
            }
            if(tetrahe->renderFaceAt(2))
            {
                    int *vertindex=
                        tetrahe->
                        getVertex();
                    // vertex 0
                    m->m_objectArray[id
                        ].m_vertices[
                        vertid].vertex
                        [0]=tetmesh->
                        getVertexAt(
                        vertindex[0]).
                        getX();
                    m->m_objectArray[id
                        ].m_vertices[
                        vertid].vertex
                        [1]=tetmesh->
                        getVertexAt(
                        vertindex[0]).
                        getY();
                    m->m_objectArray[id
                        ].m_vertices[
                        vertid].vertex
```

```
    [2]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getNormalY();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [2]=tetmesh->
    getVertexAt(
    vertindex[0]).
    getNormalZ();
m->m_objectArray[id
    ].m_faces[faceid
    ]=vertid;

m->m_objectArray[id
    ].m_vertices[
    vertid].texture
    [0] =
          m->
              m_objectArray
              [id].
              m_vertices
              [vertid].
              texture
              [1] = 0.0
              f;
vertid++;   faceid++;
// vertex 1
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
```

```
        vertindex[1]).
        getX();
m->m_objectArray[id
        ].m_vertices[
        vertid].vertex
        [1]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getY();
m->m_objectArray[id
        ].m_vertices[
        vertid].vertex
        [2]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getZ();
m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [0]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getNormalX();
m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [1]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getNormalY();
m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [2]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getNormalZ();
m->m_objectArray[id
        ].m_faces[faceid
        ]=vertid;

m->m_objectArray[id
        ].m_vertices[
        vertid].texture
        [0] =
              m->
                  m_objectArray
                  [id].
```

```
                m_vertices
                [vertid].
                texture
                [1] = 0.0
                f;
 vertid++;   faceid++;
// vertex 2
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getX();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalY();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [2]=tetmesh->
    getVertexAt(
    vertindex[3]).
```

```
                    getNormalZ();
                m->m_objectArray[id
                    ].m_faces[faceid
                    ]=vertid;

                m->m_objectArray[id
                    ].m_vertices[
                    vertid].texture
                    [0] =
                        m->
                            m_objectArray
                            [id].
                            m_vertices
                            [vertid].
                            texture
                            [1] = 0.0
                            f;
                vertid++; faceid++;
        }
        if(tetrahe->renderFaceAt(3))
        {
                int *vertindex=
                    tetrahe->
                    getVertex();
                // vertex 1
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].vertex
                    [0]=tetmesh->
                    getVertexAt(
                    vertindex[1]).
                    getX();
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].vertex
                    [1]=tetmesh->
                    getVertexAt(
                    vertindex[1]).
                    getY();
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].vertex
                    [2]=tetmesh->
                    getVertexAt(
                    vertindex[1]).
                    getZ();
                m->m_objectArray[id
                    ].m_vertices[
```

```
        vertid].normal
        [0]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getNormalX();
    m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [1]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getNormalY();
    m->m_objectArray[id
        ].m_vertices[
        vertid].normal
        [2]=tetmesh->
        getVertexAt(
        vertindex[1]).
        getNormalZ();
    m->m_objectArray[id
        ].m_faces[faceid
        ]=vertid;

    m->m_objectArray[id
        ].m_vertices[
        vertid].texture
        [0] =
              m->
                  m_objectArray
                  [id].
                  m_vertices
                  [vertid].
                  texture
                  [1] = 0.0
                  f;
     vertid++;   faceid++;
    // vertex 2
    m->m_objectArray[id
        ].m_vertices[
        vertid].vertex
        [0]=tetmesh->
        getVertexAt(
        vertindex[2]).
        getX();
    m->m_objectArray[id
        ].m_vertices[
        vertid].vertex
        [1]=tetmesh->
```

```
                    getVertexAt(
                    vertindex[2]).
                    getY();
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].vertex
                    [2]=tetmesh->
                    getVertexAt(
                    vertindex[2]).
                    getZ();
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].normal
                    [0]=tetmesh->
                    getVertexAt(
                    vertindex[2]).
                    getNormalX();
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].normal
                    [1]=tetmesh->
                    getVertexAt(
                    vertindex[2]).
                    getNormalY();
                m->m_objectArray[id
                    ].m_vertices[
                    vertid].normal
                    [2]=tetmesh->
                    getVertexAt(
                    vertindex[2]).
                    getNormalZ();
                m->m_objectArray[id
                    ].m_faces[faceid
                    ]=vertid;

                m->m_objectArray[id
                    ].m_vertices[
                    vertid].texture
                    [0] =
                          m->
                              m_objectArray
                              [id].
                              m_vertices
                              [vertid].
                              texture
                              [1] = 0.0
                              f;
                vertid++;   faceid++;
```

```
// vertex 3
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [0]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getX();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [1]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getY();
m->m_objectArray[id
    ].m_vertices[
    vertid].vertex
    [2]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getZ();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [0]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalX();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [1]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalY();
m->m_objectArray[id
    ].m_vertices[
    vertid].normal
    [2]=tetmesh->
    getVertexAt(
    vertindex[3]).
    getNormalZ();
m->m_objectArray[id
    ].m_faces[faceid
    ]=vertid;
```

```
                                                    m->m_objectArray[id
                                                        ].m_vertices[
                                                        vertid].texture
                                                        [0] =
                                                             m->
                                                                 m_objectArray
                                                                 [id].
                                                                 m_vertices
                                                                 [vertid].
                                                                 texture
                                                                 [1] = 0.0
                                                                 f;
                                                    vertid++; faceid++;
                                                }
                                            }
                                        }
                                    }
                                    m->Reload();
                                }
                                return m;
}
```