

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

**The Use of Tensegrity to Simulate
Diaphragm Motion Through
Muscle and Rib Kinematics**

Author:

Wesley BOURNE

Supervisors:

Dr. Fernando BELLO

Dr Pierre-Frédéric VILLARD

THIRD YEAR BENG PROJECT

Project Directory: <http://www.doc.ic.ac.uk/~wlb05/project/>

Abstract

Respiration is a complex process involving the interaction of several muscles and other tissue. The diaphragm has a large influence as it is the central division within the abdomen and plays a role in inflating and deflating the lungs. It is composed of different kinds of tissue which need to be modelled using different paradigms. Tensegrity is the use of rigid and elastic links to give more rigidity to systems and will be used to model the more rigid parts of the diaphragm. A mass spring will be used for the remainder. Rib kinematics and attached tissues will also be modelled to provide a more realistic simulation. The result is a Java simulation running in real time with parameters to control breathing speed, force, rib motion and mass spring parameters. The results can be exported and an evaluation of the simulator will be provided.

Acknowledgements

- My thanks go to Fernando Bello and Pierre-Frédéric Villard for their consistent, fast and constructive advice throughout the project.
- I would also like to thank Philip Edwards for agreeing to be my second marker.
- Finally, I would like to thank my parents for giving me the opportunity to study at Imperial College.

Contents

1	Introduction	5
1.1	Motivation and Intent	5
1.2	Goals	5
1.3	Salient Points	5
1.4	Contributions	6
2	Background	7
2.1	Medical Simulation	7
2.1.1	Requirements	7
2.1.2	3D Imaging Techniques	8
2.1.3	Sample Current Applications	10
2.2	Mass Spring Systems	10
2.2.1	Equation of Dynamics	11
2.2.2	Explicit Euler Solver	12
2.2.3	Choice of δt	13
2.3	Finite Element Method	14
2.4	Tensegrity	16
2.4.1	Definition and Origins	16
2.4.2	Method	17
2.4.3	Applying Tensegrity Constraints	17
2.4.4	Potential Application to Tissue Simulation	18
2.5	Diaphragm Modelling	19
2.5.1	Anatomy and Physiology	20
2.5.2	Diaphragm Composition	21
3	Biomechanical Modelling	22
3.1	Diaphragm Mechanics and Tensegrity	22
3.2	Rigid Connections	24
3.3	Fixed Points	26
3.4	Action Lines	28
3.5	Rib Kinematics	29

4	Implementation	30
4.1	SOFA Solution	30
4.1.1	Background	30
4.1.2	Diaphragm Motion in SOFA	31
4.1.3	Issues with Heterogenity	33
4.2	Java Solution	34
4.2.1	Basic Class Structure	34
4.2.2	2D Display	35
4.2.3	3D Display	36
4.3	The Final Product	38
5	Evaluation	41
5.1	Tensegrity	41
5.1.1	Effect on Diaphragm	43
5.2	Diaphragm Motion	45
5.2.1	The Patient Model	46
5.3	Simulator	46
5.3.1	Cube Simulation	47
5.3.2	Diaphragm Simulation	48
5.3.3	Influence of Mesh Size with a Heterogeneous Model	49
5.3.4	Influence of Mesh Size with a Homogenous Model	50
5.3.5	Conclusions	52
6	Conclusion	55
6.1	Discussion	55
6.1.1	Work Completed	55
6.1.2	Project Outcomes	55
6.1.3	Limitations	55
6.2	Future work	56
6.2.1	Integration with SOFA	56
6.2.2	Educational Context	56
6.2.3	Haptic Integration	56
	Bibliography	57
A	Appendix	59
A.1	User Guidance	59
A.1.1	Adding a new object	59
A.1.2	Action Line Fall Off Modication	62
A.1.3	Euler Solver Implementation	62

Chapter 1

Introduction

1.1 Motivation and Intent

The diaphragm plays a key role in the respiration of humans and its complex motion can have an impact on interventional or therapeutic procedures. It is mostly muscular and divides the torso, connected to the spine and ribs, with holes through which large blood vessels pass into the lower abdomen. The position of the diaphragm and its deformation are influenced by rib motion combined with muscle action. This project aims to model the motion and deformation of the diaphragm together with the surrounding organs and bones, using a combination of mass spring systems and tensegrity. Chapter 2 describes these methods together with the powerful and widely used finite element method. Due to the complicated interactions between neighbouring tissues, assumptions had to be made and heuristic algorithms applied. See chapter 3 for a more in depth discussion of this aspect of the project.

1.2 Goals

The goal of the project is to create a simulation of diaphragm motion using tensegrity to model the more rigid upper regions. The simulation needs to be real-time and tuneable via a control panel to allow it to easily match the breathing motion obtained from patient scans.

1.3 Salient Points

In order to understand the validation section of the report, it is important to have a basic understanding of the process whereby CT scans are filtered, segmented and meshed. This is illustrated at the beginning of chapter 2. Another important aspect of the project is the inability of the SOFA framework to process two tissue simulation methods on the same model. This resulted

in having to implement the simulation outside the SOFA framework as originally planned, using Java and OpenGL instead. The full discussion of this problem is presented in section 4.1.3.

Finally, the simulator (figure 1.1) was complete and an evaluation on the effect of tensegrity is shown in chapter 5.

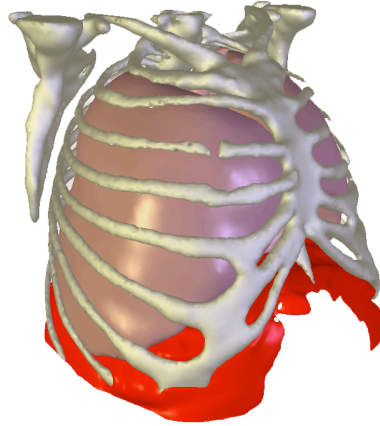


Figure 1.1: The completed Java simulator with the ribs, diaphragm, lungs sternum and ligaments visible.

1.4 Contributions

- *Tuneable Simulator*: The final product is a simulator with parameters to change the speed, magnitude of forces and other factors contributing to the diaphragm motion. This allows the simulator to match diaphragmatic or chest breathing and any mix in between.
- *Customisable to Patients*: The simulator has been tested on three patients and performs realistically thanks to the tuneable parameters.
- *Extensible Framework*: The project is extendable and object-orientated and is amenable to future work by other researchers or intern students.
- *Study and Analysis of Computational Cost of Tensegrity*: The cost of solving simulations involving rigid links has been studied and evaluated.
- *Study into Applicability of Tensegrity*: Tensegrity has been shown to have a supportive effect as a part of a tissue simulator.
- *Algorithms for Diaphragm Initiation and Muscle Motion*: Intuitive algorithms have been created that link opposing sides of tissue topologies and simulate the attachment of tissue to other models and apply muscle forces.

Chapter 2

Background

The aim of this background section is to make this document self-contained and also to give a general understanding of the field. The discussions here can be expanded more fully in the work of Meier et al[ML] and Vidal et al[VB]. The section will begin with a brief summary of the current field of medical simulation before moving to simulating tissue and ending with the physiology of the diaphragm. The description of the kinematic equations is based on the work of Baraff and Witkin[BW] and uses the syntax of Bhasin and Liu[BL],

2.1 Medical Simulation

Medicine has always been a great follower of technology and the growth in possibilities and computational power have been closely followed by those wanting to use these advances for medical purposes. Being such a massive field, there are numerous applications including medical diagnosis, training, planning and more. Another contributing factor to the rapid growth of simulation is the wider availability of data from CT, MRI and other 3D forms of medical information. Computers are excellent at summarising, manipulating and finding patterns in large datasets. Their input to medical procedures can prove invaluable.

2.1.1 Requirements

In such a high-pressure and demanding environment, any simulation must reach a level of realism and usability. The simulation must be:

- *Validated*: Experts in the field must verify the validity of the simulation as a useful tool.
- *Realistic*: There is no need for a simulation that varies so much from the truth that it may give an inconsistent or false impression of reality and put the patient in danger.

- *Affordable*: Hospital budgets must take into account which investments will give the greatest benefit to the overall population. A very expensive simulation will deter investment.

2.1.2 3D Imaging Techniques

Filtering

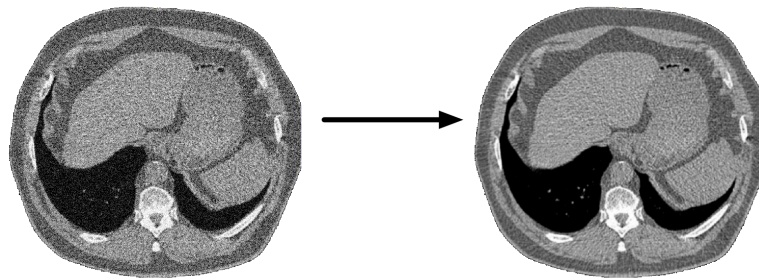


Figure 2.1: The start and end points of filtering of a diaphragm.

Information entered into a computer via a scanner will most likely contain noise. Filtering is the process by which this noise is removed by smoothing or blurring each pixel based on a combination of its neighbours. This is visible in figure 2.1. In medical simulations, this can remove crucial detail about anatomical boundaries. A more involved technique using anisotropic diffusion aims to iterate towards an equilibrium state controlled by a partial differential equation. It has proved successful and is a popular choice to improve image quality.

Segmentation

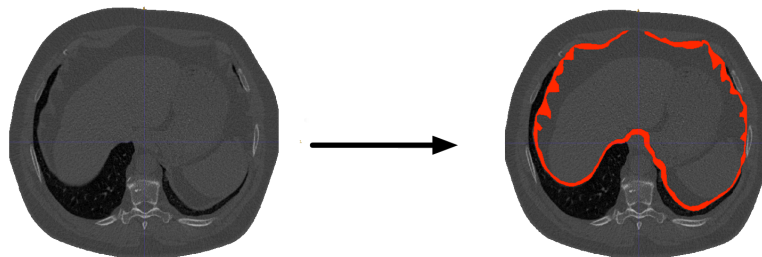


Figure 2.2: The start and end points of segmentation of a diaphragm. A sequence of slices from a 3D scan is turned into labelled voxels.

Once an image has been filtered to remove any noise, it is desirable to be able to label the different anatomical areas of the image (figure 2.2). This is

a time-consuming process and is reliant on the user. Techniques include:

- *Thresholding*: A value is chosen to be a threshold and the image is partitioned based on those that fall on either side of this boundary.
- *Region Growing*: A centre point is chosen as a seed and this region is grown up to a boundary.
- *Watershed transform*: An intuitively simple algorithm which divides the image based on basins and hills as if water were flooded into the scene.
- *Livewire*: A user's cursor is tracked and the closest approximated boundary is selected before being approved by the user who can then proceed to the next point.

Meshing

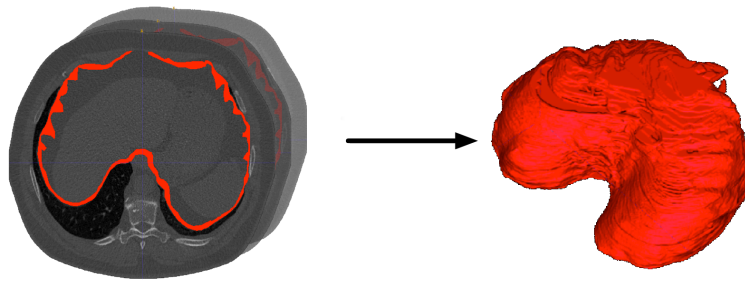


Figure 2.3: The meshing process begins with multiple slices of segmented data and proceeds to a 3D model.

After the slices of a scan have been segmented into the different organs, a meshing algorithm can be used to create a mesh (figure 2.3). Here are some popular methods:

- *Delaunay*: In 2D, for a set of points P , the Delaunay triangulation maximises the minimum angle in the triangles created, thus preventing the creation of sliver triangles. The method is more complicated but intuitively similar in 3D.
- *Marching Cubes*: Is the 3D equivalent of the marching squares algorithm. The algorithm traverses the scalar field considering eight neighbouring positions at a time. It determines the polygons needed to represent this part of the isosurface passing through this cube. These polygons are then fused to create the surface.

2.1.3 Sample Current Applications

As a result of the funding and interest applied to medicine there are a plethora of interesting and advanced applications. A brief description of examples are shown here.

Augmented Reality

Computer-generated artefacts are added to a scene in the real world. The user typically has a head-mounted display. The display can aid in the differentiation of tissue or help guide an instrument. The additions made via the computer must be correctly orientated with the position of the user. Performing this synchronisation can prove to be very difficult; without near perfect alignment the slightest discrepancies can become a hindrance.

Virtual Endoscopy

In a traditional video endoscopy, an endoscope is inserted into the patient with a small opening in the body. This is an invasive procedure and only has limited viewing ability. A virtual endoscopy creates a simulation in which a virtual endoscope can be manoeuvred through the patient in a 3D environment created using a medical scan of the patient. It allows for the interactive exploration of the tissues and removes the need for an invasive procedure.

Patient Specific Virtual Environments

Using prior data from the patient a simulation can be tailored to an exact specification. This means the surgeon can practise a procedure in an environment as near to reality as possible. This possibility allows the surgeon to be more confident and prepare for any technically challenging elements during the operation.

2.2 Mass Spring Systems

Once it has been decided that a medical simulation is needed, there is a requirement for accurate and real-time modelling of tissue. Popular methods to simulate tissue are presented in the following section. Mass spring, Finite element and Tensegrity methods will be discussed.

Mass spring systems are a popular and effective way of simulating a variety of physical interactions. A set of masses is created with predefined links. A subsection of a simple mass spring system is shown in figure 2.4. These links have stiffness values and they exert forces on the masses which in turn move. Iteratively solving and analysing the motion of the masses can be used to simulate objects as varied as cloth or human organs. A standard

method of defining the forces and solving the motion is discussed in the following section.

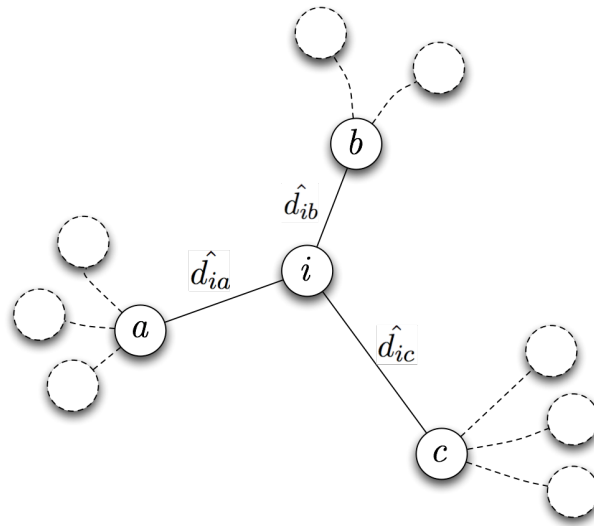


Figure 2.4: A 2D mass spring system, mass i has three neighbours which must be included in its force calculation.

Advantages	Disadvantages
<p><i>Quick Visualisation:</i> Simplicity between the mesh and the simulation means it is quick to implement.</p> <p><i>Simulation Speed:</i> Equations of motion are fast and efficient to solve.</p> <p><i>Mature Method:</i> There has been a substantial amount of research backing the method and although it isn't perfect, it performs well in many situations.</p>	<p><i>Reliance on Mesh:</i> A mesh that looks identical to another may perform very differently as a result of different low-level connections and constructs.</p> <p><i>Speed of Propagation:</i> Forces can only propagate one neighbour at a time. This removes from the realism.</p> <p><i>Tendency to Oscillate:</i> The simulation can get into a state where masses oscillate. This is discussed further in section 2.2.3.</p>

2.2.1 Equation of Dynamics

Equation 2.1 forms the basis of the forces that are calculated within the system. It is computed for each mass at the beginning of each timestep before being used along with the current position and velocity to numerically

integrate to find the position at the next timestep.

The equation indicates that the resultant force felt by a mass is the sum of all the forces of tensile links and any outside force minus any friction force.

$$f_i^t = m_i a_i^t = \left(\sum_{j \in N(i)} k_{ij} \hat{d}_{ij} (l_{ij}^t - l_{ij}^0) \right) + f_e^t - \gamma v_i^t \quad (2.1)$$

Where:

f_i^t is the force placed on m_i at time t .

m_i is a mass i .

a_i^t is the acceleration of a mass i at time t .

$N(i)$ is the set of neighbouring masses.

\hat{d}_{ij} is the distance between m_i and m_j .

l_{ij}^t is the distance between m_i and m_j at time t .

l_{ij}^0 is the distance between m_i and m_j at rest.

f_e^t is an external force on the mass at time t , such as gravity.

γ is the friction coefficient.

v_i^t is the velocity of m_i at time t .

Manipulating equation 2.1 yields the following:

$$a_i^t = \frac{\left(\left(\sum_{j \in N(i)} k_{ij} \hat{d}_{ij} (l_{ij}^t - l_{ij}^0) \right) - \gamma v_i^t + f_e^t \right)}{m_i} \quad (2.2)$$

The Continuous Solution

$$f = ma \quad (2.3)$$

$$v_i^t = \int_0^t f(a_i) da. \quad (2.4)$$

$$x_i^t = \int_0^t f(v_i) dv. \quad (2.5)$$

These continuous equations become discretised and are solved using the Euler method. They describe standard Newtonian physics where acceleration, velocity and position are interrelated via integrals and derivatives. The simulation uses a discrete timestep of milliseconds and for this reason these equations must be approximated.

2.2.2 Explicit Euler Solver

The Discrete Approximation

Once all the forces have been calculated using the force equation presented in equation 2.1, the analytic equations 2.4 and 2.5 must be discretised. The

chosen solution for the implementation was an explicit Euler solver. The Euler method uses the following intuitive approximations.

$$v_i^{t+\delta t} = v_i^t + a_i^t \delta t \quad (2.6)$$

$$x_i^{t+\delta t} = x_i^t + v_i^{t+\delta t} \delta t \quad (2.7)$$

These equations are easy to solve and lend themselves well to scaling meaning that real-time simulation is possible.

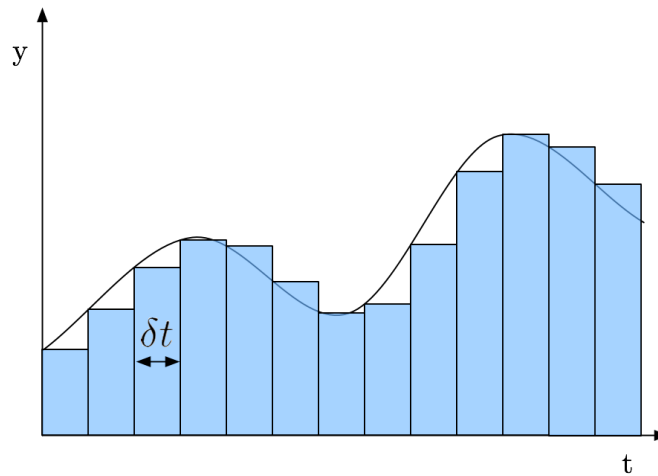


Figure 2.5: The Euler Integration method shown graphically.

2.2.3 Choice of δt

After considering figure 2.5, it becomes apparent that the correct choice of δt makes a massive difference in the convergence of any solution. Some of the possible convergent cases are shown in figure 2.6. Choosing too small a value leads to a case where there may be little movement and friction may counteract any motion towards a true solution. Choosing a value too large may lead to the simulation ‘exploding’. Both are very undesirable and this issue leads to a great deal of time in mass spring systems being spent on tuning parameters to arrive at a satisfactory tradeoff between speed of solution and stability.

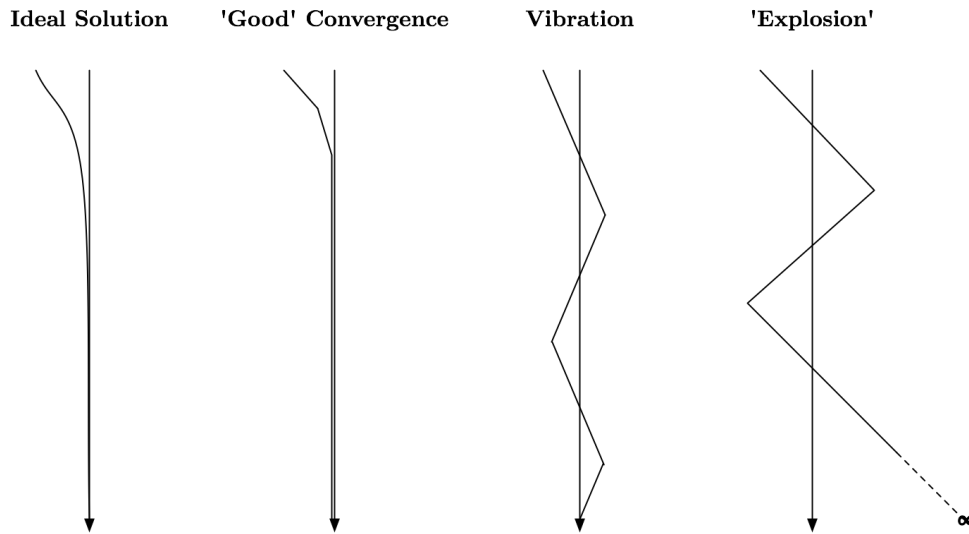


Figure 2.6: Different convergent and divergent cases.

2.3 Finite Element Method

Rather than basing the deformation model on a discontinuous approach, an alternative has been proposed which is based on the law of continuum mechanics computed in a continuous medium. Once assumptions have been made about the elasticity and the lack of internal forces, the following second order Navies[ML].

$$(\lambda + \mu) \text{grad}(\text{div } \mathbf{u}) + \mu \Delta \mathbf{u} = \mathbf{0} \quad (2.8)$$

Where:

λ and μ are the Lamè constants of the material.

\mathbf{u} is the displacement vector of any point of the object with respect to its initial position.

grad is the gradient function: $\text{grad}(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$

div is the divergence function: $\text{div}(x, y, z) = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} + \frac{\partial f}{\partial z}$

An analytic solution to this equation does not exist and therefore a numerical solution is required. The FEM method aims to solve this equation by subdividing the object into a discrete number of elements. Once this has been completed, the displacement of each one of these elements is approximated by a polynomial equation which is related to control nodes.

A system of equations arises:

$$K\mathbf{U} = \mathbf{F} \quad (2.9)$$

Where:

K is the symmetric and sparse stiffness matrix.

\mathbf{U} is the nodal displacement vector.

\mathbf{F} is the force vector.

Solving this equation yields the displacement of each node at that iteration.

Advantages	Disadvantages
<i>Based on Physics:</i> Unlike mass spring systems, the quality of the simulation doesn't depend on the mesh.	<i>Speed:</i> The method is much more computationally expensive. Real time simulation is only possible for a relatively small number of elements. <i>Hard to program:</i> The method is more involved than the mass spring system and is much harder to implement. Most people use a commercial FEM package, but these are usually not real time.

2.4 Tensegrity

2.4.1 Definition and Origins

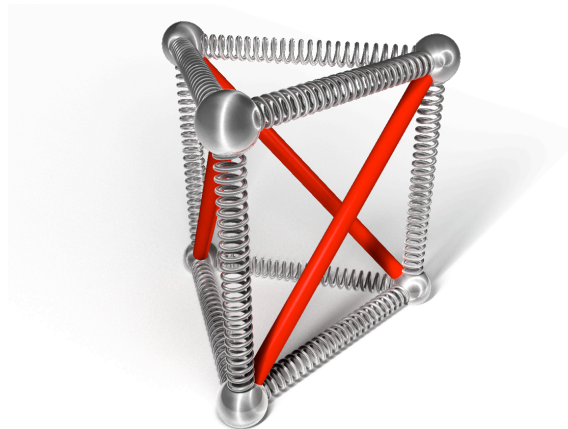


Figure 2.7: A simple figure demonstrating tensegrity. The solid bars represent rigid links.

Tensegrity is a portmanteau of tensional integrity. This arises from its use of both rigid and elastic links in order to form a stable structure (figure 2.7). In

1927 architect R. Buckminster Fuller investigated its uses in buildings[IN]. Later, artist Kenneth Snelson created structures involving tensegrity. His structure is shown in figure 2.8.

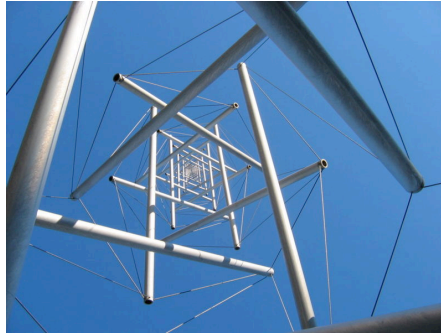


Figure 2.8: One of Snelson's tensegrity structures.

2.4.2 Method

The presence of elastic links adds a direct correspondence with the workings of mass spring systems. For this reason, mass spring systems (section 2.2) are usually the starting point in developing tensegrity. Constraining the simulation to follow the properties of the rigid links is the main challenge.

2.4.3 Applying Tensegrity Constraints

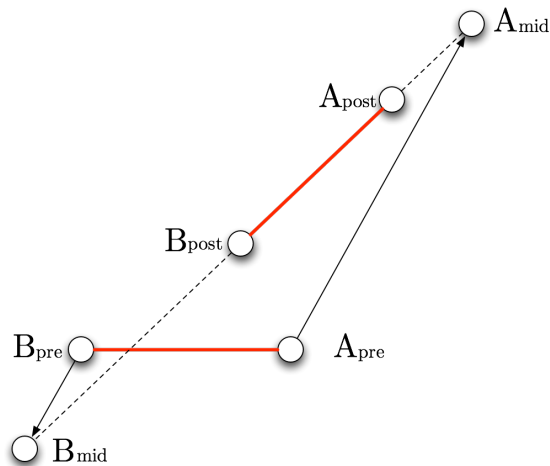


Figure 2.9: A diagram to help explain the solution to enforcing the tensegrity constraint. A_{pre}, B_{pre} signify the position before motion. A_{mid}, B_{mid} are the locations ignoring the length constraint. A_{post}, B_{post} are the final locations after applying the constraint.

Once tensile and rigid connections have been created, an algorithm is required to ensure the length of the rigid connections are respected throughout the kinematic motion. A solution was found in the work of Alexis Guillaume [GU]. It proceeds by calculating the location of each mass ignoring all rigid connections and updating their locations. Post-processing is then used to modify the locations of the masses that have rigid connections.

$$\frac{A_{pre}A_{mid}}{B_{pre}B_{mid}} = \frac{A_{mid}A_{post}}{B_{mid}B_{post}} \quad (2.10)$$

$$A_{pre}B_{pre} = A_{post}B_{post} \quad (2.11)$$

These equations along with figure 2.9 are sufficient to define the new position of A and B unambiguously. Each rigid connection is then considered individually and constrained before the scene is redrawn. This method does not allow a mass to have multiple rigid connections, otherwise the dynamics of the motion would be much more complicated. This problem is mitigated by ensuring that the rigid connection algorithm does not create such a situation (a diagram of such a disallowed situation is shown in figure 2.10).

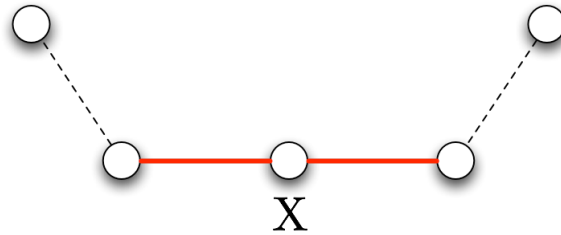


Figure 2.10: An undesirable situation with a single mass X having two rigid connections.

2.4.4 Potential Application to Tissue Simulation

The technique has been used to explain complex behaviour of materials in “viruses, nuclei, cells, tissues, and organs in animals as well as in insects and plants”[IN]. Its intuitive role to add strength to structures seems ideal for modelling muscle or ligaments. In fact, most physical movements in the human body can be explained via tensegrity. Walking is a combination of rigid links (bones) being manipulated via elastic (muscle and tendon) links to create movement. The main purpose of this report is to evaluate the potential use of tensegrity along with mass spring systems to model the movement of internal organs, specifically the diaphragm during respiration.

2.5 Diaphragm Modelling

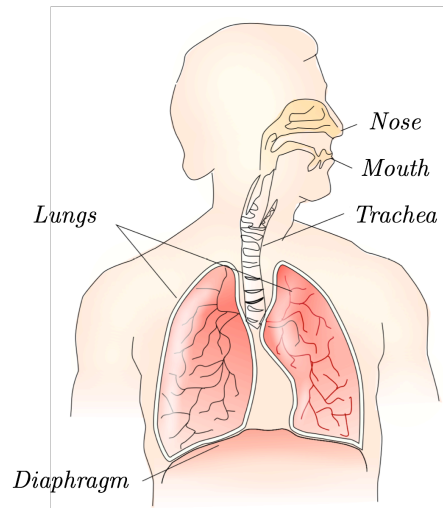


Figure 2.11: A human torso.

The thoracic-abdominal diaphragm (the diaphragm from this point onwards) is a fibromuscular sheet that sits below the ribcage and separates the thoracic cavity from the abdominal cavity. Its location in relation to the rest of the torso can be seen in figure 2.11. Its presence is crucial in respiration and breathing.

2.5.1 Anatomy and Physiology

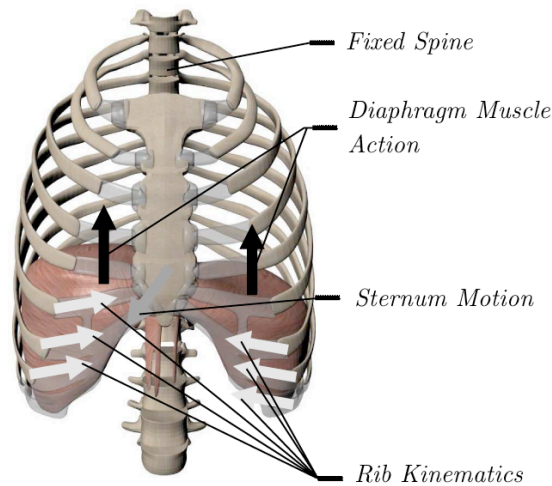


Figure 2.12: The arrows represent the motion of the diaphragm and ribs during exhalation.

The diaphragm attaches to the lower ribs, the sternum and the vertebrae that form the spine. It contains an opening through which blood vessels, such as the vena cava and aorta, nerves and the oesophagus pass.

When breathing in, the muscles of the diaphragm contract to flatten its dome-like shape. This causes a change in pressure in the upper cavity and air enters the lungs to compensate. When exhaling, the diaphragm compresses the upper cavity, forcing the air out. The forces exerted during this phase of breathing are shown in figure 2.12.

The diaphragm is composed of skeletal muscles and there are no smooth muscle fibres present. The diaphragm can be controlled voluntarily or unconsciously by the lower brain stem structures.

2.5.2 Diaphragm Composition



Figure 2.13: A diaphragm.

Figure 2.13 illustrates that the diaphragm is made up of distinct regions composed of different kinds of fibres. The upper section has a more tendonous quality. The central region is more elastic and, finally, where there are connections to ribs and the spine, a more rigid quality is present. For this reason, it would not be realistic to model the organ as one homogeneous material throughout.

Chapter 3

Biomechanical Modelling

Vital to any physical simulation is a solid understanding of the dynamics and a logical set of heuristic choices. This chapter assumes that a mass spring system has been created which enforces the constraints of tensegrity. This chapter will present the approximations and algorithms which have been implemented to match the motion of the diaphragm as closely as possible.

3.1 Diaphragm Mechanics and Tensegrity

As discussed in section 2.5.2, the nature of the diaphragm demands the use of different materials to simulate its motion accurately. Tensegrity seemed a pertinent match that would be worthy of investigation.

A decision was made that the upper tendinous section of the organ would be modelled using a combination of mass spring and tensegrity. The central region would be solely mass spring. Finally, the connections to the ribs and spine would be simulated using techniques described in ‘Fixed Points’, section 3.3. This division is visualised in figure 3.1 and the simulator’s resulting sections are in figure 3.2.

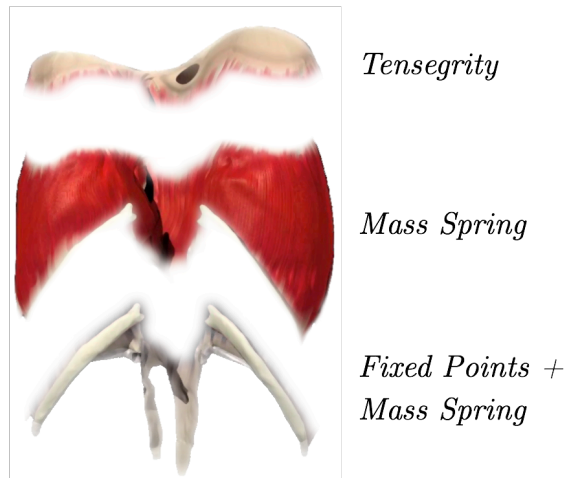


Figure 3.1: The regions of the diaphragm that will be simulated in different ways.

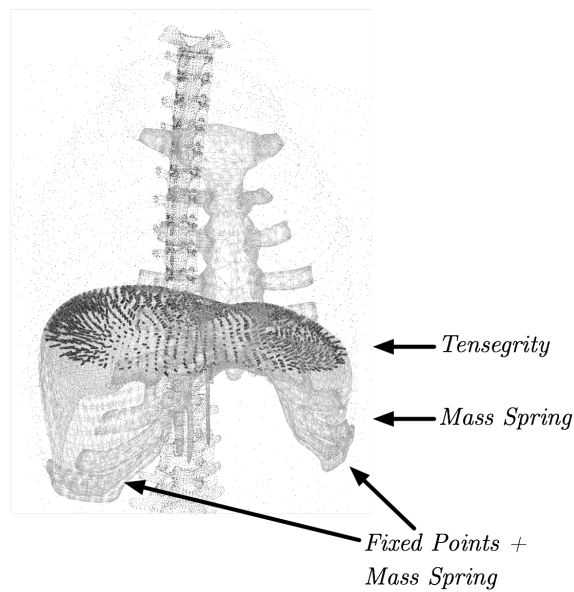


Figure 3.2: The resulting division shown in wireframe form. The rigid links are visible as the thicker lines.

As the 3D models consist of thousands or tens of thousands of points, it is important that this division can be calculated automatically. A plane defines the separation between the upper two regions using three carefully chosen points. The definition of the fixed points is achieved using a distance algorithm between the diaphragm and the rib models.

3.2 Rigid Connections

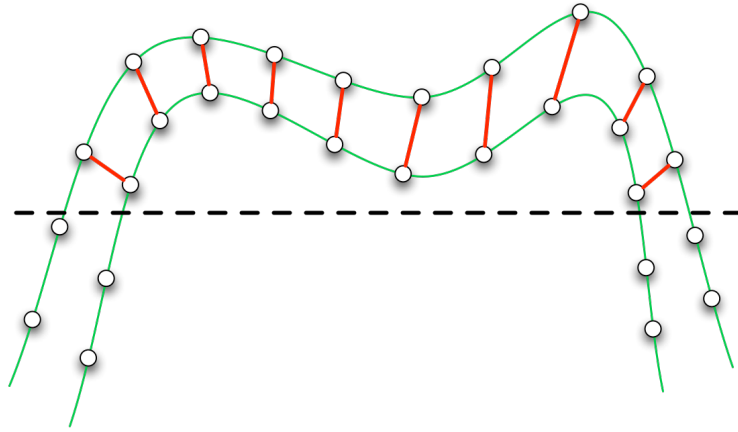


Figure 3.3: A cross-section of the desired result of the algorithm presented here.

Figure 3.3 shows two surfaces in close proximity with vertices represented by the circles. It would be ideal if the vertices directly opposite each other were connected via a rigid link, shown here by the thicker red line. The dotted line represents the separation between the tensegrity and tensile part of the organ. In the final version of the application this separation is defined by a simple plane.

The reasoning behind the heuristic algorithm is as follows. Once a suitable distance d has been chosen, a list of close masses is found. These masses could either be neighbouring masses on the same surface or ideally those on opposing surfaces. Testing the angle should remove those on the same surface, leaving a sorted list of masses on the opposing surface.

This algorithm is performed once upon the initialisation of the simulation.

```

1  foreach(Mass v) {
2    Assemble a sorted list of other vertices within distance  $d$ .
3  }
4  foreach(Mass v) {
5    foreach(Mass o in v.closet) {
6      Rem o from list if  $\theta$  between normal at surfaces of o and v is acute.
7    }
8  }
9  foreach(Mass v) {
10   while(v.closet is not empty & v has no rigid link) {
11     Pop o from v.closet
12     if (o has no rigidlink & both lie above defining plane) {
13       Create rigid link between o and v, break.
14     }
15   }
16 }

```

Figure 3.4: Pseudocode of the rigid link creation algorithm.

Shown in figure 3.5 are the results of applying the algorithm in figure 3.4. The connections between the layers of the diaphragm appear to match very closely the desired result described at the start of this section (figure 3.3). The two layers are heavily connected above the division between tensegrity and mass spring. The connections appear to be roughly at the opposite point, the aim of the algorithm.

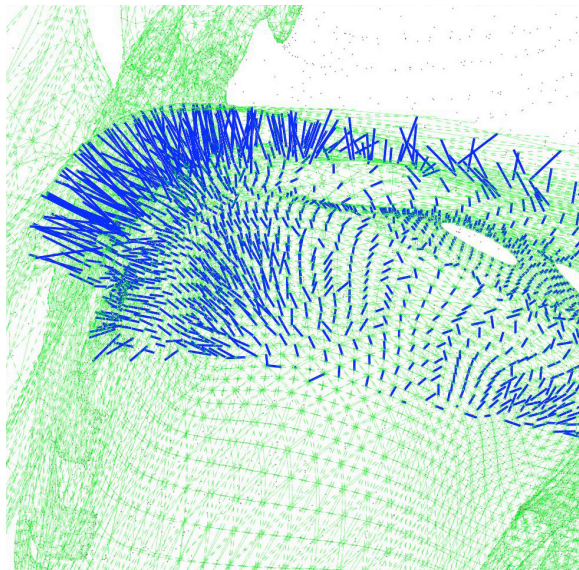


Figure 3.5: The connections created between the layers of the diaphragm.

3.3 Fixed Points

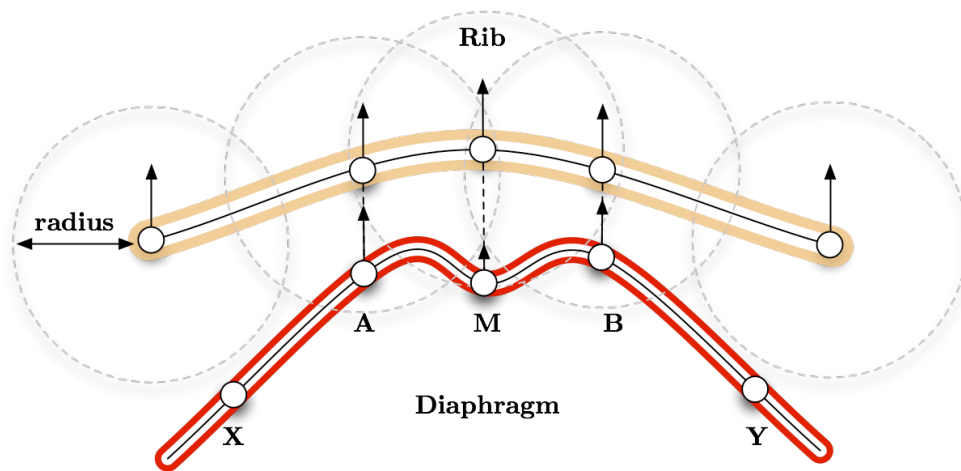


Figure 3.6: A diagram of how points on the diaphragm are transformed relative to rib motion. The length of the arrow is proportional to magnitude. The fall off is visible in the centre displaced mass, M, compared to A and B. X and Y are located out of range of all the rib masses regions' of influence.

At certain points the diaphragm is attached to the ribs. It is necessary to mimic this connection in order for the motion to be realistic. Another

heuristic method was devised to fit this situation. Figure 3.6 helps to explain how this is achieved. Firstly, pre-processing occurs in which each node in the diaphragm is checked for proximity to a rib within a cut off point. If there are multiple choices, the closest is chosen. Then, a fall-off factor is determined using $1 - \frac{d}{radius}$. At each iteration, the distance covered by each node in the ribs is calculated. A scaled value of this is applied to each attached node multiplied by the fall off. This gives the desirable result that close masses are influenced more and those further away will also move but with less impetus. The corresponding pseudocode for these two steps is shown below.

```

1  Initialisation
2  foreach(Mass v in ribs) {
3    foreach(Mass m in diaphragm) {
4      Distance  $d = \text{distance}(v,m)$ 
5      if ( $d < \text{radius}$ ) {
6        m.addToCloseList( $v, 1 - \frac{d}{radius}$ )
7      }
8    }
9  }
```

Figure 3.7: Pseudocode for initialisation of fixed points.

```

1  Update code
2  foreach(Mass m in ribs.stuckmasses) {
3    Mass closest = m.closestnode;
4    Vector v = closest.getChangeInLocation();
5    m.applyForce( $v * \text{scalingfator}$ );
6  }
```

Figure 3.8: Pseudocode run at each iteration to update the location of the fixed points.

3.4 Action Lines

Action lines are the simulation's equivalent to muscle motion. Success had been observed in the work of Nedel and Thalmann[NT] and an attempt to integrate a similar effect was made. Figure 3.9 gives an intuitive look at the approach adopted. In order to get an understanding of how the method works, it is best to imagine an infinitely long cylinder defined by a point, a direction vector and a falloff radius. Pre-processing then occurs to find all masses that fall within the fall off range of the cylinder, based on the closest [perpendicular] distance to the central vector (shown by d in the figure).

At each iteration, a scaled value of the vector defining the direction of the cylinder is applied to each mass after being multiplied by $\frac{\text{vertical distance}}{\text{radius}}$. This means that the further away the point is, the stronger it feels the pull. This is a heuristic judgement of fall off, but proved to be the most realistic of the various alternatives trialed.

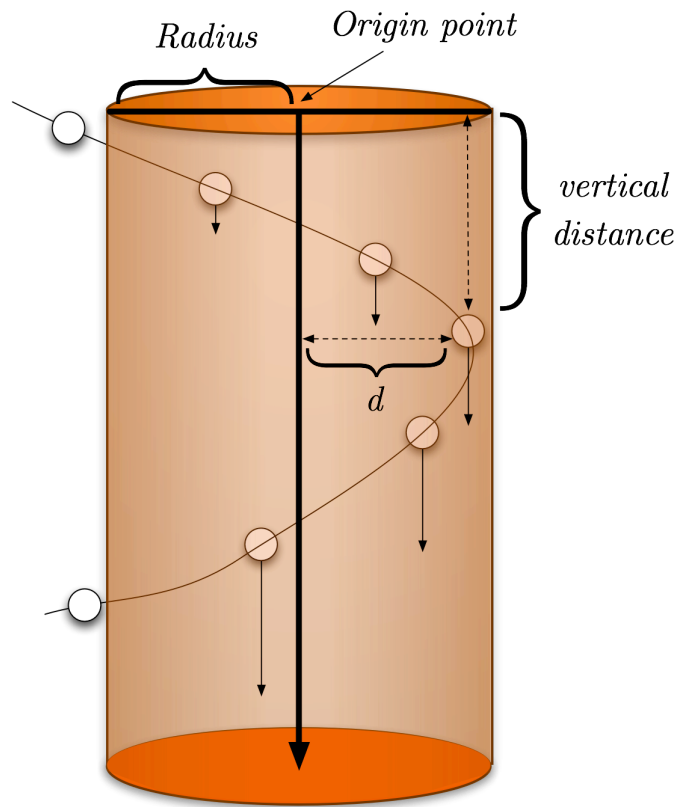


Figure 3.9: A diagram showing the basic workings of the action line method of imitating muscles.

3.5 Rib Kinematics

Another factor that plays a large role in respiration and the motion of the diaphragm is rib kinematics. Much of the work in this area was based on that presented in [DV]. Figure 3.10 shows a spine and three ribs and helps to explain how the rotations and displacements are calculated. In this project, existing C++ code implementing the rib kinematics was ported to Java.

The aim of the algorithm was to have a blend between vertical and lateral motion of the ribs. The lower ribs ‘open’ up more during deep breathing, while the upper ribs have a more directly upward translation.

For a more in depth discussion of this process look at the work of Didier *et al*[DV].

$$t_i^k = k(Ti) \quad (3.1)$$

$$a_i^k = k(Ai) \quad (3.2)$$

$$i \in [1, 20] \quad (3.3)$$

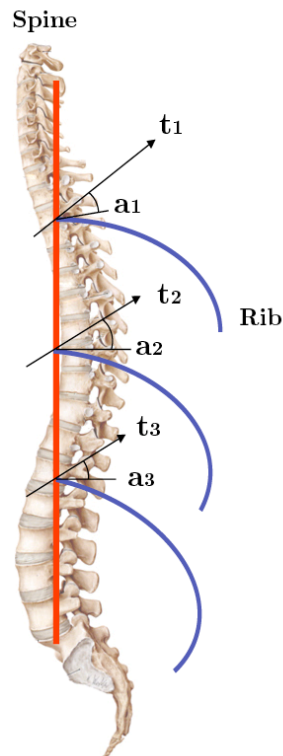


Figure 3.10: A diagram showing the layout of the variable in equations 3.1-3.3.

Chapter 4

Implementation

Once initial research and information gathering had been completed, the implementation stage of the project began with a focus on a SOFA based solution before migrating towards a custom Java solution for reasons to be discussed.

4.1 SOFA Solution

The initial goal was to create a SOFA simulation that allowed the integration of tensegrity as a forcefield. This would be highly desirable as SOFA is a framework designed for medical simulation. Its plug-and-play structure would make the code reusable even as SOFA was updated in the future. It was also envisaged that a control panel would be added to allow the tuning of parameters controlling breath length, force and rib motion.

4.1.1 Background

SOFA aims to simplify the simulation of physical and, more specifically, medical simulations, by allowing the user to focus on the mechanics and tuning of the simulation, rather than the implementation and routine algorithms for displaying, mapping and solving systems. SOFA is written in C++ and currently contains 250,000 lines of code written by twenty developers. The project benefited from the help of the CIMIT Sim Group, INRIA and ETH Zurich.

‘SOFA is an Open Source framework primarily targeted at real-time simulation, with an emphasis on medical simulation. It is mostly intended for the research community to help develop newer algorithms, but can also be used as an efficient prototyping tool. Based on an advanced software architecture, it allows to:

- Create complex and evolving simulations by combining new algorithms with algorithms already included in SOFA.
- Modify most parameters of the simulation – deformable behavior, surface representation, solver, constraints, collision algorithm, etc. – by simply editing an XML file.
- Build complex models from simpler ones using a scene-graph description.
- Efficiently simulate the dynamics of interacting objects using abstract equation solvers
- Reuse and easily compare a variety of available methods

SOFA is currently developed by 3 INRIA teams: Alcove, Evasion and Asclepios.’

The SOFA Architecture [SOFA]

4.1.2 Diaphragm Motion in SOFA

Development

The work began as an introduction to medical simulation and consisted mainly in experimentation with parameters and different models. It took time to become accustomed to features such as mapping between visual representations and the supporting mathematical simulation. This allows the visual representation to be highly complex while reading its displacements from a simplified underlying geometry. A diaphragm modelled by the finite element method is shown in figure 4.1. The associated XML file is shown in figure 4.2. The simulation uses a timestep of 0.02 seconds, uniform masses throughout. Each object is fixed by three points and a simple static solver is used.

Figure 4.1: A diaphragm in SOFA, the tree of the scenegraph is visible on the left.

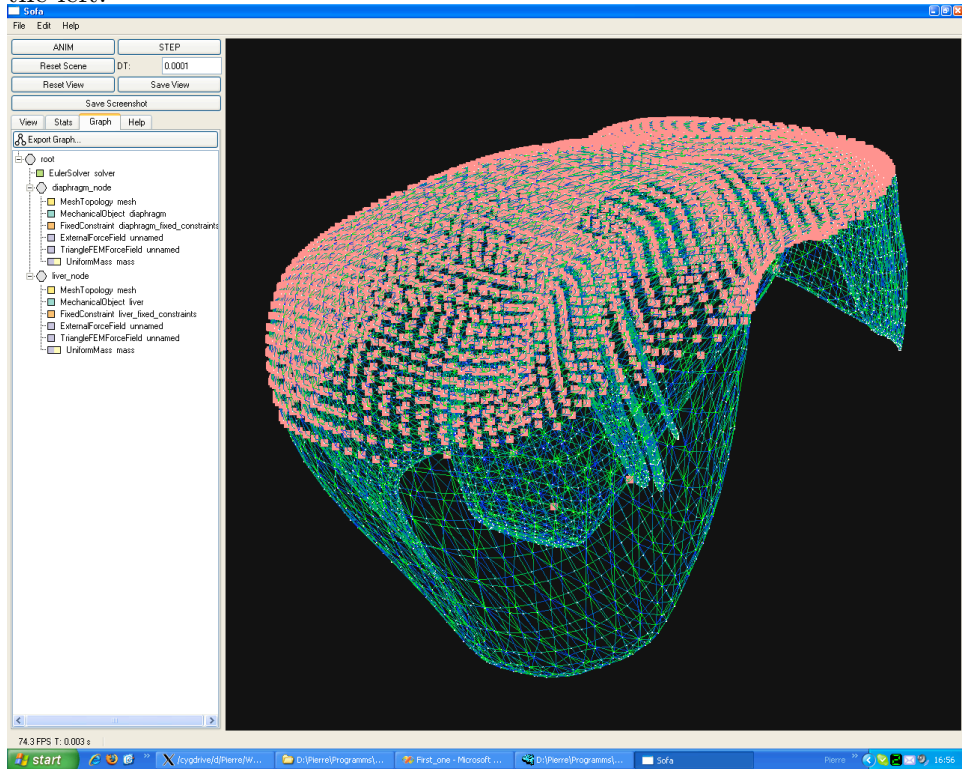


Figure 4.2: An XML scenegraph describing two objects, a liver and a diaphragm, both simulated using the finite element method.

```

<Node name="root" dt="0.02" showBehaviorModels="1" showCollisionModels="1"
  showMappings="0" showForceFields="1">
  <Object type="CollisionPipeline" verbose="0" />
  <Object type="BruteForceDetection" name="N2" />
  <Object type="CollisionResponse" response="default" />
  <Node>
    <Object type="StaticSolver" name="solver" iterations="25" />
    <Object type="MechanicalObject" name="diaphragm" />
    <Object type="UniformMass" name="mass" mass="0.1" />
    <Object type="Mesh" filename="Topology/diaphragm.msh"/>
    <Object type="Triangle"/>
    <Object type="TriangleFEMForceField" name="FEM" youngModulus="5000"
      poissonRatio="0.3" />
    <Object type="FixedConstraint" indices="3 39 64" />
  </Node>
  <Node>
    <Object type="StaticSolver" name="solver" iterations="25" />
    <Object type="MechanicalObject" name="liver" />
    <Object type="UniformMass" name="mass" mass="0.1" />
    <Object type="Mesh" filename="Topology/liver.msh" />
    <Object type="Triangle" />
    <Object type="TriangleFEMForceField" name="FEM" youngModulus="5000"
      poissonRatio="0.3" />
    <Object type="FixedConstraint" indices="3 39 64" />
  </Node>
</Node>

```

Results

The results that were shown by SOFA were acceptable (15-20 iterations per second) on a simple diaphragm model. It was during this initial testing that the issues discussed in 4.1.3 became apparent and forced a change in the development platform used in the project.

4.1.3 Issues with Heterogeneity

After research into the possibilities of combing two simulation methods, FEM and mass spring, an issue within SOFA arose. In the current implementation, mixing two force fields on one object is not possible. This was a serious problem as the aim was that parts of the diaphragm be modelled by

a simple mass-spring system and the rest with tensegrity. After consulting with the developers of SOFA, they deemed it very difficult to do this in the current codebase. It was indicated that such a mix and match approach may be available in future iterations, but no definitive time scale was given.

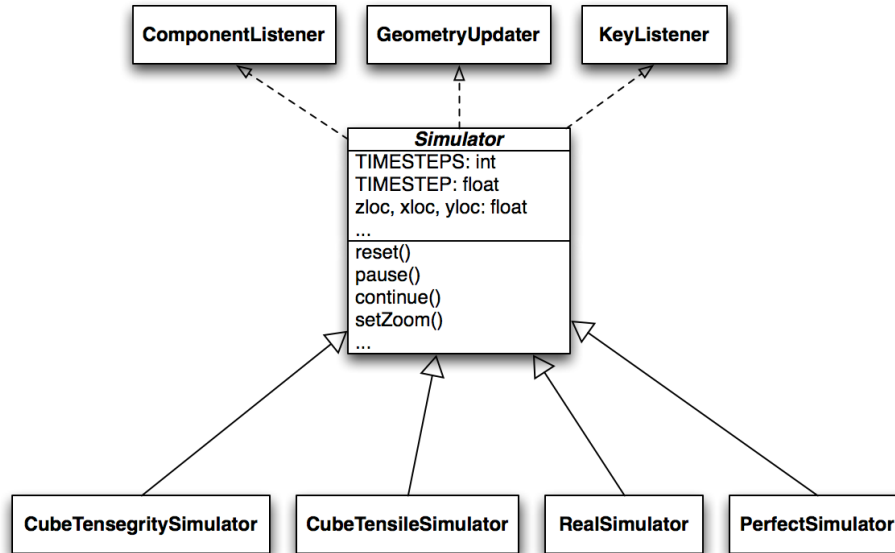
4.2 Java Solution

Forced to consider alternatives, the choice was then to proceed by making a simple 2D simulator. This simulator could then be used to study the viability of a 3D implementation. Java fit the requirements of flexibility, cross-platform nature, and prior familiarity and as such was chosen as the main language for the simulator. An iterative scheme fits well with the development plan of the project. It was first decided to concentrate on simple structures with few nodes in 2D before considering expansion into 3D. Once the transition to 3D was made, parsers of file formats, exporters and other options were added.

4.2.1 Basic Class Structure

Once development had begun, a modular design was necessary to keep flexibility for future additions to the design and functionality. The end result is a relatively concise, but productive set of classes to provide real-time interaction. There are packages to handle the importing and exporting of data, mathematical classes, model classes and those that aid in the animation and motion of the ribs. Another software-engineering decision was the creation of an abstract simulator class which is then extended upon for each individual situation. This is shown in figure 4.3.

Figure 4.3: A simplified UML diagram of the Simulator class hierarchy.



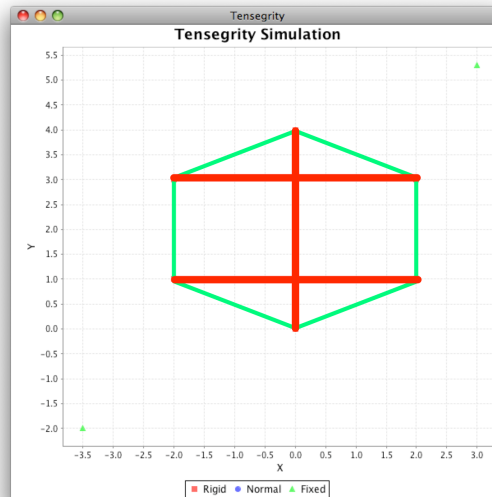
4.2.2 2D Display

The initial 2D display was just a case of plotting some points and updating this graph to provide animation (figure 4.4). A generic graphing package was chosen for this purpose, JFreeChart¹. It gave a simple method of updating and displaying charts on screen. This allowed the focus to be on the mathematics and correctness of the data, rather than on the mechanics of displaying it.

A simple point plot was chosen as it allowed the fastest development time and gave a quick mapping between the data behind the simulation and a visual representation. There are issues with flicker as a result of the continuous updating of the point locations but this is acceptable as the application was meant as more of a proof of concept.

¹<http://www.jfree.org/jfreechart/>

Figure 4.4: An annotated diagram, thin green and thicker red lines represent tensile and rigid connections respectively.



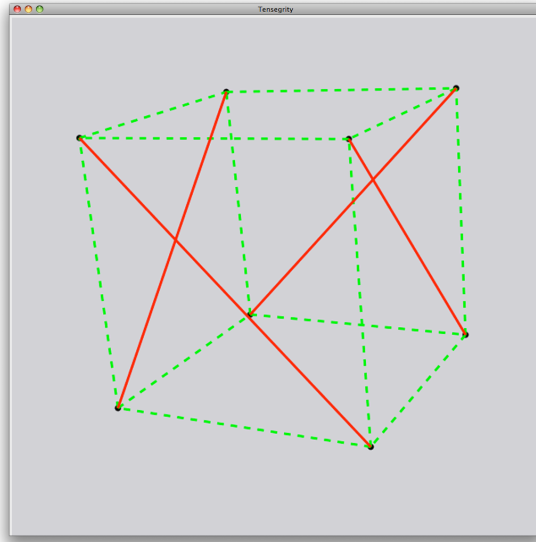
4.2.3 3D Display

Once the solving of systems involving tensegrity had been shown to be possible in real time and of possible interest in tissue simulation, the decision was taken to develop a 3D visualisation. Several options were available here, but the decision was taken to use the Java3D library² as it is a mature codebase with low-level driver integration. The fact that the intensive code is native and not part of the Java Virtual Machine means that it is much more likely to give high performance in more strenuous circumstances.

Initially, only simple nodes and connections were represented without faces. Later on, support for shading, lighting, textures and specularities were added. A simple box simulation was created and manipulated by random forces to test the behaviour of the system and check for realism (figure 4.5).

²<http://java.sun.com/products/java-media/3D/>

Figure 4.5: An early 3D version of the simulator with a basic tensegrity situation.



Changes Required

The conversion from 2D to 3D was not that involved as a result of the high cohesion within the classes. The majority of modifications were made in the vector and display classes. An extra component needed to be created and taken into account in such calculations as distance metrics. The display needed to feed the updated points to Java3D, which at first were set on an individual basis.

Exporting the result

In a medical simulation, verification and validation are a crucial aspect of the application. Even if the system looks to be performing correctly, it is imperative that there is a way to export data that can be inspected using other software. Gmsh³ was selected as a tool which provided the functionality required for visual verification of diaphragm and rib deformation.

The '.depl' format was cumbersome to implement. Its layout is shown in table 4.1. Header information alerts the parser to the kind of shape data incoming and the number of vertices. After the initial location has been stored, the change in x, y and z on a vertex by vertex basis from the previous iteration is required at each timestep. In systems with tens of thousands of vertices, storing this amount of data can become a strain on memory of the machine.

³<http://www.geuz.org/gmsh/>

Table 4.1: ‘Depl’ file format showing basic structure. The file is then flattened, preserving the newlines shown below.

Vertex	Timestep	X	Y	Z
1	$t_{initial}$	5	10	5
	t_1	0.9	0.4	0.1
	t_2	0.31	0.25	0.2
	t_3	0.4	0.1	0.3
	\vdots	\vdots	\vdots	\vdots
	t_n	$t_n(x) - t_{n-1}(x)$	$t_n(y) - t_{n-1}(y)$	$t_n(z) - t_{n-1}(z)$
2	$t_{initial}$	2	-6	18
	t_1	0.0	0.0	0.1
	t_2	0.5	0.3	0.2
	t_3	-0.17	-0.1	-0.2
	\vdots	\vdots	\vdots	\vdots
	t_n	$t_n(x) - t_{n-1}(x)$	$t_n(y) - t_{n-1}(y)$	$t_n(z) - t_{n-1}(z)$
\vdots	\vdots	\vdots	\vdots	

Optimisations

Once the system had been converted to 3D, it became possible to represent structures within the human body. These were provided in ‘.msh’ files which could be parsed using a custom built package. Once this had been completed, it was then possible to simulate the mass spring system using a large scale simulation. The results were not adequate as one frame was drawn every thirty seconds or more. After investigation, it became apparent that the bottleneck was the copying of data between the internal Java3D representation of the scene and that held in the model of the simulation. To solve this problem, Java supports another paradigm to update geometries. It allows a geometry to be passed as an array of floats. This array must be 1-dimensional, which reduces code-readability as offsets and multipliers must be used for access; this is necessary if higher speed is a requirement. It is then just a case of alerting the Java3D classes that changes have been made. This allowed the full scale simulations to run in real time therefore making tuning an option.

4.3 The Final Product

At the completion of the project, a fully-tunable simulation system was created with four options available. Two involving simple cubes to see the effect of tensegrity, one of a ‘perfectly’ modelled diaphragm and one from CT

scans (shown in figures 4.6). The control panel used to tune the simulation in real time can be seen in figure 4.7.

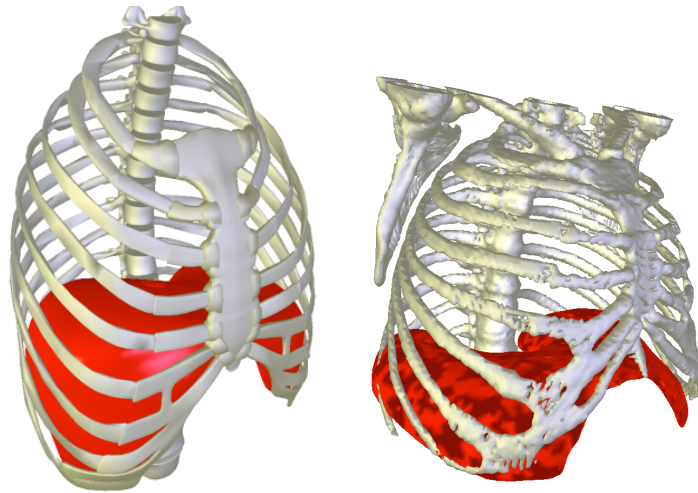
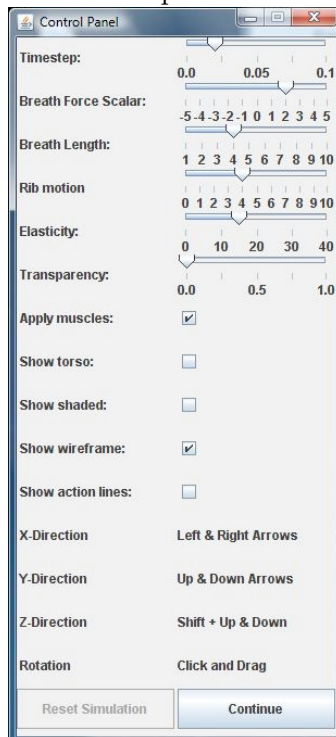


Figure 4.6: Images from the completed simulator. Left: The simulator using the ‘perfect’ model. Right: The result of using the data from the CT scan.

Figure 4.7: The final control panel for the ‘perfect’ simulator.



Where:

Timestep: The parameter used in the Euler solver.

Breath Force: Controls the force of the action lines, see figure 4.8 and equation 4.1.

Breath Length: Sets the length of each breath, see figure 4.8 and equation 4.1.

Rib Motion: Modifies the angle by which the ribs rotate.

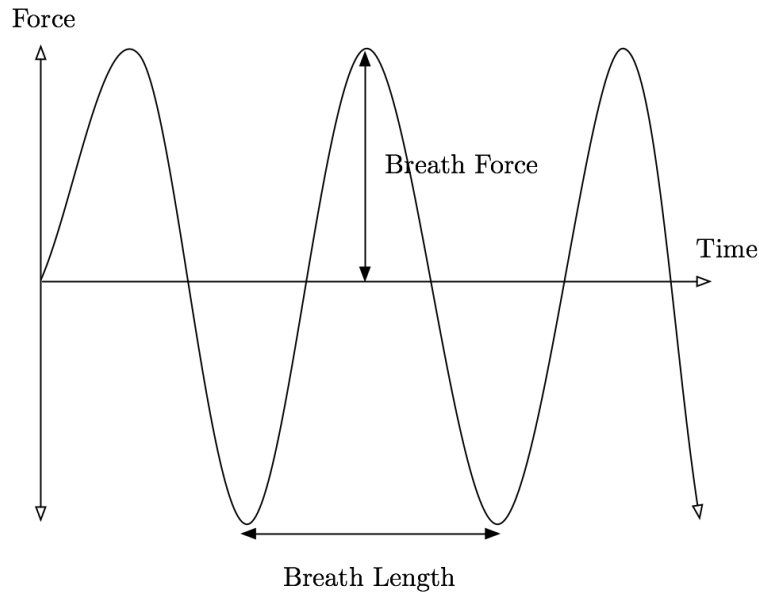
Elasticity: A parameter of the mass spring system.

Transparency: Used to show or hide the shaded faces.

Apply Muscles: Activates or deactivates the action lines.

Show etc.: Visual options

Figure 4.8: Illustrating the breathing motion.



$$y = C_1(\text{BreathForce})\sin\left(\frac{C_2T}{\text{BreathLength}}\right) \quad (4.1)$$

C_1 and C_2 are two constants chosen to make the values of Breath Force and Breath Length scale to interesting values for each simulation. T is the time elapsed in the simulation.

Chapter 5

Evaluation

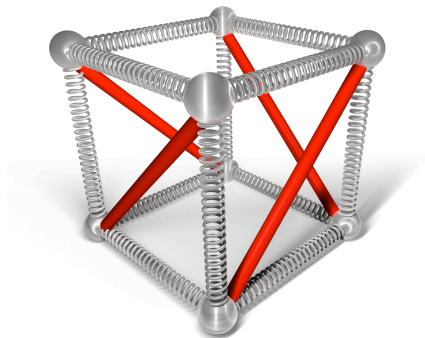
The Java implementation presented in the previous sections will now be quantitatively and qualitatively evaluated along with the influence of tensegrity.

5.1 Tensegrity

One of the main purposes of this project was to evaluate the viability of tensegrity for simulating the tendinous tissue of the diaphragm. This is not a trivial task and a few different methods were used to test this.

Firstly, it was decided to have a test to see if tensegrity did indeed provide more support than a typical mass spring system. This was tested using a simple cube as shown in figure 5.1. The cube was created with diagonal rigid supports between the upper and lower levels.

Figure 5.1: The standard tensegrity cube that was used in the initial evaluation of the effects of tensegrity.



Two points on the upper square were then manipulated by applying a sinusoidal motion. The data was exported and visually analysed. It was

obvious that the mass spring system collapses in on itself with much lower forces, while the tensegrity solution is able to maintain its shape. The results are shown in figures 5.2 and 5.3.

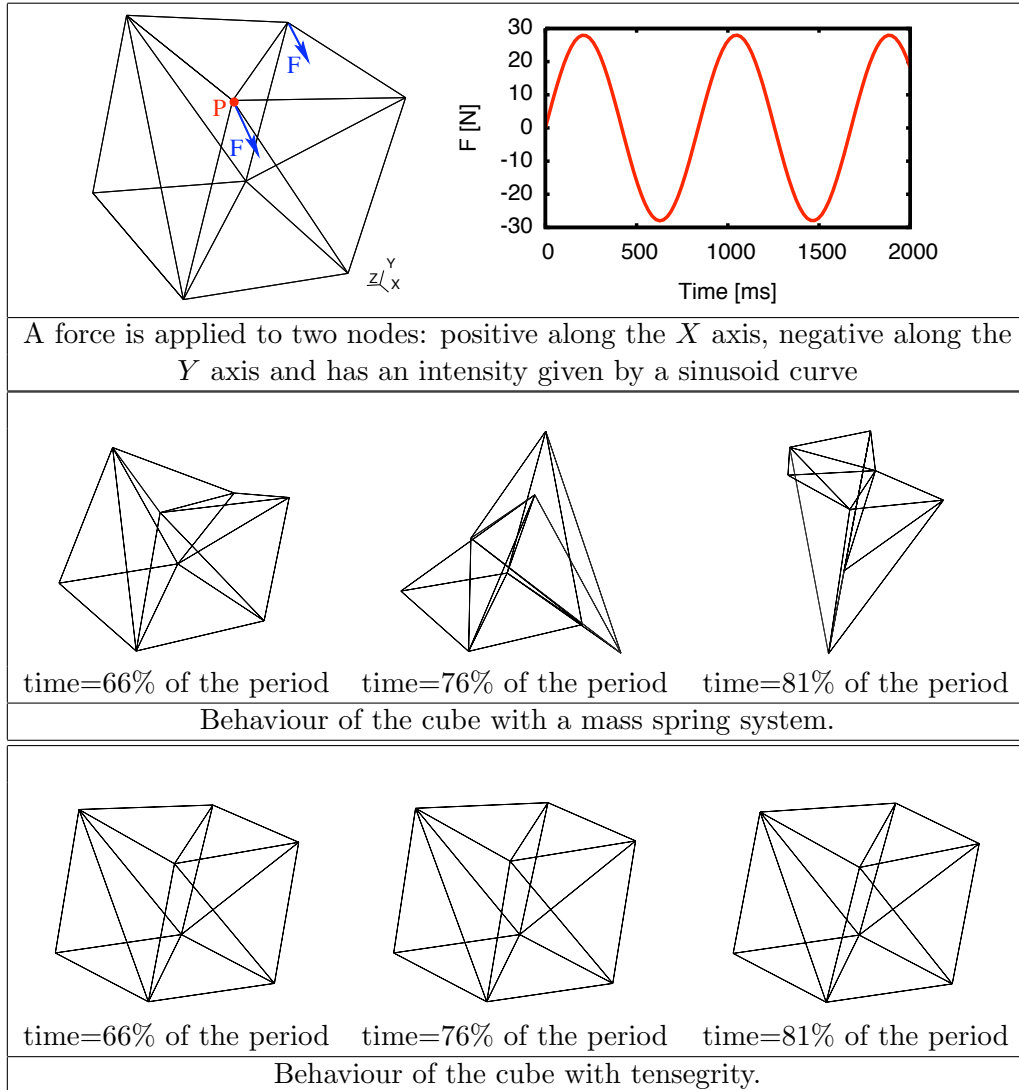


Figure 5.2: Cube ($10 \times 10 \times 10m^3$) evolution with time. *Top*: boundary conditions, *Middle*: results with a mass spring system, *Bottom*: results with tensegrity

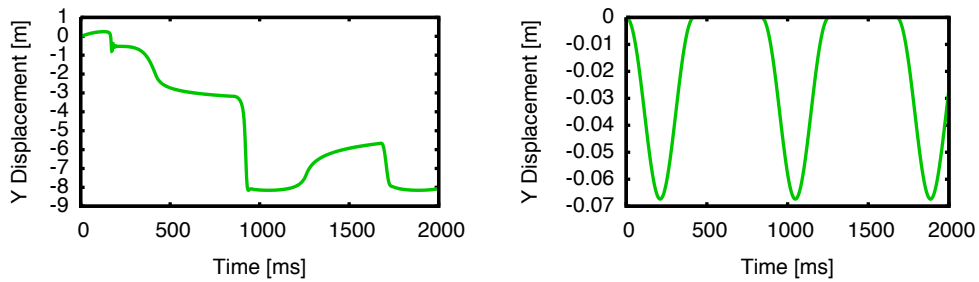


Figure 5.3: Y-value of a mass p in figure 5.2. The mass spring system is on the left, tensegrity on the right. The tensegrity is system able to absorb the pressure and rebound, the mass spring collapses.

5.1.1 Effect on Diaphragm

Figures 5.5 and 5.6 quantify the effect of tensegrity. The data for the graphs was extracted as follows:

1. A point on the top of the diaphragm was chosen, illustrated by figure 5.4.

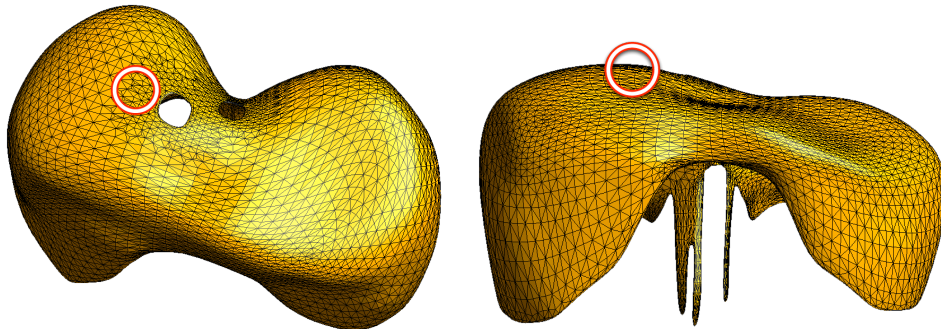


Figure 5.4: The chosen node was within the region shown.

2. 200 iterations were performed and the location of the chosen point was saved each time.
3. This simulation was performed with and without rigid links and the values were then subtracted to produce the data. The y-axis was chosen to be plotted as it is the main direction of translation in the diaphragm.

In figure 5.5 the difference is small initially and steadily rises. This could imply that there is just noise that is slowly being accumulated. It could also imply that the forces are not strong enough to require the rigid constraint.

Figure 5.6 provides a more interesting situation. In this case the forces were multiplied by a factor of 10. The difference between the values is no longer a steadily increasing function. There are abrupt troughs and hills as the constraints of tensegrity are applied. This shows that in more strenuous circumstances tensegrity can be used to enforce the proximity of the opposing sides of the diaphragm.

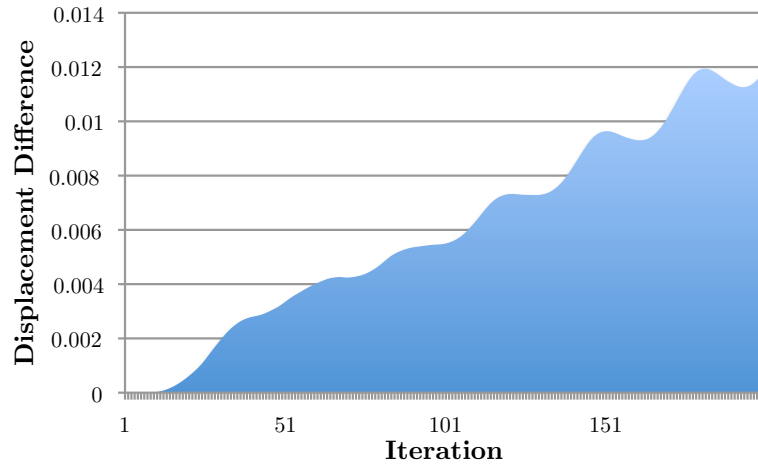


Figure 5.5: A plot of the difference between the y-component of the chosen point with and without rigid links. Weak action line forces were applied.

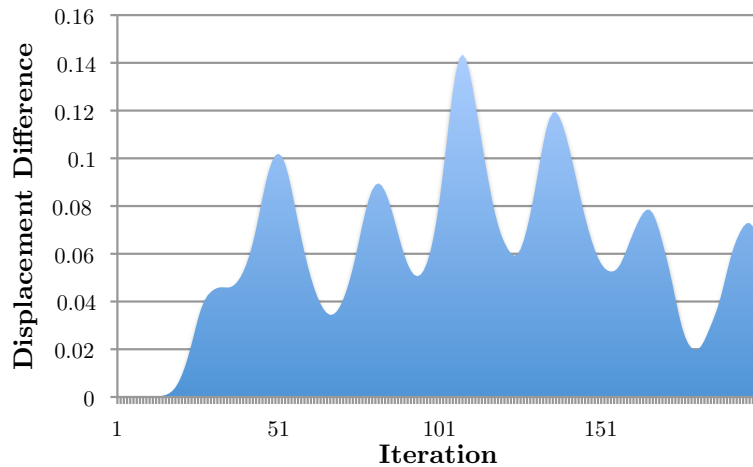


Figure 5.6: A plot of the difference between the y-component of the chosen point with and without rigid links. Strong action line forces were applied.

5.2 Diaphragm Motion

The 'Perfect' Model

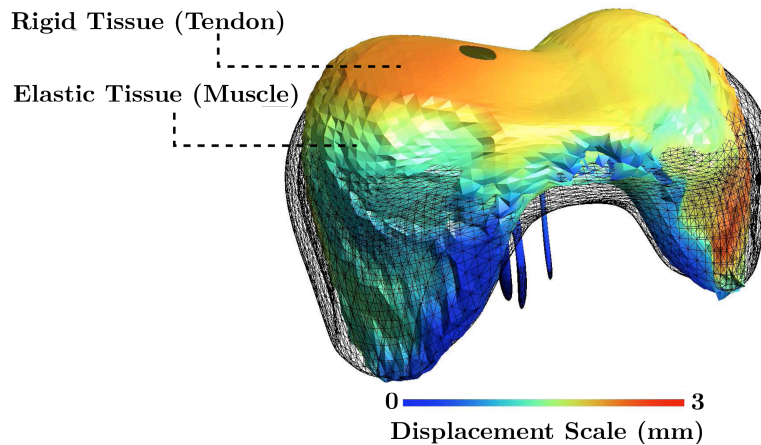


Figure 5.7: The resulting motion during the simulation.

The favourable results observed in the simple simulation were encouraging and the task of analysing the realism of tensegrity in diaphragm was to follow. For the so called 'perfect' model, a commercially available reference model of the diaphragm was used. It consisted of 11359 vertices and 22722 triangles. It can be very difficult to verify the realism of the motion in such a large model.

The wireframe visible in 5.7 represents the initial state at the end of an inhale. The coloured geometry represents the diaphragm at the end of an exhale. The upper region has kept its shape due to the tensegrity while the lower region has been more deformed as it had no rigid links to help retain its shape.

The three sources of motion on the diaphragm are visible here:

- *Muscle Motion*: The muscle relaxation that increases the height of the domes is visible.
- *Rib Kinematics*: On the sides of the diaphragm the motion as a result of the rib kinematics is visible.
- *Sternum Motion*: In the central front region the effect of the sternum is also visible as it influences the tissue to which it is connected.

Verification was also made by a clinical collaborator who validated the motion and also explained how only the diaphragm's domes descend during light breathing. The tuning options provided by the control panel mean such a situation is easily simulated. Under heavier breathing, the diaphragm can move substantially and this is also possible to recreate.

5.2.1 The Patient Model

Verification of the patient model was achieved using data from a 4D CT scan. The diaphragm was segmented and smoothed. This process is shown in 5.8. The resulting diaphragm mesh is composed of 20740 vertices and 41480 triangles.

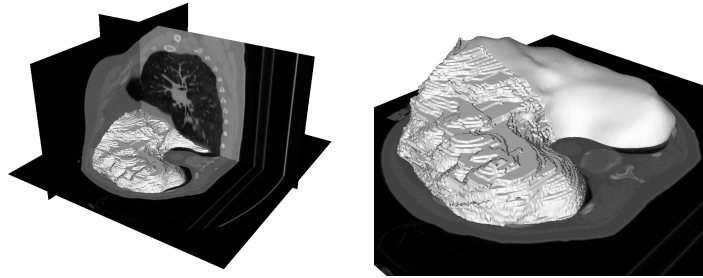


Figure 5.8: *Left*: Segmented diaphragm inside CT scan and *right*: Combination of segmentation and the mesh, the smoothing is visible on the right.

The simulation was validated by comparing the results of the simulation to the 4D CT scan data at the same point in the breathing cycle. The export function of the simulation software was used. The CT scan data was used for the real patient. The distance between the two were studied using the MESH¹ software to analyse meshes. It then found the Hausdorff distance². The results of the analysis are visible in figure 5.9. By visual inspection, clear similarities are present in several regions. The correspondence appears to be relatively close, but inaccurate in terms of magnitude in certain regions. This led to the conclusion that the tuned simulation mirrors this patient's true breathing pattern effectively.

5.3 Simulator

The tests below were performed on a dual core 2.4 ghz machine with 2 gigabytes of ram and a 256 megabyte graphics card. The operating system was Windows Vista Service Pack 1 with Java 1.6 installed. The parameters of the simulation were on their default values. All results are in seconds unless otherwise stated.

¹<http://mesh.berlios.de/>

²The Hausdorff distance between two sets of points is the longest distance an adversary can force you to travel by choosing a point in one of the two sets, from where you then must travel to the other set.

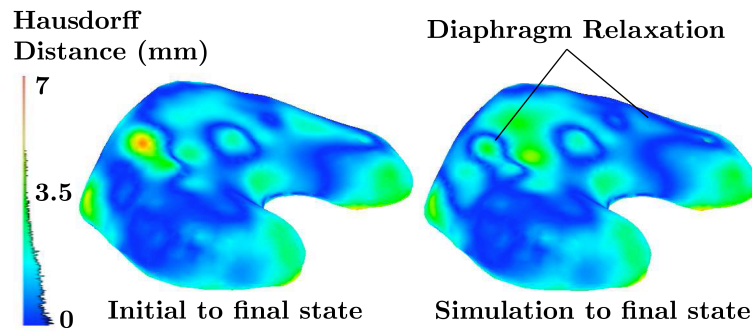


Figure 5.9: Distance error measurement between *left*: beginning and end of real inhale and *right*: simulated end of inhale and real end of inhale.

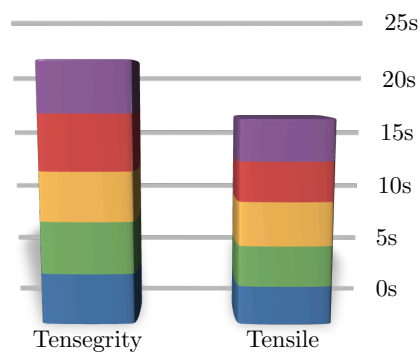
5.3.1 Cube Simulation

To test the cube simulation and the influence of tensegrity on simulation performance, 100,000 iterations were performed and the two cubes with sinusoid forces being applied.

Figure 5.10: Cube simulation performance (in seconds).

	Tensegrity	Tensile
Simulated Vertices	8	8
Trial 1	4.174	3.155
Trial 2	4.287	3.337
Trial 3	4.069	3.602
Trial 4	4.599	3.244
Trial 5	4.198	3.387
Average	4.2654	3.345

Figure 5.11: A graphical representation of the results, each colour represents a separate trial.



It can be seen from the graph that the addition of the tensegrity constraint makes for a slightly slower simulation, but it is still more that adequately fast. The use of tensegrity shows a computational cost of 27%. This cost is offset by the desirable strength that the rigid links provide as discussed in section 5.1.

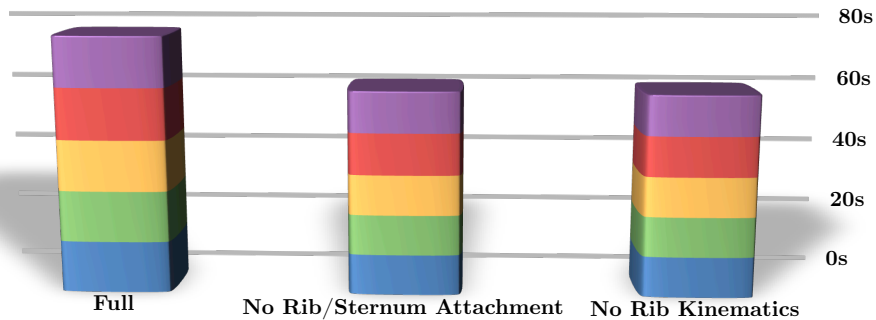
5.3.2 Diaphragm Simulation

Now that the realism of the simulations have been evaluated it is time to verify the computational cost of the simulation parameters. Two cases were tested for the perfect simulator. In one case, the fixed points were no longer calculated and adjusted at each iteration. The other variation was the removal of rib kinematics.

Figure 5.12: Diaphragm simulation performance (in seconds).

	Full	No Rib/Sternum Attachment	No Rib Kinematics
Simulated Vertices	20740	20859	11359
Trial 1	15.472	12.174	12.256
Trial 2	15.39	12.098	12.097
Trial 3	15.448	12.126	12.043
Trial 4	15.654	12.565	12.129
Trial 5	15.435	12.534	12.261
Average	15.4798	12.2994	12.1572

Figure 5.13: A graphical representation of the results, each colour represents a separate trial.



It can be seen that the slowest simulator is the ‘real’ simulator. This can be attributed to the high count of vertices involved in the simulation. The ‘perfect’ simulator is slightly faster and the variations do make a difference to the speed of calculation. Rib kinematics make a considerable difference to the speed. This is a result of some large equations being solved to update the locations at each iteration. If a patient is observed to only be using the

diaphragm and no rib action during respiration, a large computational cost can be avoided.

5.3.3 Influence of Mesh Size with a Heterogeneous Model

Another influencing factor on the speed of a simulation is the size of the mesh. Both the ‘perfect’ and patient models were tested with different resolutions.

Figure 5.14: The timed results of the perfect mesh being resized.

	High	Medium	Low
Simulated Vertices	<i>20909</i>	<i>16023</i>	<i>12047</i>
Trial 1	12.174	9.543	7.851
Trial 2	12.098	9.63	7.735
Trial 3	12.126	9.552	7.907
Trial 4	12.565	9.725	7.842
Trial 5	12.534	9.603	7.763
Average	12.2994	9.6106	7.8196

Figure 5.15: The perfect mesh times.

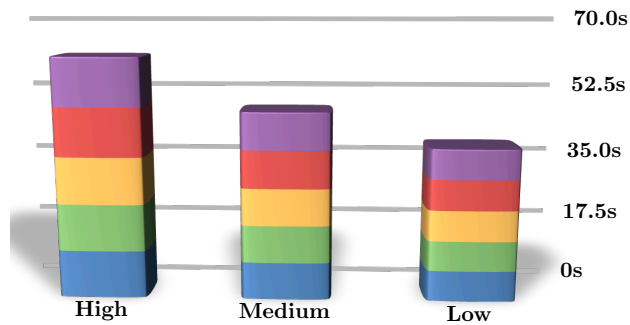
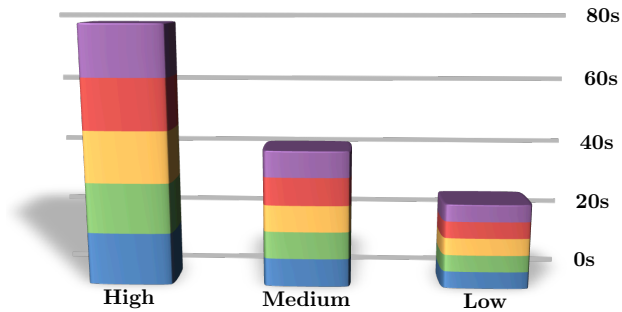


Figure 5.16: The timed results of the patient mesh being resized.

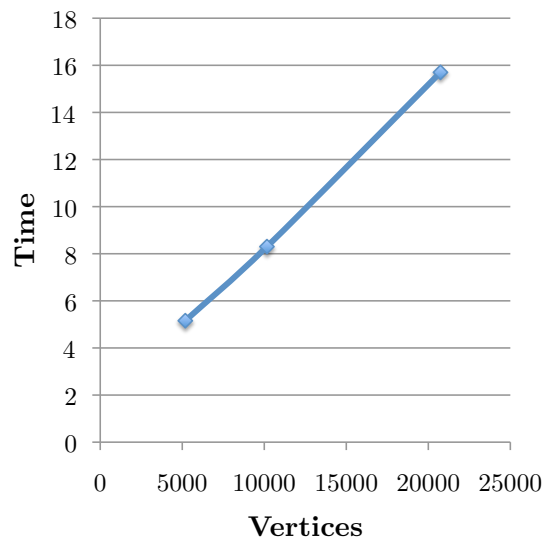
	High	Medium	Low
Simulated Vertices	<i>20740</i>	<i>10162</i>	<i>5185</i>
Trial 1	15.931	8.597	5.167
Trial 2	15.386	8.313	5.092
Trial 3	15.778	8.077	5.217
Trial 4	15.659	8.501	5.081
Trial 5	15.748	8.018	5.225
Average	15.7004	8.3012	5.1564

Figure 5.17: A graph of the patient data.



A quick look at figures 5.14-5.17 shows an intuitive linear relationship between the number of vertices in the scene and the time taken in computation. Figure 5.18 shows this relationship in the patient data case.

Figure 5.18: A graph of simulation time against the vertices in the heterogeneous patient simulator.



5.3.4 Influence of Mesh Size with a Homogenous Model

Similar to section 5.3.3 the mesh sizes were reduced in complexity and then tests were conducted without rigid links, creating a standard mass spring model.

Figure 5.19: The timed results of the perfect mesh being resized.

	High	Medium	Low
Simulated Vertices	<i>20909</i>	<i>16023</i>	<i>12047</i>
Trial 1	11.542	9.703	7.832
Trial 2	11.831	9.206	7.645
Trial 3	11.732	9.144	7.451
Trial 4	11.657	9.043	7.621
Trial 5	11.672	9.691	7.705
Average	11.6868	9.3574	7.6508

Figure 5.20: The perfect mesh times.

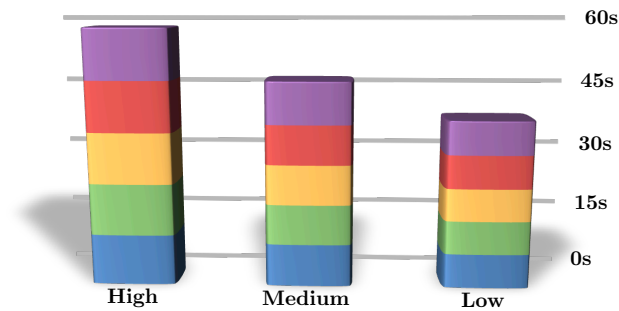
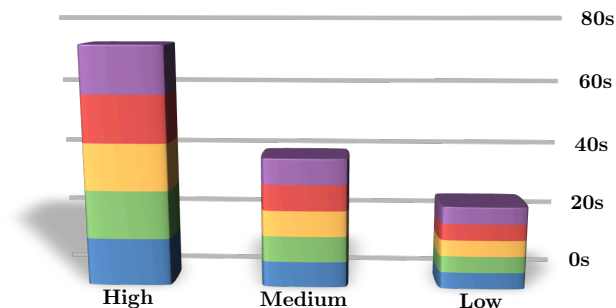


Figure 5.21: The timed results of the patient mesh being resized.

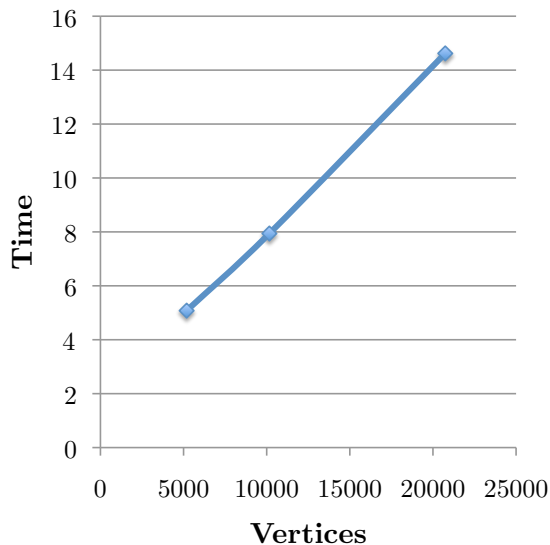
	High	Medium	Low
Simulated Vertices	<i>20740</i>	<i>10162</i>	<i>5185</i>
Trial 1	14.522	7.839	5.063
Trial 2	14.865	7.929	5.047
Trial 3	14.469	7.92	4.983
Trial 4	14.629	8.016	5.087
Trial 5	14.614	7.992	5.214
Average	14.6198	7.9392	5.0788

Figure 5.22: A graph of the patient data.



As seen in figures 5.19-5.22 the recorded times were very similar to those with tensegrity and were only off by a small percentage. This demonstrates that even in large models with hundreds of rigid links, the computational time is not massively increased. The linear relationship between vertices and time still holds as shown in figure 5.23.

Figure 5.23: A graph of simulation time against the vertices in the homogeneous patient simulator.



5.3.5 Conclusions

After many tests, table 5.24 provides a succinct summary of all the data. It gives the average iterations per second after five trials. The relationship between the parameters of the simulation and the mesh size are available here, but are better visualised in figures 5.25 and 5.26. Figure 5.25 shows more interesting behaviour. The computational time of the rib kinematics is

shown to be quite high as when they are removed in the ‘Action Lines Only’ trial the simulation hits its peak. Figure 5.26 is a fairly intuitive graph that shows an inverse relationship between model complexity and iterations per second.

Figure 5.24: A summary of the iterations per second of all the simulations. The trials have been averaged into a single value.

Perfect Model					
	Vertices	Heterogenous Model (All Forces)	Homogenous Model (All Forces)	Rib Kinematics Only (Heterogenous)	Diaphragm Contraction/Relaxation Only (Heterogenous)
High	20909	8.13	8.56	8.24	9.68
Medium	16023	10.41	10.69	10.93	11.76
Low	12047	12.79	13.07	13.27	14.70
Patient Model					
	Vertices	Heterogenous Model (All Forces)	Homogenous Model (All Forces)		
High	20740	6.37	6.84		
Medium	10162	12.05	12.60		
Low	5185	19.39	19.69		

Figure 5.25: A 3D plot of the iterations per section of the ‘perfect’ simulation. The axes are: simulation parameters, mesh quality, and iterations per second.

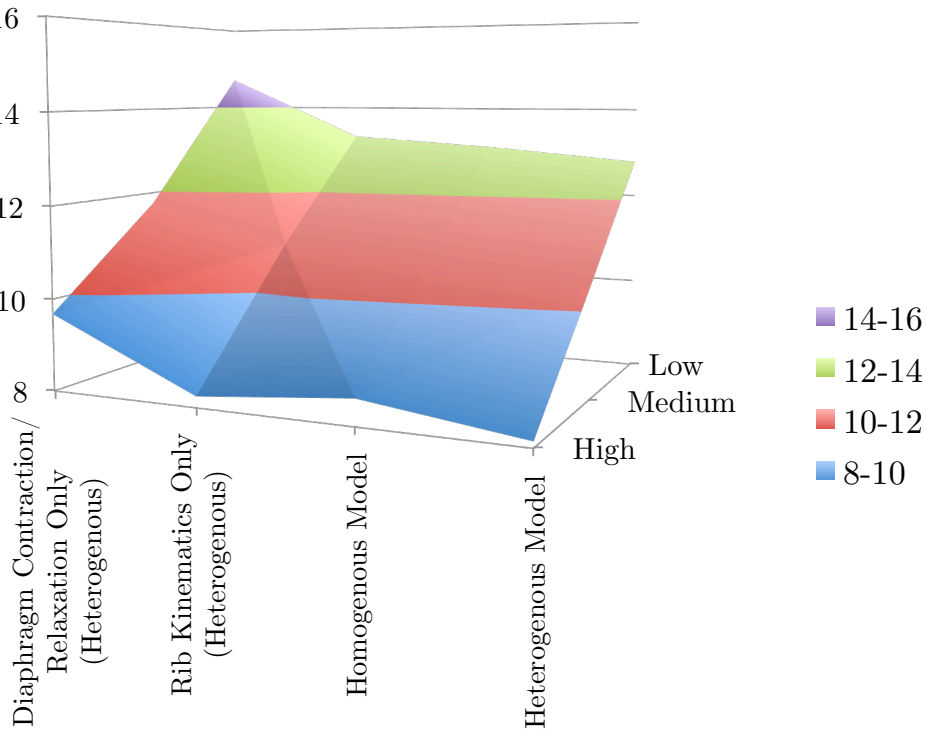
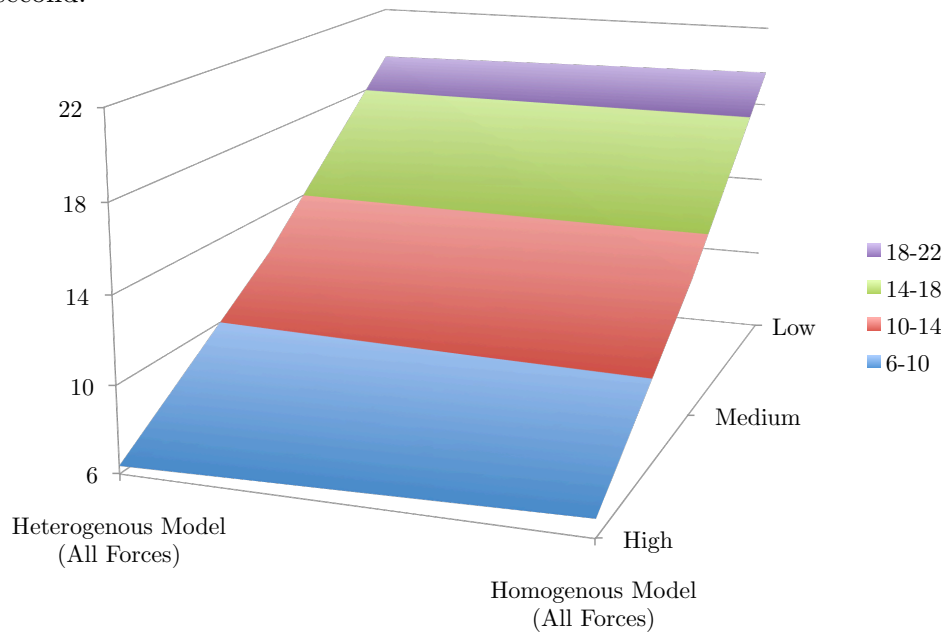


Figure 5.26: A 3D plot of the iterations per section of the patient simulation. The axes are: simulation parameters, mesh quality, and iterations per second.



Computational Cost of Tensegrity on Simulations

Figure 5.27 shows the cost on the speed of the simulator that the addition of tensegrity makes. The largest change is visible in the high quality perfect model. This is due to the very high number of connections created in this case. The overall cost of the tensegrity connections is seen to be relatively low.

Figure 5.27: The percentage change in each of the simulations.

Perfect Model					
	Vertices	Tensegrity	Mass Spring	Rigid Links: Vertices Ratio	Percentage Change
High	20909	8.13	8.56	1663:20909	5.24%
Medium	16023	10.41	10.69	1022:16023	2.71%
Low	12047	12.79	13.07	602:12047	2.21%
Patient Model					
	Vertices	Tensegrity	Mass Spring	Rigid Links: Vertices Ratio	Percentage Change
High	20740	10.40	10.78	4858:20740	3.65%
Medium	10162	10.41	10.69	648:10162	2.71%
Low	5185	10.39	10.79	940:5185	3.85%

Chapter 6

Conclusion

6.1 Discussion

6.1.1 Work Completed

By the end of the project, a tuneable simulator capable of simulating tissues within the body was created. It receives input in the form of mesh files and rigid links are created automatically once a plane has been defined. Muscle action is simulated through action lines. After initialisation, the simulation is controlled via a panel containing parameters such as breath length, force and rib motion. These parameters allow the simulation to match closely a patient's breathing pattern. Finally, an analysis and evaluation was conducted to investigate the effectiveness and computational cost of tensegrity.

6.1.2 Project Outcomes

The result is a code base that is malleable to future modifications to fit other purposes. It can be seen through the background work that the diaphragm is composed of materials of various properties and heterogeneity is needed for a true simulation of diaphragm motion. Tensegrity can be shown to provided additional support in structures (figure 5.2). Verification of realism was adequate but, for an organ with so many influencing factors, more evaluation is required.

6.1.3 Limitations

At the current time, although the design is very tuned to the purpose of simulating the diaphragm, some minor work would be required to modify it to a more generic form. There are also several other useful paradigms that SOFA has that could be desirable here. Mapping between a visual model and a mechanical backing would allow the simulation to take place at a lower level and the visual model to be of a much higher quality. Finally, probably

the largest limitation is the lack of interchangeability of simulation methods. If the project had been completed wholly in SOFA, other tissue simulation methods could quickly be substituted for easy comparison.

6.2 Future work

The project had a large research aspect to it and therefore there is plenty of scope for future work.

6.2.1 Integration with SOFA

Now that the use of tensegrity has been found to be valid and worth implementing, it would be ideal to integrate it into SOFA. Creating a system of mixing mass spring and rigid links could be completed in the current version of SOFA. For the potential mixing of rigid and the finite element method, a newer version of SOFA with the combination of force fields would be required.

6.2.2 Educational Context

Once a more thorough and complete evaluation has been completed and any modifications made, the tool could be used to give an insight into the motion of the diaphragm during different kinds of breathing. The results should be enough to give the correct impression and coupled with the interactivity and visible combination of forces, it could prove to be an interesting teaching tool.

6.2.3 Haptic Integration

Haptic means pertaining to the sense of touch. In medical simulation, the use of touch to give the impression of applying forces or the feeling of vibration adds a very strong element of realism. A future project could be the integration of haptic elements to the simulator. Liver access is greatly complicated by the motion of the diaphragm and a haptic aspect of the simulation could prove to be a valuable tool in training surgeons.

Bibliography

- [AC] *M.N. Acharya*, “Modelling of Diaphragm Motion for Simulation of Liver Access”, *Surgery and Anaesthesia BSc Project, Imperial College London, May 2008, Pages 13-14, 33-35.*
<http://www1.imperial.ac.uk/resources/A7C4A779-D4F9-4527-90BC-6D5C1A6435BC/>
- [BL] *Y. Bhasin, A. Liu*, “Bounds for Damping that Guarantee Stability in Mass-Spring Systems”, *The Surgical Simulation Laboratorye, 2005, Pages 1-3.*
<http://simcen.org/pdf/bhasin%20mmvr%202006.pdf>
- [BW] *D. Baraff, A. Witkin*, “Large Steps in Cloth Simulation”, *Robotics Institute, Carnegie Mellon University. Pages 1-5.*
<http://ai.stanford.edu/~latombe/cs99k/2000/cloth.pdf>
- [DV] *A. Didier, P. Villard, J. Bayle, M. Bewe, B. Shariat*, “Breathing Thorax Simulation based on Pleura Physiology and Rib Kinematics”, *Hôpital Louis Pradel, Lyon, France, 2007, Pages 1-5.*
<http://www710.univ-lyon1.fr/~mbeuve/pvillard/zurich07.pdf>
- [GU] *A. Guillaume*, “Simulation 3D du comportement biomécanique des cellules”, *Masters Thesis, Université Claude Bernard Lyon I, 28 June 2004, Pages 7-12.*
- [IN] *D.E. Ingber*, “Opposing views on tensegrity as a structural framework for understanding cell mechanics”, *Journal of Applied Physiology 89: 1663-1678, 2000.*
- [ML] *U. Meier, O. Lopez, C. Monserrat, M.C. Juan, M. Alcaniz*, “Real-time deformable models for surgery simulation: a survey”, *Computer Methods and Programs in Biomedicine (2005) 77, 183-197.*
- [NT] *Nedel, Luciana Porcher & Thalmann, Daniel*, “Real Time Muscle Deformations Using Mass-Spring Systems”, *EPFL - Swiss Federal Institute of Technology, Pages 2-11.*
- [SOFA] SOFA, *Simulation Open Framework Architecture*,
<http://www.sofa-framework.org/>

- [VB] *F.P. Vidal, F. Bello, K.W. Brodlie, N.W. John, D. Gould, R. Phillips, N.J. Avis*, “Principles and Applications of Computer Graphics in Medicine”, *Computer Graphics Forum*, Volume 25, Number 1, March 2006 , Pages 113-137.

Appendix A

Appendix

A.1 User Guidance

A.1.1 Adding a new object

1. *Convert your model to '.msh' format:* Before importing a model it needs to be converted into the correct format. A parser has been created that will do this for you and is included. It requires simple '.obj' models without textures or lighting.
2. *Add Java3D and Topology variables:* Java3D needs to have variables to store the geometries you are going to be adding, these are declared at the top of the simulator classes.

```
static Topology diaphragm = null;
```

```
TriangleArray diaphragmtriangles;  
Shape3D diaphragmshape;
```

3. *Create an entry in init3D():* This method is where the Java3D parameters of the display such as colour, specularly and material are set.

```
if (diaphragm != null) {  
    Appearance app = new Appearance();  
  
    Material material = new Material();  
    PolygonAttributes pAttr = new PolygonAttributes();  
  
    pAttr.setPolygonMode(PolygonAttributes.POLYGON_FILL);  
    pAttr.setCullFace(PolygonAttributes.CULL_NONE);  
}
```

```

pAttr.setBackFaceNormalFlip(false);

material.setShininess(128f);
material.setDiffuseColor(1.0f,0.0f,0.0f);
material.setAmbientColor(1.0f,0.0f,0.0f);

app.setMaterial(material);
app.setPolygonAttributes(pAttr);

diaphragmtriangles = new TriangleArray(diaphragm.triangles.size()*3,
GeometryArray.COORDINATES | GeometryArray.BY_REFERENCE |
GeometryArray.NORMALS);
diaphragmtriangles.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
diaphragmtriangles.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
diaphragmtriangles.setNormalRefFloat(diaphragm.normals);
diaphragmtriangles.setCoordRefFloat(diaphragm.locations);

diaphragmshape = new Shape3D();
diaphragmshape.setCapability(Shape3D.ALLOW_GEOMETRY_WRITE);
diaphragmshape.setAppearance(app);
diaphragmshape.addGeometry(diaphragmtriangles);
transformGroup.addChild(diaphragmshape);
}

```

4. *Parse file and setup connections in init():* The model needs to be bound to the diaphragm topology. Connections also need to be created.

```

MSHParser diaphragmparser = new MSHParser();
diaphragm = new Topology("Diaphragm");

try {
    diaphragmparser.parseFile("diaphragm.msh",diaphragm,
MSHParser.SETUPCONNECTIONS,-135);
    diaphragm.setupArrays();
    new ActionLine(diaphragm.nodes.get(9113).initialloc,
    new Vector(0.0f,1.0f,0.0f), 17.0f, new Vector(0.0f,0.2f,0.0f), diaphragm);
    new ActionLine(diaphragm.nodes.get(1557).initialloc,
    new Vector(0.0f,1.0f,0.0f), 14.0f, new Vector(0.0f,0.2f,0.0f), diaphragm);
} catch (Exception e) {
    System.out.println("Error parsing files");
    e.printStackTrace();
    return;
}

```


The `MSHParser.SETUPCONNECTIONS` tells the parser that you would like the mass spring connections to be created for you. This may not be desirable if you are going to be using this as a rigid structure, such as the spine. In this case you should use `MSHParser.NOCONNECTIONS`. The `-135` argument of the `parseFile(...)` call is a y-value scalar. This is used to centre the diaphragm as the model was off-center. 0 should be placed here if the model is already centred around the origin.

Actionlines are also created here with the form: `new ActionLine(Origin, Direction Vector, Radius, Force Vector, Object);`

5. [Optional] *Create rigid links in init()*: The topology can be told to create rigid links between its internal surfaces.

```
diaphragm.setupInternalConnections(3474,1994,636,1.5f);
```

The first three are the node numbers of the nodes to define the plane. The fourth argument is the maximum distance each node will consider as neighbours.

6. [Optional] *Create attachments between objects in init()*: The topology can be attached to other topologies to simulate connected tissues.

```
diaphragm.stickTo(sternum_ligaments, 0.5f,20f);
diaphragm.resolveClosestEntries();
```

The first argument is the opposing tissue you wish to attach to, the second is the distance limit for neighbours and the third is a scalar for the forces to be applied.

Finally, the computation to find closest node in the neighbouring topology is completed by `diaphragm.resolveClosestEntries()`.

7. [Optional] *Add references in toggleShaded()*: The GUI allows shaded models to be hidden. A reference needs to be added here if you want your object to be hidden also.

```
public void toggleShaded() {
    if (diaphragmshape.numGeometries() > 0) {
        diaphragmshape.removeAllGeometries();
    } else {
        diaphragmshape.addGeometry(diaphragmtriangles);
    }
}
```

A.1.2 Action Line Fall Off Modication

The action line method is very amenable to tuning for a variety of situations. For this reason I will show the method that needs to be modified to perform the tuning.

```
public void applyForces(float scalingfactor) {
    for (Node n: nodes) {
        float[] p = n.getLocation();
        Vector3f point = new Vector3f(p[0],p[1],p[2]);
        float distance = point.y-origin.y;
        float falloff = distance/radius;

        if (falloff > 0 & n.force == null) {
            forcevector.Multiply(scalingfactor*falloff);
            n.addForce(forcevector.clone());
            forcevector.Divide(scalingfactor*falloff);
        }
    }
}
```

As is visible here the fall off is currently concerned with the ratio between the difference of the y values of the point and the origin and the radius. Changing this fall off calculation will provide vastly different behaviour of the action line.

A.1.3 Euler Solver Implementation

The solver is another crucial aspect of the project that could be tuned to provide better convergence or a faster simulation.

The simulator has the following code within the `performIteration()` method.

```
for(Node n: t.nodes) {
    Vector v = n.calculateTensionForces();
    n.applyForce(v);
    n.updateVelocity();
    n.updateDamping();
    n.updatePosition();
}
```

The tension forces are calculated using equation 2.1. `applyForce(Vector f)` is defined as follows and is a simple application of Newton's second law, $f = ma$.

```

public void applyForce(Vector f) {
    /* f = ma, a = f/m */
    if (!this.isFixed()) {
        f.Divide(mass);
        acceleration = f;
    } else {
        acceleration = new Vector(0,0,0);
    }
}

```

updateVelocity() is defined as:

```

public void updateVelocity() {
    acceleration.Multiply(Simulator.TIMESTEP);
    velocity = Vector.add(velocity, acceleration);
}

```

updateDamping() is defined as follows where dampingfactor is an experimentally chosen constant of -0.02 .

```

public void updateDamping() {
    applyForce(Vector.Multiply(velocity, dampingfactor));
    velocity = Vector.add(velocity, acceleration);
}

```

Finally updatePosition():

```

public void updatePosition() {
    Vector distance = Vector.Multiply(velocity, Simulator.TIMESTEP);
    newloc.x += distance.x;
    newloc.y += distance.y;
    newloc.z += distance.z;
}

```

The numerical integration shown here is simple and quick to process. Other methods such as Runge–Kutta could be used to provide a better solution.