Real-time Clothing Animation: 
An Analysis of the Model Proposed by 
Nikitas Tsopelas

Robert Bolter 
MRes CVIPGS 
Department of Computer Science 
University College London

This report is submitted as part requirement for the MRes Degree in Computer Vision, Image Processing, Graphics and Simulation at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.
Abstract

In 1994, Nikitas Tsopelas completed his PhD thesis in which he proposed a model for clothing modelling, claiming to be real-time. Since then little attention has been paid to his model, yet the problem of effective real-time clothing modelling still exists. In this report I examine Tsopelas’s model in detail and evaluate the potential of Tsopelas’s model for real-time or even interactive-time animation.

This work has built upon the original code of Nikitas Tsopelas. Improvements have been made to the rendering of the garments with the addition of texture mapping and conversion to OpenGL. Interactive viewing and deformation of the surface has been implemented. A method for the randomisation of hinge placement is also detailed.

In a simulation of a sleeve bending, (5520 polygons) frame rates of 10 f/s have been achieved on a Pentium II based 700MHz PC. (without hardware graphics acceleration) frame rates of 3 f/s have been achieved for complete modeling of a jumper, (31500 polygons).

Nikitas’s layered kinematic models for animating complex deformations remains unimplemented as does the Inverse dynamics procedure for modeling elastics. A detailed analysis of the complexity and predicted timings for the layered model are given.
Chapter 1 - Introduction

Background and Motivation
One of the goals of the computer graphics community is the synthesis of realistic images of the real world. Cloth is an important part of many real world scenes, most notably our clothing is principally made from cloth. In most cases it is not just our task to render cloth, we also need to somehow model its physical behaviour. Cloth, at least at a high level, is a relatively simple real-world object. But the forces applied to it by its environment such as interaction with objects, gravity etc. cause it to deform into complex structures that are difficult to model ‘virtually’. If we can get a computer to model these deformations we can reduce, or even remove, the need for human intervention when modeling cloth in a virtual environment. This makes the generation of animations of cloth a possibility. By removing the need for any human interaction we can then (if fast enough algorithms exist) produce interactive models of cloth, thus enabling the cloth to be modeled in an interactive virtual environment. One possible application is to be able to realistically model clothing responding to the movements of a virtual actor.

In computer graphics visual results are as important if not more so than physical accuracy. For physically based models of complicated objects such as cloth, in general, physical accuracy comes with the penalty of greater computation times and complexity. A broad range of cloth modeling techniques have evolved that reflect various compromises between physical accuracy and computation time. Some models are capable of producing dynamically deforming animations of cloth, whereas others are only suitable for producing static images of cloth draping over objects.

Tsopelas' model uses a Hybrid technique to model cloth, that is, it uses geometric techniques enriched by physical constraints. This results in a dynamic model that aims to be both quick and realistic in simulating clothing, but it does not aim to be completely physically accurate. Nikitas devotes a chapter of his thesis on the CAD of clothing and the ‘seaming’ together panels, but this is not within the scope of this project. I have concentrated on his deformation algorithm and his layered model as detailed analysis of these should enable me to answer the following question: Is Nikitas’s model capable of real-time or interactive-time animation?

Not being from an engineering background, I have decided that the detailed treatment of elasticas is beyond the scope of this project, and I will build upon other areas of the deformation algorithm.

In the rest of this chapter I give a review of various other clothing modeling techniques, classifying the various techniques into classes and identifying particular cloth modeling techniques that are relevant to real-time cloth modeling. In chapter 2 I give a detailed description of Nikitas’s deformation algorithm. Chapter 3 then analyses Nikitas’s code, identifies areas for improvement and discusses possible ways to implement these improvements. The implementation of these improvements is then detailed in Chapter 4 and the results presented in chapter 5. Finally Chapter 6 concludes the report and answers the question of whether this model is capable of interactive or real-time animation.

Literary review
The study of the behaviour of cloth started in the textile and mechanical engineering communities. Their focus was on developing ways to measure certain aspects of cloth behaviour (most notably the ‘Drapometer’ developed by Chu et.al. for measuring the drape of cloth) and model its mechanical properties at the low level of thread crossings. The problem of predicting the shape and dynamics of deformed cloth objects was not addressed by the textile community until the late 80’s, by which time the computer graphics community had also began to research this area. The computation of cloth shape based on its mechanical properties is expensive; but physical accuracy was not the prime concern of the computer graphics industry, they looked for simpler models that aim to resemble the appearance of
cloth, rather than reproduce it exactly. This lead to the development of various classes of techniques for cloth modeling. In the following section I identify and present a classification of these techniques.

**A Unified Classification of Cloth Modeling Techniques.**

In their survey paper on computer graphics techniques for modeling cloth Ng et.al.\(^2\) Identified three categories for cloth modeling techniques specific to the computer graphics industry:

- **Physical**
  These are models that are based around the physical properties of cloth. They usually approximate cloth as a discrete grid of particles. Force or energy calculations between neighbouring points are used to simulate particle interaction at a low level, which leads to the macroscopic behaviour of cloth.

- **Geometric**
  In these models the macroscopic features of cloth such as folds and drape are modeled geometrically. They usually require a significant amount of human intervention and are more of a drawing aid than a stand alone cloth model.

- **Hybrid**
  This is a mixture of the Physical and Geometric modeling. Physical properties of cloth are used to provide physical constraints that control a geometric model, removing some or all of the need for human intervention.

These were also the classes identified by Nikitas Tsopelas in his thesis. Nikitas also made the distinction between static and dynamic models.

- **Static**
  These models are only capable of simulating the static behaviour of cloth. Geometric models are usually static as they require human intervention and can only provide a geometric approximation to the shape of cloth, not it's underlying behaviour.

- **Dynamic**
  When producing animated simulations of cloth, models need to be able to simulate the progressive formation of deformations. Physically based models based on force are capable of this as they model the underlying behaviour of cloth. Techniques that can model this progressive formation are called Dynamic models.

**A survey of Cloth modeling techniques for computer graphics.**

There has been much research that covers many aspects of cloth modeling, from the automated design of clothing to specific problems of cloth modeling such as collision detection. Relevant to this project are those that describe methods for simulating the deformation of cloth, in particular dynamic methods. In the rest of this chapter, I provide a survey of these techniques. Due to the huge volume of papers published in this area I have not been able to review every one. I present those that best provide a cross-section of the broad range of approaches used, concentrating on those that are likely to be important to real-time cloth modeling and animation.

Criteria for the evaluation of cloth modeling algorithms.

The goals of various approaches differ, some aim to be physically accurate whilst others sacrifice accuracy for speed. In order to compare the various of approaches used I have used the following criteria:

- **Speed and Complexity**
  Speed is of particular importance when we want to use the algorithm for generating animated sequences. In assessing speed, we need to know the complexity of the algorithm, i.e. to what extent the algorithm becomes more complex as the size of the model increases.

- **Aesthetic / Physical Accuracy**
  How realistic the model looks, and whether the model preserves certain physical constraints of cloth such as area preservation and maximum curvature.

- **Capability**
Which particular complex behaviours of cloth the model is capable of simulating.
?? Level of interaction
How much human intervention the model requires.
?? Collision Detection
To what extent the model deals with self-collision and collisions with external objects, and how realistic the dynamics of collision response is.

Geometric Approaches

**J. Weil - 'The synthesis of cloth objects'**

This paper describes a simple geometric algorithm suitable for the synthesis of static images of draping cloth where some of its points are constrained to fixed locations in space.

The catenary equation describes the shape of a perfectly flexible cable hanging under its own weight between two fixed endpoints. The shape of a catenary is given by

\[ y = a \cosh\left(\frac{x}{a}\right) \]

Where \( a \) is the height of the lowest point, corresponding to \( x = 0 \).

![Figure 1-1 - A catenary curve with \( a = 2 \)](image)

The fixed end points can be at any two points along this curve. Using the arc length of the curve the equation can be transformed and reparameterised to give the shape of a cable given its length and two end points.

The Algorithm

The cloth is represented as a discrete rectangular grid of 3D points. The grid is a discrete mapping between \([u,v]\) coordinates and 3D points \([x,y,z]\). Initially a few of the grid points are constrained to given fixed locations in 3D space, the rest are unknown.

For each pair of known fixed points a catenary is draped between them with length proportional to the distance between them in the \([u,v]\) plane. The grid points that lie between the known fixed points are then constrained to equally spaced positions along the catenary curve. Where two catenary curves cross the same grid point the lower curve is removed to avoid trying to constrain the grid point in two different places.
Figure 1-2 - Original constrained points and the initial catenaries ([u,v] plane)

Figure 2-2 (a) shows the original constrained points as black squares, the constrained points after the initial catenaries have been traced are shown in (b). The algorithm then recursively traces threads from triangle vertexes to the midpoint of the opposite edge, as shown in figure 2-3, until all the grid points within the convex hull are constrained.

Figure 1-3 - Recursive subdivision of Triangles on grid

The points that lie outside the convex hull of the initial constrained points are set according to the desired effect, for example to simulate a hanging effect they are set to –infinity. The next step of the algorithm is the iterative relaxation process, which will reposition these points to their desired positions.

The Iterative Relaxation Process

The surface is refined using the iterative relaxation process described by M. B. Hsu where for each grid point \( x_i \) its neighbours are considered fixed and the nodal stiffness matrix \( k_i \) is generated for that point. The equation

\[
k_i \cdot u_i = F_i
\]

is then solved to find \( u_i \). \( F_i \) is the force vector for node \( i \) which includes factors such as the weight and any other forces applied to that point. \( u_i \) is the displacement vector that moves \( x_i \) to its new equilibrium position i.e.

\[
x_i(\text{new}) = x_i + u_i
\]

The relaxation process is repeated until displacements fall below a certain threshold, or a certain number of iterations are reached.

This algorithm is of order of the number of grid points * the number of iterations required by the relaxation process. The first step of the algorithm helps reduce the number of iterations required to reach equilibrium state and the computations required at each step are relatively simple. Although I have no execution timings for this model, I estimate that the model is reasonably fast and should be able to produce real-time results for reasonable grid sizes. The
algorithm is only suitable for simulating static images of rectangular pieces of cloth draping from fixed points. This model attempts to preserve the surface area of the cloth by adding the constraint that points that are within two grid points from each other must be less than a maximum distance apart in 3D space. This is not physically accurate but sufficient to prevent significant stretching effects on the surface leading to reasonable visual results.

**Terzopoulos et.al. – ‘Elastically Deformable Models’**

This paper introduces a generalised deformable model for flexible objects based on elasticity theory. The model can be applied to curves, surfaces, solids or even objects of higher dimensions. For cloth modelling we are interested in surfaces so I will specifically present this case. This model is categorised as a Physical, elastic continuum based approach.

Given a parametric representation of a surface O (such as a B-Spline representation) the shape of the surface in its natural undeformed state is given by the vector-valued function

\[ \mathbf{r}(a_1 \mathbf{a}) \]

where \( a = [a_1, a_2] \) are the intrinsic coordinates of the surface. This model models the progressive deformation of the surface which is given by the time varying function

\[ \mathbf{r}(a, t) \]

The dynamics of this deformable model are governed by the Lagrange equation of motion;

\[ \frac{\partial}{\partial t} \left( \frac{\partial \mathbf{r}}{\partial a} \right) - \frac{\partial \mathbf{f}}{\partial t} = \mu \frac{\partial}{\partial a} \left( \frac{\partial}{\partial t} \mathbf{r} \right) \]

Eqn 1-1

where \( \mu \) is the mass density and \( \mathbf{f} \) the mass density. These can vary over the surface so they are functions of \( a \). \( \mathbf{f} \) is the net externally applied force, i.e. the sum of all the individual external forces. The first term in equation 2-1 gives the inertial force due to the bodies mass and the second term the damping force due to dissipation. \( e(\mathbf{r}) \) gives the net instantaneous potential energy of the body, its variational derivative \( \frac{\partial e}{\partial \mathbf{r}} \) gives the elastic force.

The net instantaneous potential energy or strain energy represents how much the body has deformed from its original shape. To develop an expression for \( e(\mathbf{r}) \) Terzopoulos used the first two fundamental forms of the surface. The fundamental theorem of surfaces states that two surfaces are identical apart from rotations or translations if their metric tensors \( G \) and their curvature tensors \( B \) are identical functions of \( a \). This is what we need since deformations of the surface away from its natural shape will change \( G \) and \( B \), but rigid motions that do not deform the surface will not. Using this fact Terzopoulos defined the strain energy as

\[ \frac{1}{2} \mathbf{G}^0 \mathbf{B}^0 \mathbf{G} \mathbf{B} \mathbf{G}^0 \mathbf{B} \mathbf{G}^0 \mathbf{B} \mathbf{G} \mathbf{B} \]

Eqn 1-2

Where \( \mathbf{G}^0 \) and \( \mathbf{B}^0 \) are the fundamental forms of \( r^0 \) and \( \mathbf{B} \) are weighted matrix norms. When the surface is in its natural state \( G = G^0 \) and \( B = B^0 \) so the stain energy is zero as we would expect.

The net externally applied force is the sum of various external forces acting on the surface. These can include gravity, forces due to a spring attached to a surface point and an external point, force due to a viscous fluid (i.e. wind). Example equations for calculating these forces are given in his paper. Terzopoulos also explains how to use a potential energy field around impenetrable external objects to simulate collision dynamics. The resulting force due to a collision is given by

\[ f_{\text{collision}} \]

\[ \frac{2}{3} e^{-\text{exp}} \frac{f(\mathbf{r})}{e^2} \frac{\mathbf{h}}{\mathbf{h}^2} \]
where \( \mathbf{n}(\mathbf{a}) \) is the unit normal to the surface at \( \mathbf{a} \), \( f() \) the object inside/outside function and \( e \) a constant parameter defining the shape of the potential field. Self-collisions can be handled by surrounding the surface itself with a potential field.

In order to solve these equations Terzopoulos et.al. discretized Eqn 2-1 in space, first simplifying Eqn 2-2 as follows:

\[
\begin{align*}
\sum_{i,j} e \phi_i \phi_j \mathbf{G}_{ij} \mathbf{G}_{ij}^\top \mathbf{B}_{ij} \mathbf{B}_{ij}^\top \, da_i da_j
\end{align*}
\]

where \( \phi_i \) and \( \phi_j \) are weighting functions of \( \mathbf{a} \).

The first variational derivative of this can apparently be approximated by

\[
\begin{align*}
e \frac{\partial}{\partial \mathbf{a}_i} \frac{\partial}{\partial \mathbf{a}_j} \phi_i \phi_j \mathbf{G}_{ij} \mathbf{G}_{ij}^\top \mathbf{B}_{ij} \mathbf{B}_{ij}^\top \, da_i da_j
\end{align*}
\]

with

\[
\begin{align*}
\phi_i \frac{\partial \mathbf{a}_i}{\partial \mathbf{a}_j} \phi_j \mathbf{G}_{ij} \mathbf{G}_{ij}^\top \mathbf{B}_{ij} \mathbf{B}_{ij}^\top
\end{align*}
\]

\( \phi_{i1} \) and \( \phi_{i2} \) are the resistances to length deformation in each principle direction and \( \phi_{12} = \phi_{10} \) determine the shear resistance. \( e_{11} \) and \( e_{22} \) determine the resistance to bending deformation and \( e_{12} = e_{21} \) is the resistance to twist deformation.

The equations are discretized by finite difference or finite element approximations giving rise to a set of linked ordinary differential equations. The dynamics are simulated by integrating the equations over time using an implicit procedure.

Terzopoulos et.al. give examples of their deformable dynamic model in action including sequences depicting a surface being lifted from one of its corners by a spring, a ball landing on a deformable solid cube, a membrane shrink wrapping a jack, a flag waving in wind and a carpet falling over objects. The images of cloth however appear too elastic, as the model is based around the theory of continuous elastic sheets, the model is more valid for simulating materials like plastic, rubber as they only have structure at a very small scale. Cloth has much more structure at a much higher scale leading to the buckling and folding behaviour of cloth that is not well simulated by this model. A particular problem of this model is that it will always deform back to its natural state when forces are removed, cloth however will tend to suffer non restoring deformations. The order of this algorithm is dictated by the order of the finite difference discretization method used, and the number of time steps required between frames. Terzopoulos et.al. mention that work in progress is a more sophisticated integration method using finite element methods.

Terzopoulos and Fleischer. – *Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture*  

This paper draws together two models, the one described above and a variation on this model described in a second paper by Terzopoulos and Witkin, *Physically based models with rigid and deformable components*. This work aims to provide a unified model that can simulate elastic, viscoelastic, plastic and tearing behaviour. Viscous material is fluid, it deforms without any restoring force. Elastic material on the other hand deforms under applied force, but restores itself back to its original shape after deformation. Viscoelastic material exhibits characteristics of both, a good example given in the paper is Silicon Putty. Silicon Putty flows under sustained force, but bounces under sudden impulse forces.

Terzopoulos et.al. simulate linear elastic, viscous and plastic deformation with Uniaxial deformation units, see figure. 1-4.
The stress-strain relationships for the deformation units are given by

- **Elastic unit (Hookean Spring)**
  \[ f = ke \]

- **Viscous unit**
  \[ f = \beta \dot{e} \]
  where \( \dot{e} \) denotes the time derivative of strain

- **Plastic unit**
  This will arbitrarily elongate or contract as soon as the applied force exceeds a yield force.

These units are combined in certain configurations in order to simulate various behaviours.

- **Viscoelastic Model**
  A four-unit assembly called the ‘Maxwell Voight’ viscoelastic model, models linear viscosity.

  \[
  f = \begin{cases}
  b_1 f & \text{if } f > b_1 f \\
  b_2 f & \text{if } f \leq b_1 f
  \end{cases}
  \]

- **Elastoplastic model**
  During plastic deformation the material properties change, \( k \) and \( ? \) become smaller. This is called elastoplasticity and is modeled by the following configuration.

- **Fracture**
  This is modeled by placing an upper bound on the displacement in the above, causing the unit to give way under a certain amount of force.
In Terzö the ideas of Terzó (primary formulation) are extended to explicitly model rigid body motion (hybrid formulation). This is done by replacing the position function $\mathbf{r}(\mathbf{a}, t)$ with a two component model with a separate reference and deformation component;

$$\mathbf{q}(\mathbf{a}, t) \rightarrow \mathbf{r}(\mathbf{a}, t) + \mathbf{e}(\mathbf{a}, t)$$

Where $\mathbf{q}$ gives the position of point $\mathbf{a}$ on the body, at time $t$. $\mathbf{r}$ is the reference component and describes the position of the object due to rigid-body motion. $\mathbf{e}$ describes elastic deformation of the object, with respect to the reference component $\mathbf{r}$. This allows easy modeling of rigid body motion combined with elastic deformation.

In this paper [Terzé] explains how to incorporate inelastic behaviour into the primary and hybrid models. In the case of the hybrid model, viscoelasticity is described by evolving $\mathbf{r}$. During each time step a proportion of the instantaneous elastic displacement $\mathbf{e}$ is transferred into $\mathbf{r}$, thus loosing its restoration energy. In the case of a simple Maxwell unit depicted above, $\mathbf{e}$ plays the role of the spring and $\mathbf{r}$ the role of the viscous unit and the evolution is described by

$$\frac{d}{dt} \mathbf{r}(\mathbf{a}, t) = \frac{1}{\alpha} \mathbf{e}(\mathbf{a}, t)$$

This can be extended for the case of the full viscoelastic unit depicted above and parameters can be dynamically adjusted so as to be able to simulate more complex behaviours such as fracture.

As before LaGrange’s equation of motion and an elastic potential function $\mathbf{e}$ is used, to generate a large system of differential equations that are discretized using numerical techniques, and are integrated over time to simulate the dynamics of the model.

Examples are given of a ‘plasticine’ model of a bust deformed by a virtual hand, when the hand is removed we see the model partially restore towards its original shape we also see the permanent deformed due to viscous effects. A second simulation shows a falling cloth object as it hits a impenetrable sphere and tears as the deformation exceed the materials elastic limit.

This model is more capable for simulating the dynamics of cloth than the pure elastic approach, as it allows the modelling of deformations without restoring force. However, it is still based on continuous surfaces, which unlike cloth do not exhibit structure at a relatively large scale. However the results, especially those of tearing do appear visually realistic simulations of cloth.

This model is far more complicated than the pure elastic approach, with many more parameters generating a larger set of equations. The large number of parameters also lead to difficulties with instabilities in the solution of the equations, forcing small time-steps.

**D. E. Breen et.al. – “A Physically-Based Particle Model of Woven Cloth”**

Acknowledging that cloth is not a continuous sheet but a complex mechanical structure, in a series of papers Breen et.al. model cloth at the thread-crossing level. Each thread-crossing in a woven fabric is treated as particle, forming a grid of particles 4-connected in warp and weft directions. The energy of particual $i$, is given by

$$U_{total} = U_{repel} + U_{stretch} + U_{bend} + U_{trellis} + U_{gravity}$$

where $U_{repel}$ is an artificial repulsion energy included to help prevent self intersection of the surface, $U_{stretch}$ is tensile strain energy between particle $i$ and its 4-connected neighbours. $U_{bend}$ is the energy due to threads bending out of their local plane. $U_{trellis}$ is the gravitational potential energy of the particle. $U_{gravity}$ is the energy due to bending around a thread-crossing in the plane. They formulate these energy functions as follows.

?? Stretching and Repelling Energy

The aim of these functions it to try and keep 4-connected particles a distance $s$ apart, and the repulsion also helps prevent self-intersection. The repelling energy of a point $i$, is found by summing the individual repelling forces between particle $i$, and all the other particles $j$. The equation is given below.
\[ U_{\text{repel}} = \sum_{j \neq i} \frac{R_{ij}^2}{r_{ij}^2} \]

where \( R_{ij} = \frac{C_0}{r_{ij}} \), \( r_{ij} = \frac{r_i + r_j}{2} \), and \( r_{ij} \) is the distance between point \( i \) and point \( j \). Similarly, the stretch energy is given by

\[ U_{\text{stretch}} = \sum_{j \in N_i} S_{ij} \]

where \( S_{ij} = \frac{0, r_{ij} \leq 0}{\frac{C_0}{\mu} \frac{s}{r_{ij}} \frac{0}{s} \frac{r_{ij} \leq s}{s}} \)

where \( N_i \) is the set of 4-connected neighbours of particle \( i \).

?? Gravity

This is simply defined as \( U_{\text{gravity}} = m_i g h_i \) where \( m \) is the mass of the local area of fabric represented by particle \( i \), and \( h_i \) is the height of particle \( i \).

?? Bending and Trellising Energy

\[ U_{\text{bend}} = \sum_{j \in M_i} B_{ij} \]

\[ U_{\text{trellis}} = \sum_{j \in K_i} T_{ij} \]

where \( \theta_i, \theta_j \) are the angles between point \( i \) and the set of 8 neighbouring points \( M_i \), \( \theta_{ij} \) are the trellising angles between \( i \) and its set of 4-connected neighbours \( K_i \). Breen et.al. show how to tune the functions \( B \) and \( T \) so that they best fit experimental data produced from the Kawabata System\(^{11}\). This is developed further in a second paper, I refer to Breen et.al. “A particle-Based Model for Simulating the Draping Behaviour of Woven Cloth”\(^{12}\) for more details.

The simulation then proceeds in two stages:

1. The particles are allowed to fall under the effect of gravity, free from interparticle constraints until they hit objects in the scene.
2. Interparticle constraints are reinforced and each particle’s position is adjusted so as to achieve a local energy minimum.

Figure 2-5 shows this model in action, the image is taken from David Breen’s website. This model is only suitable for static images. It aims to be very physically accurate by modeling cloth at the thread crossing level and linking the models parameters to actual data taken from experiments with real cloth. The time taken for a simulation of a rectangular piece of cloth with 51x51 particles, is quoted as 1 CPU-week on an IBM RS/6000. This makes the model incapable of real-time results. In Eberhardt et.al.\(^{13}\) various optimisations are made to Breen’s model including the use of higher-order explicit integration methods. This lead to faster simulation of about 20-30mins per frame on a SGI R8000.
Baraff and Witkin. – “Large Steps in Cloth Simulation”

This paper makes a significant step forward in improving the performance of physically-based cloth modeling. Previously physically based models used explicit numerical integration techniques. Due to the ‘stiffness’ of the underlying differential equations of motion, many small time steps are required in order to maintain stability. Baraff and Witkin introduce an implicit integration scheme that allows for much larger time steps to be taken, reducing the number of required time steps to 2-3 per frame of animation regardless of the models size. Explicit methods normally require of the order of \(n\) time steps, where \(n\) is the size of the model. This makes them at least \(O(n^2)\) in total.

Witkin and Baraff give timings for a similar simulation to Breen’s above. A cloth model of 2600 points is draped over a cube, with a running time of 2 seconds per frame on an SGI R10000. Timings for the deformation of a skirt (4530 points) a shirt (6450 points) deforming on a model of a dancing character are given. These show the model can achieve computation times of 8-14 seconds per frame.

Hadap et.al. – ‘Animating Wrinkles on Clothes’

In this technique, a coarse mesh is used to model the large-scale deformation of a garment. A bump map is then applied to the deformed surface in order to add the small-scale detailed wrinkles on the clothing that are not possible to model through the course mesh. This bump map is predefined by the animator. As a triangle from the course mesh stretches, one would expect the wrinkles within that triangle to flatten, preserving the surface area of the cloth. This model modulates the bump map in order to achieve this effect.

This model requires the animator to provide a bump map for the clothing object and does not simulate the progressive formation of wrinkles. However, visual results are pleasing and the model can achieve 20 frames per second on a MIPS R10000.
Chapter 2 – The Clothing Model of Nikitas Tsopelas

An analysis of Nikitas Tsopelas’s clothing model.

Tsopelas presented a unified deformation algorithm that aims to simulate the deformation of a broad class of ‘thin-walled’ objects. Thin-walled structures are objects whose thickness is small compared to its other dimensions, this includes cloth and rigid structures such as metal tubing.

Tsopelas’s model uses a Hybrid technique to model the deformation of thin-walled objects. Unlike pure physical models, his model does not require any expensive iterative relaxation processes or solutions of large systems of partial differential equations. This results in a dynamic model that aims to be both quick and realistic in simulating cloth, but it does not aim to be completely physically accurate. The model is dynamic and can simulate the progressive deformation of folds on a cloth surface, thus allowing it to be used to generate animations.

Background.

From engineering and textile literature, Nikitas studied the buckling patterns that appear on two types of rigid thin-walled structures, perfect cylinders and perfect square tubing. The deformations that he studied were those formed by axial loads (forces in the direction of the axis of the cylinder). Clothes are made up of cylinder and square tubing like objects, which makes the deformations he studied a suitable basis for clothing modeling. It is not however, suitable for modeling the general drape of fabric. Cloth is in fact not a rigid structure but a flexible one, the main difference being in the loads that initiate the collapse of the object. Rigid structures require applied loads in order collapse whereas flexible structures can collapse under their own weight. Nikitas focused on rigid thin-walled structures because at the time little research had been done on the deformation of flexible thin-walled structures. However his original assumption that rigid and flexible structures would deform in the same way was validated by the work of Shinohara et.al. This importantly justifies the use of this model for clothing modeling.

Geometric characteristics of the buckling patterns.

What is important to this model are the macroscopic geometric characteristics of the buckling formations so they are examined in detail, in view to geometrically reconstructing them on a surface later. There are two cases that require separate analysis, inextensional and extensional buckling. When the material has a greater resistance to stretching than bending, inextensional buckling occurs, conversely extensional buckling is observed on ‘elastic’ materials which can easily stretch.

Inextensional buckling of a flexible cylindrical tube

The geometric characteristics of the patterns that appear in this case depend mainly on the thickness to diameter ratio. When this is above a certain threshold, an axisymmetric deformation occurs (bellow shapes). When t/D is below some threshold diamond shaped patterns of lobes (folds on the surface) are formed. However, this behavior is also effected by the structure of the material, it has been observed that woven and leather fabrics form diamond shaped buckling patterns whilst knitted fabrics form bellows.
Figure 2-2-1 - Diamond Shaped Lobes

The number of lobes formed is independent of the length of the cylinder, the theory states that the number of lobes $n$ in the circumferential direction is given by

$$ n \approx \frac{1}{2} \frac{D}{t} \frac{1}{\sqrt{1 - \frac{6t}{D^2}}} \quad \text{Eqn 2-1} $$

But from real experiments on garments the following relation better fits observed data.

$$ n \approx 1.44 \ln \left( \frac{t}{D} \right) + 1.75 \quad \text{Eqn 2-2} $$

Figure 2-2-2 - Comparison of the two formulations for the number of lobes formed. On the left is Eqn 2-1, on the right, Eqn 2-2.

We see that thicker specimens will form less lobes than thin ones, which we can observe in real life when comparing, for example, silk garments with leather.

?? Buckling wavelength ?

The buckling wavelength is the height of the diamonds (see Figure 2-1) that form on the surface. This depends on the material and is given by
For woven cloth, $C_f$ is around 1.7, whereas for metallic tubes $C_f$ is 0.953

**Extensional buckling.**

Some materials such give little resistance to a small compression or tension in the circumferential direction. Because of this, they exhibit extensional axisymmetric buckling (symmetrical about the main axis). This mode of buckling forms below shapes where the bellows follow a sinusoidal shape

The wavelength $W$ of the buckling is given by,

$$W = \frac{3.38}{t D}$$  \hspace{1cm} \text{Eqn 2-4}

where $t$ is the thickness of the material, $r$ the radius of the cylinder, and $E1$ and $E2$ are the Hooke’s moduli in the axial and circumferential direction.

**Square Tubes**

Nikitas then goes on to examine the buckling of square tubes in similar detail. He concludes that the mode of collapse of a square tube depends on the ratio $c/t$, where $c$ is the length of a side and $t$ the wall thickness. These three modes of deformation are observed:

- $c/h > 40.8$ Symmetric collapse mode.
- $0.75 < c/h > 40.8$ Asymmetric collapse mode.
- $c/h < 0.75$ Extensional collapse mode.

Extensional collapse mode very rarely occurs with square tubing. Figure 2-3 shows the symmetric and asymmetric collapse modes. In asymmetric collapse mode A, at layer I three lobes deform inward, whilst one deforms outward. In Layer B, three lobes deform outward whilst one deforms inward. In Asymmetric collapse mode B, all four lobes of Layer I deform inward whilst two adjacent lobes deform inward and two outward in Layer II. The formulas for the height of the collapsed lobes are

$$H = \frac{3.38 c}{h}$$  \hspace{1cm} \text{Eqn 2-5}

Where $C_m$ depends on the collapse mode

- symmetric collapse mode
  \[ C_m = 0.99 \]
- asymmetric collapse mode A
  \[ C_m = 0.73 \]
- asymmetric collapse mode B
  \[ C_m = 0.83 \]
- extensional collapse mode
  \[ C_m = 0.883 \]
Generalization for garments.
The models above can be used to reproduce exactly the geometrical deformation patterns of perfect cylinders and square tubes of cloth under axial compression. Clothing however, is not made up of perfect cylinders or tubes. But some forms of clothing, for example a shirt, are made up of cylinder like sleeves and a square tube like body. In his model Nikitas makes the assumption that they will also deform in a similar manner as the perfect cases forming the diamond and below shaped patterns as described above. In order to make this generalisation realistic, a few extra assumptions about the axial deformation of surfaces with non-uniform cross-sections are needed. Nikitas does not mention where these facts came from.

- The collapse of thin-walled structures begins at the areas of high absolute maximum principle curvature, where the direction of absolute maximum curvature is perpendicular to the applied force.
- Lobes will begin to form in these areas then expand laterally until they meet other lobes.
- With cloth, creases are persistent, they will always tend to appear in the same place. (Ironing or dry cleaning tend to reset these persistent patterns)

These deformation patterns form the basis of Nikitas’s deformation algorithm described in the next section.

The Deformation Algorithm
The input is a collection of surface panels (usually in B-Spline representation) which make up the clothing object. These can be either from a scan of real clothes or from a computer-generated model of a clothing object from a CAD system. The clothes can already show deformations.

Typical input for the case of a shirt would be four panels, two tube-like sleeves and the front and back panels of the shirt body. The algorithm is a two-stage process, first the input surfaces are interrogated to extract geometric features that are used to generate a model of the surfaces. This model characterises already existing deformations and the deformation patterns that are likely to appear. The second stage then simulates deformations based on this model.

1) Converting the surface representation into a model based on ‘Elasticas’
- Identify which surface panels will deform as one.
- The panels of the clothing object are examined to identify areas that will deform as one, e.g. across the seams between the front and back panels of a shirt. Buttons, such as those on the front of a shirt can be considered join panels into a continuous region. However, as I will discuss later, the model could be enhanced to model this more accurately.
- Interrogate surface for existing deformations and where new lobes are likely to form.
Here we calculate the principle curvatures $k_1$ and $k_2$ of the surface. In order to do this we need access to the first and second derivatives of the surface which are easy to obtain from a B-Spline representation of the surface. The features we are interested in are extrema and zero-crossings of $k_1$ and $k_2$. The extrema are found by thresholding principle curvature values (in the implementation the threshold is set to 10% of the largest absolute value) and applying a nonmaximal suppression in the direction of the absolute maximum principle curvature. This direction should be perpendicular to a fold or edge on the surface.

These extrema are classified into six types of surface points
1. Single extremum of $k_1$
2. Single extremum of $k_2$
3. Extremum of $k_1$ associated with a zero-crossing of $k_2$
4. Extremum of $k_2$ associated with a zero-crossing of $k_1$
5. Extremum of $k_2$ that lies between two type 3 points
6. Extremum of $k_1$ that lies between two type 4 points

Along with each surface point, the tangent direction of the principal curvature line is also stored. In the next step these surface points of interest are used to identify three types of surface feature: single buckles, collapsed lobes and areas where new lobes will form.

Nikitas does not explain what ‘associated with zero-crossings’ means, but I presume it refers to the way a buckle causes a surface inflection, giving rise to a zero-crossing in principle curvature. Detection of this zero-crossing near our extrema, lets us identify buckles already existing on the surface.

![Figure 2-4: Extrema (Yellow) associated with zero-crossings (Red)](image)

Detect existing buckles and surface edges where new buckles will form.

Adjoining points of the same type are grouped together if the orientations of their principle curvature are compatible. See Figure 2-5
Figure 2-5 - Detecting surface features from extrema points. Nonmaximal points (purple) are suppressed along the curvature lines (green), leaving only the local maxima (in red). These are all the same type and their tangent directions (along yellow line) are compatible so they are grouped along the interpolating curve (yellow line) to form a surface feature (hinge).

New lobes will be formed in areas displaying extrema of curvature in a direction perpendicular to the applied force. As we are only considering axial compression, this direction is perpendicular to the curvilinear axis. Thus, if a group of Type-1 or Type-2 points form an interpolating line that is parallel to the curvilinear axis (±10°), then we have found an area where a new lobe will form. This is called an edge and corresponds to the virtual diagonal of the lobe that will form.

When we have a group of Type-3 or Type-4 surface points, whose interpolating line is not parallel to the curvilinear axis then we have found an existing buckle. This is called a single hinge.

Similarly a group of Type-5 or Type-6 surface points whose interpolating line is not parallel to the curvilinear axis indicates that we have found an existing lobe.

Partition the surface into regions
The surface areas where existing lobes have been detected are considered separate regions. The remaining surface is partitioned into regions around the detected edges. Curvature lines in the circumferential direction are processed and areas of local minimum in curvature are selected and interpolated to form the borders of regions. Border curves are generated by $G^2$ interpolation using a piecewise Bezier scheme in the parameter space of the border points.

Determine the buckling mode for each region.
The Gaussian and mean curvature of region points are used to determine the buckling mode for each region.

If the Gaussian and mean curvatures are close to zero for the majority of points within the region except in the vicinity of existing edges, then a rectangular like surface with planar edges is detected. The buckling modes of square tubing are then assumed for this region, with the edges forming region borders. The height of the lobes formed $H$ is obtained from Eqn 2-5, with $c$ the mean width over the region.

If the majority of points within the region are parabolic, hyperbolic or elliptic (Non zero Gaussian curvature, or only one of the principle curvatures is zero) then a curved region is detected and the buckling modes as for a cylinder are assumed. The buckling mode is determined as in the perfect cylinder case above, and the wavelength is calculated from Eqn 2-2 or 2-3, but instead using the mean average diameter as the diameter for the region.
Place hinges on the surface

Hinges are placed on each undeformed region according to the regions buckling mode and wavelength. Nikitas’ system allows the animator to add new hinges and modify hinge placement.

If the buckling mode is axisymmetric, then hinges are placed on the surface at equal spacing of $2H$. In the case of asymmetric buckling mode, i.e. diamond shaped lobes, hinge points are placed along the border curves at a spacing of $\frac{\pi}{2}, \frac{3\pi}{2}, \frac{5\pi}{2}, ...$ and in the middle of the region at spacing $0, 1\frac{\pi}{2}, ...$ as shown in Figure 2-6.

![Figure 2-6 - Hinge placement according to collapse mode](image)

Generating the ‘Elastica’ model.

The model consisting of a collection of regions combined with a description of the hinges on the region is then converted into a model based on Euler elasticas. This is the model upon which deformations will be physically modelled. Rather than using a mass-spring based model to connect hinges, Euler elasticas are used, as they give the deformed position of the hinges and a good approximation of the shape of the surface between them.

The elastica model is generated by tracing ‘Threads’ along the surface in the direction of the curvilinear axis. The threads are then split up into elasticas at the hinge points as shown in Figure 2-7. There are always two outer-threads, one near each border, an inner-thread that intersects the midpoint vertices of the hinges, and one or more mid-threads that lie in between. The longer elasticas connect together the short elasticas that lie over the hinges. The length of the elasticas that lie over hinges effects the ‘sharpness’ of the crease generated. Short elasticas giving sharp creases like those seen on leather. For silk or cotton fabrics the elastica lying on hinges should be longer.
Figure 2-7 - The tracing of threads and splitting into Elasticas in Asymmetric mode

A similar technique is used to trace threads and elasticas on the surface of planar region with axisymmetric buckling patterns. In this case, any number of threads are traced at even intervals across the region, and split up into elasticas as shown in Figure 2-8.

Figure 2-8 - Tracing Threads and Elasticas in Axisymmetric mode

The deformation mechanism

We now focus on the mechanism by which our elastica model will deform.

The Euler elastica.

The Euler elastica is a model of how elastic rods buckle under axial forces. I will not go through the derivation of the equations that govern the deformation of an elastica as this is beyond the scope of the project. For a detailed study of elastica theory see Timo61 and a study
of how elasticas can be used to model the collapse of fabric threads is given in Grossberg et.al “The mechanical properties of woven cloth, Part III The Buckling of woven fabrics”17.

Figure 2-9 - Elastica deformation
Elastica deformation is governed by a second order non-linear differential equation that is solvable numerically. There are many versions of the elastica model, including many ways in which boundary conditions can be specified. Important to this model is the Inverse dynamics procedure as described by Nikitas in section 5.4.5 of his thesis. This is where the end-points of an elastica and its length are known, the forces that could cause an elastic beam of this length to deform to fit the prescribed end-points are searched, giving the final shape of the elastic beam. There are two kinds of end-point conditions for the inverse dynamics procedure, Fixed end-points, and Free end-points. Fixed end-points are constrained at a constant angle, Free end points on the other hand cannot transfer any torques so are free to rotate to whatever angle they like. Figure 2-9 shows the progressive buckling of the two cases of elastica. This image was generated from Nikitas’s code.

Displacing Elastica End-points
Deformation of the surface is modeled by displacing elastica end-points and then using the inverse dynamics method to find the new shape of the elasticas and hence the surface. The displacement for elastica end-points are generated by a number of possible methods discussed in the next section.

Surface Reconstruction
Each elastica now has an array of points describing its deformed shape. In order to reconstruct the surface, pairs of elastica points are grouped with equivalent pairs of elastica points on the neighbouring thread forming four sided surface polygons. As inner and outer threads have fewer elasticas than the mid-threads, some of the polygons connecting these threads are degenerated into triangles.
Chapter 3 - Analysis and Design

Preliminary stage – Compiling Nikitas’s Code

In the beginning, I was faced with the decision of whether to implement my own version of Nikitas’s model or to make improvements to his code, of which I had obtained a copy. Nikitas implemented his model on a Sun SPARCstation 2, using C and the primitive graphics functions of the X11 libraries. There was great potential for speed improvements just from compiling his code to run on newer machines and by converting his code to use Open GL. The massive size of his program, and the complexity of the model lead me to believe a complete reimplementation would take to much time so I decided to work on his code.

Before assessing his model and looking at making any improvements, my first task was to get his code compiled and running on my machine, a 700MHz Pentium II based PC running linux. The first problem associated with this was the fact that Nikitas’s code came with only a token of documentation, describing only a small section of his code so I had no idea what most of it did. In the next few pages I will briefly describe his data structures, what he actually implemented and the compilation process. In the following sections, I will then describe how I built upon this.

Directory Structure

The following describes the directories of interest from Nikitas’s code.

Nikitas\MODELER
- Nikitas’s program to design garments. Its main relevance to this project is that it generates the input files that are used by the deformation algorithm. These files represent NURBS surfaces, from now on I will refer to these files as NURBS files.
- Nikitas\MODELER\DisplayGarment
- Nikitas\MODELER\DisplayModel
- Nikitas\MODELER\DisplayPanel
- Nikitas\MODELER\OldSkeleton
- Nikitas\MODELER\Seaming
- Nikitas\MODELER\Shirt
- Nikitas\MODELER\Trousers

Nikitas\TWGCO
- This directory contains various subdirectories that each performs some aspect of the deformation algorithm.
  - Nikitas\TWGCO\1INTERG
    - Surface interrogation on a NURBS surface. The input is a NURBS file, the output is a file describing the threads and elasticas that have been traced on the surface. From now on, I will refer to this intermediary file as an Elastica file.
  - Nikitas\TWGCO\2INTCURV
    - More surface interrogation on NURBS files.
  - Nikitas\TWGCO\4SHD
    - The example images and timing given in the thesis.
  - Nikitas\TWGCO\Deformations
    - The input is a set Elastica files describing a clothing object, this program displays and performs deformations on these Tread-Elastica models.
  - Nikitas\TWGCO\dmcollapse
    - Simple example of a cylinder collapsing under its own weight.
  - Nikitas\TWGCO\dmpattern
  - Nikitas\TWGCO\ELAST
    - Demonstration of the Elastica code, gives the output shown in Figure 2-9

Nikitas\TWGCO\Patterns
- Performs Hinge placement and the tracing of threads and elasticas according to the patterns described in Chapter 2. It does not do any surface interrogation, as the input is assumed undeformed. The input is a NURBS file, the output an Elastica file.
File Formats
I will now describe the formats of the NURBS and Elastica files.

NURBS file
This is an ascii text file describing a NURBS patch as follows,

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Width</td>
<td>The width of the Patch grid (u points)</td>
</tr>
<tr>
<td>int</td>
<td>Height</td>
<td>The height of the patch grid (v points)</td>
</tr>
<tr>
<td>float*(width+5)</td>
<td>Ku</td>
<td>Knot values in u direction</td>
</tr>
<tr>
<td>float*(height+5)</td>
<td>Kv</td>
<td>Knot values in v direction</td>
</tr>
<tr>
<td>float, float, float <em>((width+5)</em>(height+5))</td>
<td>k(u,v)</td>
<td>Knot vectors listed in u then v.</td>
</tr>
</tbody>
</table>

Nikitas provides some example surface files created by his modeller package. These are for a jumper and two pairs of trousers.

BackShirt   Back shirt panel
FrontShirt  Front shirt panel
LeftSleeve  Left sleeve
RightSleeve Right sleeve

LeftTrouser Left trouser sleeve
RightTrouser Right trouser sleeve

Elastica file
The variable names given below correspond to the variable names in the data structures as described next.

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>index</td>
<td>Surface index</td>
</tr>
<tr>
<td>int</td>
<td>surftype</td>
<td>Surface type</td>
</tr>
<tr>
<td>int</td>
<td>pattype</td>
<td>Pattern Type</td>
</tr>
<tr>
<td>int</td>
<td>RegionsNr</td>
<td>Number of Regions</td>
</tr>
<tr>
<td>int</td>
<td>ThreadsNr</td>
<td>Number of Threads</td>
</tr>
<tr>
<td>int</td>
<td>InNr</td>
<td>Number of Elasticas (of inner thread)</td>
</tr>
</tbody>
</table>

Each Thread has an entry of the following form...

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>type</td>
<td>Thread Type</td>
</tr>
<tr>
<td>int</td>
<td>elsnr</td>
<td>Number of Elasticas in this thread</td>
</tr>
<tr>
<td>int</td>
<td>telsnr</td>
<td>???</td>
</tr>
</tbody>
</table>

Following each Thread is a list of the Threads Elasticas...

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>type3 and type4</td>
<td>Elastica type</td>
</tr>
<tr>
<td>int</td>
<td>deformed</td>
<td>Deformed flag of elastica</td>
</tr>
<tr>
<td>float</td>
<td>length</td>
<td>Length of Elastica</td>
</tr>
<tr>
<td>float</td>
<td>dist</td>
<td>Distance between elasticas endpoints.</td>
</tr>
<tr>
<td>float, float, float</td>
<td>a,b,c,d</td>
<td>Plane equation of Elastica</td>
</tr>
<tr>
<td>float, float, float</td>
<td>firstp</td>
<td>First point of Elastica</td>
</tr>
<tr>
<td>float, float, float</td>
<td>nm</td>
<td>Normal of elastica</td>
</tr>
</tbody>
</table>
Example files of this type, as generated by the patterns module are provided for the NURBS surfaces listed above. The important files are those that make up the clothing objects used in the following sections. These are given the following names:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackShirtOut</td>
<td>Back shirt panel</td>
</tr>
<tr>
<td>FrontShirtOut</td>
<td>Front shirt panel</td>
</tr>
<tr>
<td>LeftSleeveOut</td>
<td>Left sleeve</td>
</tr>
<tr>
<td>RightSleeveOut</td>
<td>Right sleeve</td>
</tr>
<tr>
<td>LeftTrouserOut</td>
<td>Left trouser sleeve</td>
</tr>
<tr>
<td>RightTrouserOut</td>
<td>Right trouser sleeve</td>
</tr>
<tr>
<td>LeftTrouser1Out</td>
<td>Left trouser sleeve</td>
</tr>
<tr>
<td>RightTrouser1Out</td>
<td>Right trouser sleeve</td>
</tr>
</tbody>
</table>

**Important Data Structures**

The data structures across the various code modules are similar, here I give details of some of the data structures used by the deformations program as I will build upon these later.

**coord**
float x, y, z 3D co-ordinates

**hcoord**
float x, y, z, w 3D Homogeneous co-ordinates

```
<table>
<thead>
<tr>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>float x, y, z 3D co-ordinates</td>
</tr>
<tr>
<td>hcoord</td>
<td>float x, y, z, w 3D Homogeneous co-ordinates</td>
</tr>
</tbody>
</table>
```

**ElasticaPt**
A point on an Elastica

```
<table>
<thead>
<tr>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hcoord</td>
<td>Elastica point</td>
</tr>
<tr>
<td>hcoord</td>
<td>Normal of the Elastica</td>
</tr>
<tr>
<td>float</td>
<td>Shading Value</td>
</tr>
<tr>
<td></td>
<td>A stored value used by the renderer for calculating shading</td>
</tr>
</tbody>
</table>
```

**Tile**
Over the surface, elastica points from neighbouring threads are grouped together by the surface reconstruction procedure to form surface polygons. These are stored as Tiles

<table>
<thead>
<tr>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Number of sides (3 or 4)</td>
</tr>
<tr>
<td></td>
<td>Pointer to vertices</td>
</tr>
</tbody>
</table>

**Elastica**
This describes an elastica, including the array of calculated elastica points.

```
<table>
<thead>
<tr>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Type of boundary conditions to be used:</td>
</tr>
<tr>
<td></td>
<td>1: free-free ends</td>
</tr>
<tr>
<td></td>
<td>2: fixed-fixed ends</td>
</tr>
<tr>
<td></td>
<td>3: free-fixed ends</td>
</tr>
<tr>
<td>int</td>
<td>Type of orientation in space:</td>
</tr>
<tr>
<td></td>
<td>1: horizontal elastica</td>
</tr>
<tr>
<td></td>
<td>2: vertical elastica</td>
</tr>
<tr>
<td>int</td>
<td>Type of way it is going to buckle:</td>
</tr>
<tr>
<td></td>
<td>type3, type4 = 1: negative coords</td>
</tr>
<tr>
<td></td>
<td>type3, type4 = 2: positive coords</td>
</tr>
<tr>
<td>int</td>
<td>A flag indicating if the elastica is deformed or not:</td>
</tr>
<tr>
<td></td>
<td>0: undeformed</td>
</tr>
<tr>
<td></td>
<td>1: deformed</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Type of boundary conditions to be used:</td>
</tr>
<tr>
<td></td>
<td>1: free-free ends</td>
</tr>
<tr>
<td></td>
<td>2: fixed-fixed ends</td>
</tr>
<tr>
<td></td>
<td>3: free-fixed ends</td>
</tr>
<tr>
<td>int</td>
<td>Type of orientation in space:</td>
</tr>
<tr>
<td></td>
<td>1: horizontal elastica</td>
</tr>
<tr>
<td></td>
<td>2: vertical elastica</td>
</tr>
<tr>
<td>int</td>
<td>Type of way it is going to buckle:</td>
</tr>
<tr>
<td></td>
<td>type3, type4 = 1: negative coords</td>
</tr>
<tr>
<td></td>
<td>type3, type4 = 2: positive coords</td>
</tr>
<tr>
<td>int</td>
<td>A flag indicating if the elastica is deformed or not:</td>
</tr>
<tr>
<td></td>
<td>0: undeformed</td>
</tr>
<tr>
<td></td>
<td>1: deformed</td>
</tr>
</tbody>
</table>
### Solution

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>t1,t2</td>
<td>poo</td>
</tr>
<tr>
<td>float</td>
<td>P,R,W,g</td>
<td>Load, resistance, weight and guess for the elastica solution</td>
</tr>
<tr>
<td>int</td>
<td>pnr</td>
<td>The number of points for the solution</td>
</tr>
</tbody>
</table>

### Thread

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>type</td>
<td>Type of Thread: 1: outer 2: mid 3: inner</td>
</tr>
<tr>
<td>float</td>
<td>load, weight, resist</td>
<td>Appears again here (but not used)</td>
</tr>
<tr>
<td>int</td>
<td>elsnr</td>
<td>Number of thread elasticas</td>
</tr>
<tr>
<td>int</td>
<td>telsnr</td>
<td>Number of lower thread elasticas (not used)</td>
</tr>
<tr>
<td>Elastica</td>
<td>*els[70];</td>
<td>Pointers to the Elasticas of the thread</td>
</tr>
<tr>
<td>Solution</td>
<td>*sls[70];</td>
<td>Pointers to the Solutions of the elasticas</td>
</tr>
</tbody>
</table>

### Surface

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>index</td>
<td>An assigned index to identify which surface – front-shirt back-shirt left-sleeve etc.</td>
</tr>
<tr>
<td>int</td>
<td>surftype</td>
<td>0: Open 1: Closes surface</td>
</tr>
<tr>
<td>int</td>
<td>pattype</td>
<td>Surface Pattern 0: Axisymmetric 1: Diamond pattern</td>
</tr>
<tr>
<td>int</td>
<td>RegionsNr, ThreadsNr, InNr</td>
<td>The number of Regions, Threads within a region and in the number of Elasticas in an Inner-thread</td>
</tr>
<tr>
<td>Thread</td>
<td>*Threads[50]</td>
<td>Pointers to the Threads of the Surface</td>
</tr>
<tr>
<td>Tile</td>
<td>*Tiles[1000][1000]</td>
<td>Pointers to the tiles used for reconstruction of the surface.</td>
</tr>
</tbody>
</table>

### What Nikitas’s implemented in his code

As mentioned in his thesis, Nikitas’s code is not a complete implementation of his deformation algorithm, rather a demonstration of some of the main features. It is unclear from the text what had actually been implemented so I examined and compiled sections of his code in order to find out. The important parts of the code are the Interrogation, Patterns and Deformations modules.

### Surface interrogation

Surface interrogation as described in chapter 2 has been implemented. There are two simple input files used to demonstrate this section, however, there are no examples of output files provided.
In order to compile this section of code some changes needed to be made to the \texttt{xgr.c} and \texttt{xi.c} files. These files provide the user interface and graphics functions used by the program. Small changes in the way graphics contexts are handled by X meant they didn't work under later releases of the X windows libraries. My first task was to fix these problems, details of which are given in the next chapter. However, I soon found that this section of code very rarely terminates correctly, if at all.

Patterns
This package bypasses the interrogation stage and goes straight into the placing of hinges and tracing of Threads and Elasticas on an undeformed surface. The input is a NURBS file, the user inputs the thickness of the surface, the number of regions and the desired pattern (axisymmetric or diamond). The buckling wavelength is then calculated for the surface and hinges placed appropriately. In this implementation an indicator is set in the header file that determines which file is being input, (\texttt{FrontShirt}, \texttt{LeftSleeve} etc) and for these surfaces, predetermined values for the buckling wavelength and pattern are used instead of the calculated ones.

This program has similar \texttt{xi.c} and \texttt{xgr.c} files as the Surface interrogation program and similar changes were needed in order to get it to compile. Additionally, a few other small bugs needed fixing, they were possibly due to changes in the compiler.

Deformations
This program takes a set of Elastica files that make up a clothing object. For example, a typical set of input files would be \texttt{FrontShirtOut}, \texttt{BackShirtOut}, \texttt{LeftSleeveOut}, \texttt{RightSleeveOut}. It is actually important that they are listed in this order as a somewhat simplistic approach to ‘seeming’ is taken which assumes the first two surfaces will be seemed together along their edges. This is done by constructing a row of \texttt{Tiles} between them. (Placing them in the wrong order leads to the generation of random features such as the shirt body being seamed to the length of a sleeve.) There is no such seeming implemented between sleeves and the shirt body however.

The deformation program contains the files \texttt{control1.c} to \texttt{control10.c}. Each of these files corresponds to one of the example Elastica files and conditions each elastica of the file with its Weight, Load, Resistance and the guess for its solution. In most cases, this is ignored and a uniform conditioning is used over all the elasticas, so prior knowledge of the surface structure is not necessary needed here.

The very simple case of bending a joint by rotation and translation of elastica points is implemented. Prior knowledge of the structure of the Elasticas on the surface is required, as the centre and plane of rotation is determined from an Elastica, which is identified through Thread and Elastica numbers. This also makes the assumption that uniform patterns have been traced on all regions of this surface, thus making this program unsuitable for output generated by the surface interrogation package.

Once the elasticas have been moved to their new positions and conditioned, the elastica equation is solved for each elastica. The surface is then reconstructed and displayed, again surface reconstruction assumes uniform tracing of Threads and Elasticas.

There is currently no facility for automated generation of animation sequences by this program, but it has been written in such a way that this feature can easily be added.

As with the other two programs this section of code needed a few changes in order to get it to compile. The major changes were in \texttt{xi.c} and \texttt{xgr.c}. Full rendering of a frame with shading took a long time, of the order of minutes. The visual results were poor, due mostly to the simple rendering techniques used. In addition, physical accuracy was poor, bending of a sleeve for example is only visually pleasing for small angles, for larger angles the surface appears to self-intersect or stretch.
Summary
Various simplifications and optimisations have been made to the Deformation algorithm which make it only suitable for the uniform example files given (LeftSleeveOut etc.) and not a more general configuration of elastica that might be produced by the surface interrogation package. This means that the interrogation program is only useful as an example of how the interrogation stage could be implemented, it currently can not be easily incorporated into the rest of the package. The Patterns and Deformations programs do however, provide a complete clothing modeling system capable of modeling simple garment deformation due to bending of tube like surfaces. This bending usually corresponds to elbow, knee or hip joints and the package can easily be extended to produce simple animations of bending joints, thus, it is capable of producing simple animations.

Nikitas has not implemented the layered model for clothing a synthetic actor, any collision detection, or the database of elasticas that he mentions in his thesis.

In order to use this system, typically the following steps are required.

1) Generate surface panels of a clothing object in B-Spline form. This can be done with Nikitas’s modeller or a CAD package, output in NURBS file.
2) For each panel, use the Patterns program to trace Threads and Elastica files for each panel. Each surface is given an index so that the deformations program can identify them.
3) Identify where on the surface bending is required - for each joint, an Elastica on a thread needs to be identified to define the bending point and plane of rotation.
4) Set up the Deformations program with bending points, details of seaming, and elastica conditioning.
5) Supply joint angles to the Deformation program and run to obtain the output image.

Problems to be addressed
?? Rendering poor and slow
?? Unrealistic bending of joints
?? Surface interrogation has compilation and run rime problems.
?? Some parameters are ‘hard programmed in’ meaning recompilation is needed to change any of these parameters.
?? Only capable of simple deformations.
?? Far too uniform and ‘stiff’

Analysis of the BendThreads function.
The bending of joints is an important part of this modelling system, I now give a detailed analysis of how Nikitas implemented this. In the following sections, I will also analyse possible improved methods for performing joint bending.

The bending of joints is performed by the BendThreads() function defined in the file defthread.c within the Deformations program. This provides the displacements of elastica end-points and works as follows.

Five parameters are supplied by the user, \( f_1, f_2, f_3, f_4 \), and \( \text{angle} \). As shown in Figure 3-1, \( f_3 \) identifies the thread number and \( f_4 \) the elastica number of the inner most elastica at the bend point. The last point of this elastica \( (el->lastp->pt) \) provides the centre of rotation. The plane of rotation is given by the elastics plane equation, which describes a plane containing the elastics end-points and the equivalent points on the curvilinear axis of the surface. Mid-Threads have more elasticas than the other types of threads. To define the range of the rotation, two numbers are given, for a mid-thread, elasticas 0 to \( f_2 \) are rotated, for other threads elasticas 0 to \( f_1 \) are rotated. There is usually the following relationship between \( f_1, f_2, f_3 \) and \( f_4 \).

\[
\begin{align*}
f_2 &= 2f_1 + 1 & \text{Outer and inner-threads have } 1+2^x \text{ the elastics of a mid-thread.} \\
f_4 &= f_1 & \text{if Thread } f_3 \text{ is a Inner-Thread} \\
f_4 &= f_2 & \text{if Thread } f_3 \text{ is Otherwise}
\end{align*}
\]
This ensures all elastica points to the right of the elastica f4 will be rotated.

![Diagram](image)

**Figure 3-1 - Bending a sleeve**

If the angle is negative then the elasticas over the bend will stretch, if the angle is positive they will compress, but this will cause surface self-intersection when the angle becomes more than a few degrees.

**Figure 3-2 – Stretching and self-intersection of surface.**

Once the end-points and the normals of the elasticas have been rotated into their new positions, each elastica length is set to the distance between its end-points and deformed. In his thesis, Nikitas frequently states that an advantage of using elasticas is their length preserving properties yet here this breaks down as the Elastica lengths are explicitly changed. We can clearly see the results of this as the surface stretching and compressing in Figure 3-2. This is of course an undesired feature for cloth modeling.

**Design of improvements**

**Rendering – Conversion to OpenGL.**

The best way to tackle the problem of rendering speed is to convert Nikitas’s code to use a faster graphics library (possibly hardware accelerated), OpenGL is the obvious choice. The number of polygons that make up a clothing model is typically of the order of 20,000 triangles. Cheaply available hardware implementations of OpenGL can easily cope with rendering this number of triangles with smooth shading and texture mapping, in real-time frame rates. This means rendering should not be a barrier this model becoming real-time. Visual results could be improved by improving lighting, shading methods and implementing texture mapping. All
these are relatively easy to do in OpenGL, the only non-trivial problem comes with the generation of texture co-ordinates from the Elastica based model.

The way Nikita’s Deformations program is structured makes conversion to using OpenGL straightforward. All the high level UI and drawing routines are in the files handler.c and display.c. These then call the low-level routines found in these files;

- `xi.c`: User interface, mostly menu handling
- `xgr.c`: Opens and closes windows, draws lines, and simple polygons
- `grlib.c`: Performs view transforms, vector algebra etc.
- `shader.c`: Draws shaded polygons
- `clipper.c`: Clips polygons

OpenGL is a graphics library that will handle clipping, shading, and view transforming automatically, all we need supply are 3D descriptions the polygons that make up our surface, and various view, lighting and other parameters. This means that within the high level drawing routines in display.c, calls to the lower-level code can be removed or replaced with calls straight to OpenGL, removing the need for shading, clipping, view transform and simple line polygon rendering routines. Additional code is needed to set up OpenGL parameters.

The way the UI has been written also makes converting handler.c and display.c to use GLUT rather than xi.c not trivial, but straightforward. The UI is menu driven using calls to xi.c which draws menus and handles mouse-input etc. These calls can not be directly substituted with calls to GLUT, a reimplementation of the high level UI code is necessary. Some of this code resides in the file handler.c, some in display.c.

With the graphics routines converted to OpenGL, testing of the models capabilities for interactive deformation can begin. This requires implementation of a user interface to control deformation interactively. Even if interactive deformation is not possible with this model, interactive viewing (rotating, translating, scaling) of the surface should be possible. I did not go into the design of ways to implement this in detail.

Generating Texture coordinates

In order to implement texture mapping in OpenGL, we define a surface as a set of Polygons. For each vertex of a polygon, texture coordinates (s, t) are stored. These are interpolated between vertices to provide a mapping between a 3D surface point and the 2D (s, t) coordinates of a texture.

![Figure 3-3 - Texture coordinates](image)

Texture coordinates (s, t) range from 0 to 1 as shown in Figure 3-3. Texture coordinates can be specified outside this range, where the texture is wrapped around accordingly. Cloth is an ideal material to render as a texture mapped surface as it exhibits small-scale structure, but large-scale self-similarity. Repeating texture bitmaps for cloth such as the example in Figure 3-3, can be easily generated or are easily acquirable from texture archives.

In order to get the cloth texture mapping to look realistic, careful placing of texture coordinates is needed so that the texture does not appear to stretch or shear on the surface. At the deformations stage of the algorithm, we do not have access to the original surface in B-Spline form, we only have the Threads and Elasticas model.
Surface reconstruction generates polygons whose vertices are shared with elastica points. Therefore, what we need are texture coordinates for all the elastica points in the model. These only need be calculated once and stored. To store them I propose extending the ElasticaPt data structure to include a set of texture coordinates. Threads are not necessarily evenly spaced, nor need they be parallel. Elastics can also vary in length along a thread so finding texture coordinates from Elastics is not straightforward. I now examine possible ways to solve this problem.

**Texture Solution 1**
Fortunately, the Patterns program generates threads from isoparametric curve-lines on the input NURBS surface, so they should be roughly parallel. Using this fact, we should be able to get reasonable results by presuming each thread will also be isoparametric in texture coordinate s. When loading an elastica file [See Section - Reference here], Threads are read sequentially, and within each thread is a sequential list of the Threads Elastics. Each elastics entry contains the elastics length, which can be used to provide the change in texture coordinate t between the elastica points along a thread. In order to calculate the change in s between threads we need a way of estimating the surface distance between Threads. This is summarised by the following pseudo code.

\[
s = 0;
\]

\[
\text{For each thread } T_i \ (0 < i < \text{ThreadsNr})
\]

\[
t = 0;
\]

\[
\text{For each elastica } E_j \ (0 < j < T_i.elsnr)
\]

\[
\text{Set } E_j.texcoordstart = (s, t) \text{ and } E_j.texcoordend = (s, t + E_j.length)
\]

\[
t = t + E_j.length.
\]

\[
s = s + \text{SurfDistEstimate}(T_i, T_{i+1})
\]

The SurfDistEstimate() function aims to estimate the distance between two threads and thus provide the change in texture coordinate s between threads. In order to get good results from this technique, it is required that the start points of each thread are reasonably aligned, and that the distance between Threads does not vary much along their length. If this is the case a good solution for the SurfDistEstimate() function is the Euclidean distance between the two threads start points.

**Texture Solution 2**
Rather than leave it up to the Deformations program to calculate texture coordinates from the Elastics, an alternative is to change the Elastica file to include texture coordinates for each elastica end-point. This then leaves it up to the program that generates the elastica file to decide on texture coordinates. With the Patterns program, the elastica end-points come from points that lie on a B-Spline surface with parameters (u, v) these parameters provide reasonable texture coordinates in most cases, but the surface may appear to stretch or compress in places.

**Texture Solution 3**
The ideal solution is to be able to go back to the stage when the model of the clothing object was constructed from flat panels. Texture coordinates would then follow a regular grid on these flat panels. Alternatively, the paper by Aono et.al. – “Fitting a woven cloth model to a curved surface” describes a method that can find a 2D pattern that could be deformed into the shape of the surface under the constraints of woven fabric. Given this it would be possible, (but perhaps somewhat an overkill), to get texture coordinates from the 2D pattern of as surface. Unfortunately, both of the ideas suggested here are beyond the scope of this project.

**Bending Sleeves**
Initial visual results from sleeve bending were not pleasing, especially when the bending angle is large. Here I propose some alternative methods for incorporating sleeve bending into this model.
Distributing the Bending

The same bending function as described earlier can be used in multiple positions along a thread. We split the one rotation of a large angle into several rotations of a smaller angle. For each rotation, a different elastica of the bend thread is used to define the centre of rotation. This combination of several bends through a small angle will distribute the unwanted stretching effect over several elasticas, reducing the visual effect of the distortion to the surface.

![Figure 3-4 – Single vs. Multiple rotation points to simulate a 90° bend.](image)

To implement this, the bend function could be called several times. A more efficient solution is to change the BendThreads function so that it will progressively changes the angle and centre of rotation for each Elastica.

Randomising Hinge Patterns

The visual output from the clothing models generated by the Patterns program displays deformation patterns that are too uniform and not like those one would expect to see on real garments. In order to attempt to improve this I now propose a method for randomising the hinge placement.

The way in which the deformations program has been written, places restrictions on how hinges can be placed. These are the restrictions for hinge placement in the case of diamond patterns.

- Each region has one diamond in its width
- Each region will have \((3 + 2 \times \text{InNr})\) Threads (InNr is a given constant (usually 2), and is the number of inner-threads [See Figure ???])
- Each Thread is an isoparametric curve (in v-direction) on the surface.
- All the midpoints of the Diamonds are assumed to lie on a mid-thread, i.e. they must lie on the isoparametric curve that bisects the region.

These restrictions mean that random displacement of diamond vertices can only be permitted in the v-direction, but this is sufficient to add some degree of non-uniformity to the model. I shall experiment with sampling these displacements from a Uniform or Gaussian distribution, a parameter being the variance of the distribution. Extra checks may be needed to ensure that random displacements do not cause diamond vertices to crossover.

Animation

Currently Nikitas’s code only produces static images, but the generation of simple animation from the code should be possible. The thread-elastica model only needs to be built once. The Deformations program then makes displacements to the elastica endpoints, and then reconstructs the shape of the elasticas in-between thus producing the deformed shape of the garment. In order to produce animation sequences, further displacements can be made to the elastica endpoints and the surface reconstructed again.

The displacements can be generated by using the `BendThreads()` function, set up to model the bending of each joint. For each animation frame, the bending angle (or change in angle) for each joint can be provided and linked to the joint angles of an underlying artificial human. With this simple way to provide animation, we have to be careful about repeatedly applying the
BendThreads() function as this might lead to unwanted accumulative stretching effects. It should be a requirement that the bending of joints satisfies the following for simple bending in one direction:

- A bend of angle \(a\) followed by a bend of angle \(b\) should be equivalent to a bend of angle \(a+b\).
- Bending should be invertible, i.e. a bend of angle \(a\) followed by a bend of angle \(-a\) = Identity

This should be satisfied by the current BendThreads() function when bending is restricted to the same centre and plane of rotation. This limits us to only allow bending in one direction.

A better way to provide the displacements needed for animation is to implement Nikitas’s Layered kinematic model.

Layered Kinematic Model.

Nikitas proposed but did not implement a ‘Layered Kinematic model for Animating Complex Deformations’. The layered model he proposes is a way of generating the required displacements of elastica endpoints from the movements of an underlying virtual body. Nikitas describes a layered model for rigid structures and flexible structures, the later is more relevant to clothing modeling so I describe it here.

The virtual body can be either an articulated skeleton, or a polygonal surface representation of the body.

A number of key curves are constrained to the character. These are in the proximity of joints and will follow the characters motion.

The key curves are connected together with long strait elastics. These are length preserving and their endpoints are constrained to lie on the key curves. When the virtual body moves, the key curves will be displaced and the long elastics deform between them, providing the large scale deformation of the surface.

![Key Curve, Long Elastica, Articulated Skeleton](image)

**Figure 3-5 - Long Elasticas providing large scale deformation**

The long elastics are then linked to the underlying short elastics by constructing normals from the long elastica that intersect the short elastics endpoints, as shown in Figure 3-6. These are then fixed to the long elastica as it deforms, providing the new positions of the short elastica endpoints. The short elastics then provide the small scale buckling on the surface.
Preserving Surface area

If the long elasticas are allowed to deform completely independently then the elasticas could bend in different directions, causing the underlying surface to stretch. In order to ensure that the surface area of the underlying model will be preserved an extra constraint is added; the curve that interpolates the midpoints of the long elasticas must preserve its length during deformation. In order to implement this, Nikitas suggests the following steps with reference to Figure 3-7.

1) A circumferential thread is added that interpolates the midpoints of the long elasticas.
2) The circumferential thread is split into three elasticas of equal length, with a vertex (P) inline with the long elastica (L) that will suffer the most deformation (i.e. the long elastica on the inside of the bend)
3) The new position of P is then calculated as the midpoint of the deformed elastica L.
4) The two circumferential elasticas connected to P are deformed using the length preserving Inverse Dynamics procedure.

Figure 3-6 - Mapping between long and short elasticas

The long elasticas can also be used to provide the response of the system to external impact forces. These forces can be distributed over the higher layer to provide the large-scale deformation with relatively little cost.

Figure 3-7 - Preserving Surface area with Circumferential Elasticas
6) Mid points for all the other long elasticas are calculated from intersecting the circumferential elasticas with the deformation planes of the long elasticas.

7) A modified Inverse Dynamics procedure is used to give the shape of the long elasticas given the long elasticas endpoints and midpoint.

The model can be enhanced to provide a simple way of modeling collision detection around a joint.

**Implementation**

A complete implementation of the layered model is beyond the scope of this project. Instead, in order to test the ideas behind the model I planned to implement a simplified Layered model just for the case of a sleeve bending.

The input will be the same elastica file produced by the Patterns program as before. On top of this, I will need to build the higher layer model.

The surface will be segmented into three segments (as in Figure 3-5), the borders of each segment will be the key curves. The first and last key curves will be constrained to the start and end points of the surfaces threads. In order to determine two mid key curves we can use the end points of elasticas. Since the configuration of elasticas produced by the Patterns program is uniform, each Thread has the same number of elasticas (mid-threads have a different but proportionate number of elasticas). By selecting the endpoints of the ith elastica from each thread, we can generate a key-curve corresponding to the set of ith elasticas from each Thread. We then need to decide on two numbers (i and j) in order to generate the two mid key-curves we need.

Each underlying thread will have three associated long elasticas, one for each segment. The end-points of each long elastica will lie on the key-curves and hence will coincide with the endpoints of the underlying elasticas at the position of the key-curve.

For each Thread, a mapping between long and short elasticas needs to be constructed, like that shown in Figure 3-6. For each long-thread, all the associated short elasticas are stored, along with the mapping that gives the short elasticas endpoints by tracing a normal from the long elastica. This mapping consists of a position along the long elastica, combined with the distance from that position to the elastica endpoint, in the direction normal to the long elastica. This direction is also constrained to lie in the bend plane of the long elastica. In order to define a position along an elastica, the elastica can be parameterised on the interval [0,1], I call this parameter t from now on.

I propose adding a new data structure for a long elastica. This will need to contain the following:

- A description of the shape of the elastica (could be a pointer to the normal Elastica data type).
- An array of pointers to the underlying short elasticas of this long elastica
- An array of positions (t-values) along the long elastica corresponding to the vertices of the underlying short elasticas.
- An array of distances to the underlying elasticas.
For each Thread, three long elasticas will need to be stored, this can be easily added to the Threads data structure as an array of three long elasticas. The key-curves are just the endpoints of the long elasticas. Provided we are careful how these end-points are displaced, we do not need to explicitly model or store the key-curves.

Once we have constructed the Layered model, a simplified version of the \texttt{BendThreads()} routine can be used to deform the long threads. Alternatively, displacements could be mapped directly to the endpoints of the long threads, possibly linked directly to mouse movements giving some degree of interaction.

A simple routine is now needed to get the displacements of the lower layer from the deformed long elasticas. I will now give the outline of such an algorithm.

For each long-elastica;

1) Get the array of associated short elasticas.
2) For each of these short elasticas do the following
- 3) Get the associated stored \( t \)-value \( t \), and distance \( d \)
- 4) Find the point \( p \), and the bend-plane normal \( n \), of the long-elastica at \( t \)
- 5) Set the endpoint of the underlying elastica to the point that is distance \( d \) from \( p \) along the direction of the normal \( n \).
- 6) The start-point of the underlying elastica is the same as the endpoint of the previous elastica, or if it is the first elastica, the start point is set to the same as that of the long elastica itself.

The same Inverse Dynamics routine that calculates the shape of the short elasticas should be applicable to the long elasticas. All the short elasticas currently use 11 points to store their shape, for the longer elasticas we might want to use more.
Chapter 4 - Implementation

Compiling Nikitas's code

A significant proportion of my time was spent figuring out how Nikitas's code worked and fixing bugs. These problems are not directly relevant to what I was trying to implement, but as they took up more time than anything else, I will give brief detail of the sort of problems I was faced with.

Compiler issues

Nikitas's code had been written for a different version of the compiler than that I was using. This meant the makefiles had to be rewritten, and certain lexicographic changes made such as converting '+ = ' to '+= '.

Changes in libraries

An example is with the xgr.c and the xi.c files. Changes in the way graphics contexts are handled by the X11 libraries, meant that the following line had become invalid. Code similar to this appeared in many of the functions of xi.c and xgr.c :

```c
/*get the current function*/
currentop = gc->values.function;
```

This needed to be replaced by the following:

```c
XGCValues gc2;
/*get the current function*/
XGetGCValues(display,gc,GCFunction,&gc2);
currentop = gc2.function;
```

Lack of documentation

A lack of documentation and commenting of code was a major problem. I had to examine in detail extensive amounts of code in order to ascertain the necessary information such as the following:

?? What should be entered at the command line.
?? What are the input/output files.
?? What needs to be setup in the header file.

A typical example of a problem is this. When I was using the deformations program to render more than one surface at a time, strange polygons would appear looking like wings between the sleeves and body of the pullover model. It turned out that this was because it was assumed that the first two surfaces entered would be seamed together along their common borders. Originally, I had assumed it was a problem with surface reconstruction, perhaps a memory problem.

Problems with code

There were many problems with the surface interrogation code. Despite spending several weeks working on the code, I only ever managed to get the program to terminate correctly with the simplest input (a cylinder) and even then the output file it produced was incorrect. There are many reasons why it could have caused so much trouble, the complexity of the algorithm, the changes in compiler and the fact that the code was only written to test and demonstrate the ideas, rather than to be a robust complete implementation of the model. I had to give up working on this as it was taking up too much time and going nowhere. This meant I could not complete the task of testing the surface interrogation model on a scan of real clothing.
Conversion to OpenGL and GLUT UI

All the OpenGL code was added to the display.c file within the Deformations package. I have renamed the OpenGL version of this package DeformationsGL. The Patterns package didn't require conversion to OpenGL as rendering speed is not of concern to this part of the algorithm. However, the display.c function within this package is very similar to that of the Deformations package. It turned out easier to edit the OpenGL version of display.c for use in the Patterns program, than to fully debug the various components of the original display.c and the associated xi.c and xgr.c.

Setting up the environment for OpenGL

Within the display.c file I set-up global parameters to describe the following

GLint screen_width, screen_height;
This is the current width and height of the display window.

enum {drawsurface, quit, smooth, flat, wireframe, colour, reset, dispsurface, dispthreads, dispnormals, dispplane};
enum {red, darkred, green, darkgreen, blue, darkblue, yellow, darkyellow, customcolour};
enum {lightpos, lightdir};
enum {scale, translate, rotate, bend};
These are identifiers used to identify menu selections.

GLuint mousemode;
An integer that determines what effect mouse actions will have.

GLfloat view_r1, view_r2, view_scale, view_tx ,view_ty;
View parameters; two rotations a scaling and two translation components.

GLfloat obj_bend;
A parameter used to interface between mouse movements and the deformation model

int origx, origy;
This record the position of the mouse when a button is pressed

GLfloat view_tx2, view_ty2, view_r12, view_r22, obj_bend2;
These parameters are used to record the current state when a mouse button is pressed so that change in state can be linked to a mouse drag.

GLfloat mat_specular[] = {0.3, 0.3, 0.3, 1.0};
GLfloat mat_shininess[] = {1.0};
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
Material lighting properties. As we are modeling cloth, we choose to have little specula reflection, and a low shininess value.

int displaythreads, displaynormals, displayplane;
int displaysurface=1;
These are flags that tell the drawing routine whether to display threads, normals, elastica plane normals, or the surface reconstruction respectively.

int bobsidle;
More on this later

The View Transform

The procedure projection takes the view parameters and sets up OpenGL with the desired projection and model view transform.

void projection(void) {

}
glMatrixMode(GL_PROJECTION);
gLoadIdentity();

/* Setup 3D orthographic projection */
gOrtho(-(GLdouble)screen_width/400.0, (GLdouble)screen_width/400.0,
       -(GLdouble)screen_height/400.0, (GLdouble)screen_height/400.0,
      -10.0, 10.0);

/* Set up transform to rotate scale and translate the object */
gMatrixMode(GL_MODELVIEW);
gLoadIdentity();
gScalef(view_scale,view_scale,view_scale);
gRotatef(view_r1, 1.0, 0.0, 0.0);
gRotatef(view_r2, 0.0, 1.0, 0.0);
gTranslatef(-view_tx,-view_ty,0.0);

/* adjust viewport */
gViewport(0,0,screen_width,screen_height);
}

Setting up the UI.
All the UI is menu driven, with text input through the standard in stream. I made a few changes to Nikitas’s original menus to reflect some of the extra capabilities of OpenGL. Options to display surface Threads, Elastica Normals and Elastica Plane normals were added. The number of available colours was expanded to eight plus the ability to define your own colour. Finally, the way in which the view was transformed was changed to take into account the degree of intractability OpenGL provides. Nikitas has a menu to select a particular transform, and then requires the user to enter the details of the transformation through the keyboard. With OpenGL however, the rendering speed is fast enough to allow interactive transformation of the surface. To implement this I replaced the transformation menu with a mouse mode menu. This allows the user to select a particular action for the mouse, for example when rotate is selected, dragging the mouse over the window causes the view to rotate.

The main menu has the following submenus and entries.

<table>
<thead>
<tr>
<th>Draw Mode</th>
<th>Colour</th>
<th>Lighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smooth Shaded</td>
<td>Red</td>
<td>Position Light</td>
</tr>
<tr>
<td>Flat Shaded</td>
<td>Dark Red</td>
<td>Direction Light</td>
</tr>
<tr>
<td>Wireframe</td>
<td>Green</td>
<td></td>
</tr>
<tr>
<td>Surface</td>
<td>Dark Green</td>
<td></td>
</tr>
<tr>
<td>Threads</td>
<td>Blue</td>
<td></td>
</tr>
<tr>
<td>Normals</td>
<td>Dark Blue</td>
<td></td>
</tr>
<tr>
<td>Elastica Plane Normals</td>
<td>Yellow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dark Yellow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Custom Colour</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mouse Mode</th>
<th>Reset</th>
<th>Quit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bend</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Setting up glut and OpenGL.
The GLInitialise() routine does the following initialisations
?? Set the default state of view parameters
?? Set up the drawing window
?? Registers these call back routines with the drawing window
  glutDisplayFunc(draw); Handles redrawing of the window
  glutReshapeFunc(reshape); Handles resizing of the window

4-38
```

```glutMouseFunc(mouse); Handles mouse clicks
```glutIdleFunc(idle); Called when the system is idle
```glutMotionFunc(mouseMove); Handles mouse motion

?? Setup the menus
?? Setup lighting parameters and enable lighting
?? Enable Z-buffering
?? Set Default drawing mode to smooth shaded
?? Setup 3D view transform

Surface drawing

Surface reconstruction is carried out by the ReconstructSurfaces() function. This sets up the Tiles[][] array with the Tiles of the surface. A tile is a list of Elastica points (ElasticaPt) that together make a surface polygon. The rendering of the surface is carried out by the DisplayPolygonalSurface(sf) function. This takes each Tile of the surface in turn and constructs a GL_POLYGON using the elastica points and normals. By specifying the normals at each vertex, we can get OpenGL to calculate shading values for us.

```DisplayPolygonalSurface(sf)
Surface *sf;
{
    /* Displays the polygons of the surface */

    int i,j,l;
    GLfloat x,y,z,w, nx,ny,nz,nw;
    ElasticaPt **epts;

    /* make a count of polygons */
    PolygonsNumber = 0;

    for(j=0; j<sf->ThreadsNr; j++){
        i = 0;
        while(sf->Tiles[i][j] != NULL){
            epts = sf->Tiles[i][j]->v_list;
            glBegin(GL_POLYGON);
            for(l=0; l<sf->Tiles[i][j]->sides; l++){
                x = (*(epts+l))->pt->x;
                y = (*(epts+l))->pt->y;
                z = (*(epts+l))->pt->z;
                nx = (*(epts+l))->nm->x;
                ny = (*(epts+l))->nm->y;
                nz = (*(epts+l))->nm->z;
                glNormal3f(nx,ny,nz);
                glVertex3f(x,y,z);
            }
            glEnd();
            PolygonsNumber++;
            i++;
        }
    }
    glFlush();
}
```

The DisplayThreads() function displays the elasticas of each Thread. An elasticas colour depends on the type3 flag of the elastica, this indicates whether the elastica should bend inwards or outwards (elasticas over a hinge bend outward, whilst elasticas between hinges bend inward). In order to extract the shape of elasticas from the data structure DisplayThreads(sf) does the following steps.
Surface sf is the given surface of which we want to draw the threads. The array sf->Threads[i] provides us with the threads of the surface, with 0<\text{i}<sf->ThreadsNr. For each Thread we can obtain the threads elastics from sf->Threads[i]->els. This is a pointer to the elastics of the thread. el=*(els+j) will give us the jth Elastica, with 0<\text{j}<sf->Threads[i]->elsnr. el->pts then gives us an array of ElasticaPt which we use to draw the elastica.

DisplayThreads(sf)
Surface *sf;
{
  int i, j, l;
  ElasticaPt *pts;
  hcoord *pt1, *pt2;
  Elastica **els, *el;

  glDisable(GL_LIGHTING);
  glDisable(GL_LIGHT0);
  glDisable(GL_COLOR_MATERIAL);
  for(i=0; i<sf->ThreadsNr; i++) {
    els = sf->Threads[i]->els;
    for(j=0; j<sf->Threads[i]->elsnr; j++) {
      el = *(els+j);
      pts = el->elpts;
      glBegin(GL_LINES);
      if(el->type3 == 1) glColor3f(1,1,0);
      else if(el->type3 == 2) glColor3f(0,1,1);
      for(l=0; l<el->elptsnr-1; l++) {
        pt1 = (*(pts+l))->pt;
        pt2 = (*(pts+l+1))->pt;
        glVertex3f(pt1->x, pt1->y, pt1->z);
        glVertex3f(pt2->x, pt2->y, pt2->z);
      }
      glEnd();
    }
  }
  glFlush();
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_COLOR_MATERIAL);
}

The routines DisplayNormals() and DisplayPlaneNormals() are similar to the DisplayThreads() routine. For each elastica el we draw a unit length line from the elastics start point (el->startp->pt) in the direction of either the elastics start point normal (el->startp->nm) or the Elastics plane normal. The elastica plane normal is obtained from the Elastica plane equation, (el->a, el->b, el->c).

**Implementation of texture mapping**

I decided to implement Texture mapping solution 2 from chapter 3. This meant the format of the Elastica file needed to be changed so that it contains texture coordinates for the endpoints of Elasticas. I added a data type for texture coordinates called tcoord

<table>
<thead>
<tr>
<th>Type tcoord</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x, y</td>
<td>Texture coordinates s and t</td>
</tr>
</tbody>
</table>
The ElasticaPt data structure was then updated to include Texture coordinates, note we no longer need to store shading values so this has been removed from the ElasticaPt data structure.

<table>
<thead>
<tr>
<th>Type</th>
<th>ElasticaPt</th>
</tr>
</thead>
<tbody>
<tr>
<td>hcoord</td>
<td>pt</td>
</tr>
<tr>
<td>tcoord</td>
<td>tex</td>
</tr>
<tr>
<td>hcoord</td>
<td>nm</td>
</tr>
</tbody>
</table>

World co-ordinates of Elastica point
Texture coordinates of elastica point
Normal of the Elastica

Elastica Entry in the Elastica file
The entry for each elastica in the Elastica file format is changed to the following.

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>type3 and type4</td>
<td>Elastica type</td>
</tr>
<tr>
<td>int</td>
<td>deformed</td>
<td>Deformed flag of elastica</td>
</tr>
<tr>
<td>float</td>
<td>length</td>
<td>Length of Elastica</td>
</tr>
<tr>
<td>float</td>
<td>dist</td>
<td>Distance between elasticas endpoints.</td>
</tr>
<tr>
<td>float, float, float, float</td>
<td>a,b,c,d</td>
<td>Plane equation of Elastica</td>
</tr>
<tr>
<td>float, float, float</td>
<td>firstp-&gt;pt</td>
<td>First point of Elastica</td>
</tr>
<tr>
<td>float, float</td>
<td>firstp-&gt;tex</td>
<td>Texture coordinates of first point</td>
</tr>
<tr>
<td>float, float, float</td>
<td>firtp-&gt;nm</td>
<td>Normal of elastica</td>
</tr>
<tr>
<td>float, float, float</td>
<td>lastp-&gt;pt</td>
<td>End point of elastica</td>
</tr>
<tr>
<td>float, float</td>
<td>lastp-&gt;tex</td>
<td>Texture coordinates of last point</td>
</tr>
</tbody>
</table>

Interpolating Texture coordinates
When calculating the deformed shape of an elastica, firstly a unit length elastica is deformed under the same conditions in a local planar coordinate system. This is done by the DeformCurve() function which calculates 11 points that describe the elasticas shape and stores them in a temporary array. The function CalcElasticaPoints() then transforms this temporary array into an array of 11 ElasticaPt, by scaling the elastica to the required length, rotating it to the required orientation in 3D space, and calculating the normals at each point along the elastica.

The texture coordinates of the endpoints of the elastica are known. In order to find the texture coordinates of the intermediary elastica points we can linearly interpolate between the texture coordinates of the endpoints. This is valid as the elastica points are equally spaced along the elastica curve, so the texture coordinates should be equally spaced too.

To do this we add the following to the CalcElasticaPoints() function, I have omitted most of the irrelevant details here.

```
calcElasticaPoints(el)
Elastica  *el;
{
  int            i;
  float          theta, dtx, dty, tx, ty;
  hcoord         *nm,*pt;
  ElasticaPt     **elpts,*ep;

  /* precalculate values for linear texture coord interpolation */
  tx=el->firstp->tex->x;
  ty=el->firstp->tex->y;
  dtx=(el->lastp->tex->x - tx)/10;
  dty=(el->lastp->tex->y - ty)/10;

  elpts = el->elpts;
  for(i=0; i<el->elptsnr; i++){
    *(elpts+i) = ep = ElasticaPtAlloc();
    /* calculate elastica point texture coord */
    /* other code here */
  }
  /* other code here */
}```
 ep->tex = TCoordAlloc();
 ep->tex->x = x + i * dtx;
 ep->tex->y = y + i * dty;

 /* calculate elastica point */
 /* calculate elastica normal */
 /* adjust for horizontal elastica && negative Ycoords */
 /* adjust for vertical elastica && negative Xcoords */
}
}

Loading Textures
As part of a previous project, I implemented a texture browser in Java that converts picture files into a particular format that can be directly loaded into an array and used as a texture by OpenGL. This did the following

?? Read the input image, Java offers a uniform interface to many image formats
?? Rescale the image (using bilinear filtering) so that is length and width are a power of 2 (requirement for OpenGL Textures)
?? Writes out an Image file of the following binary format

<table>
<thead>
<tr>
<th>File offset (bytes)</th>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GL_SHORT</td>
<td>width</td>
<td>The width of texture map</td>
</tr>
<tr>
<td>2</td>
<td>GL_SHORT</td>
<td>height</td>
<td>The height of texture map</td>
</tr>
<tr>
<td>4 + 4 * (j * width + i)</td>
<td>GL_UNSIGNED_BYTE</td>
<td>R</td>
<td>Red value of pixel[i][j]</td>
</tr>
<tr>
<td>5 + 4 * (j * width + i)</td>
<td>GL_UNSIGNED_BYTE</td>
<td>G</td>
<td>Green value of pixel[i][j]</td>
</tr>
<tr>
<td>6 + 4 * (j * width + i)</td>
<td>GL_UNSIGNED_BYTE</td>
<td>B</td>
<td>Blue value of pixel[i][j]</td>
</tr>
<tr>
<td>7 + 4 * (j * width + i)</td>
<td>GL_UNSIGNED_BYTE</td>
<td>A</td>
<td>Alpha value of pixel[i][j]</td>
</tr>
</tbody>
</table>

I used this program to convert a set of 40 cloth texture files and stored the resulting files as cloth1.rgb to cloth39.rgb in the cloth directory.

The procedure load_texture() loads a texture file into the global array texture, and sets the global parameters texwidth and texHeight to the texture size. The following command in the G initialise() routine then loads the texture into OpenGL.

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texwidth, texheight, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, texture);

Texture mapping parameters are set to perform linear filtering and repeat wrapping of texture coordinates.

Rendering with Textures
The DisplayPolygonalSurface() needs the simple addition of code to supply the texture coordinates along with the vertices of polygons. This is done with the additional lines of code within the inner loop of the DisplayPolygonalSurface() procedure.

  tx=2.0*(*(epts+l))->tex->x;
  ty=2.0*(*(epts+l))->tex->y;
  glTexCoord2f(tx,ty);

Generating Texture coordinates in the Patterns Program
The texture coordinates that are supplied in the new Elastica file format are generated by the Patterns program. This simply uses the (u, v) surface parameters of the underlying B-Spline surface as the texture coordinates. Within the Patterns package, the file inp.c contains the WriteData() function that writes an elastica file. Fortunately, the data structure used for an elastica in the Patterns program gives us easy access to the (u, v) parameters of the elastica end points. For elastica el these are;

  el->pstart->uprmvl, el->pstart->vprmvl
  el->pstop->uprmvl, el->pstop->vprmvl

We simply need to add a couple of extra lines to the WriteData() function to write out these values in the Elastica file.
Implementing Improvements to the BendThreads function

This simply involved experimenting with various configurations of the BendThreads() function within the ControlDeformation() procedure. I found that the proportion of time spend in this procedure did not make a more efficient solution necessary. The results from these experiments are discussed in the next chapter.

Implementation of Randomisation

I decided not to implement randomisation of axisymmetric buckling patterns as this mode is very rarely found on flexible garment structures, and is not used for any of the clothing examples given.

In the Patterns program, surfaces are split up into regions. For those regions deemed to buckle with diamond patterns, each region will have one diamond in its width and a number in its height determined by the buckling wavelength of the region. The function RegionDiamonds() calculates the number of diamonds for a region (or uses predetermined values in the case of the example input files) and then constructs the diamond hinge patterns on the surface.

A diamond is stored as a set of four vertices called up, down, left and right. Left and right vertices should lie on the borders of the region, and the up and down vertices should lie on the bisecting curve of the region.

Type dvertex

<table>
<thead>
<tr>
<th>int</th>
<th>inum, jnum</th>
<th>Corresponds to the discretized surface coordinates of the vertex.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SurfacePoint</td>
<td>*spt</td>
<td>Not used at this stage</td>
</tr>
</tbody>
</table>

Type Diamond

<table>
<thead>
<tr>
<th>dvertex</th>
<th>*left, *right, *up, *down</th>
</tr>
</thead>
</table>

Diamond         *Diamonds[10][20]

The global array Diamonds[][] stores all the diamonds for each region. The first index is the region number.

The function RegionDiamonds() finds the diamond vertices, constructs Diamonds and stores them in the Diamonds array appropriately. Once this is done I have added an extra bit of code to the RegionDiamonds() function that then randomises the newly placed diamonds by displacing diamond vertices in the vertical direction. This code is listed below, the function BobRand() returns a random number uniformly distributed on the interval [-1,1]. This function can be replaced by the function BobRandGauss() which returns a random sample from a Gaussian distribution. The listing for this is also given below.

```c
if (randomiseDiamonds) {
    previup = Diamonds[RegionNumber][0]->down->inum;
    for (l=0; l<DiamondsNr; l++) {
        diamond = Diamonds[RegionNumber][l];
        /* make random displacements in the vertical (i) direction */
        diamond->left->inum+=DRAND*BobRandGauss()*vleftwavelength;
        diamond->right->inum+=DRAND*BobRandGauss()*vrightwavelength;
        /* we don't want to displace the very top vertex */
        if (l<DiamondsNr-1) diamond->up->inum+=DRAND*BobRandGauss()*vleftwavelength;
        /* ensure down vertex is the same as the previous up vertex */
        diamond->down->inum=previup;
        previup=diamond->up->inum;
    }
}
```

4-43
float BobRandGauss() {
    /* returns approx Gaussian distributed random number */
    /* uses fact that sum approximately normally distributed */
    int i;
    float r=0.0;
    for (i=0; i<4; i++) {
        r=r+BobRand();
    }
    return r*0.866;  /* = 2 * Sqrt(4 * 3)/ (2 * 4) */
}

Implementation of Layered model for sleeve

The first step is to construct the layered model on top of the underlying elastica model.
For each Thread, the function ConstructLongThread() splits the thread into three long elasticas
according to two floating point numbers, \( f_1 \) and \( f_2 \). These are the desired proportions of
elasticas to lie between the first and second key curves and the first and third key curves
respectively.

This is done by the following code segment;

\[
nr=t->elsnr-1;
\]

\[
  l1=0;
  l2=rint(f1*nr);
  l3=rint(f2*nr);
  l4=nr;
\]

Elasticas \( l_1 \) to \( l_2-1 \) correspond to the first long elastica
Elasticas \( l_2 \) to \( l_3-1 \) correspond to the second long elastica
Elasticas \( l_3 \) to \( l_4-1 \) correspond to the third long elastica

For example if a thread has 28 elasticas and \( f_1 = 0.3 \) and \( f_2 = 0.7 \), then the three long elasticas
will constructed as straight undeformed elasticas as follows.

- Between the start point of elastica 0 and the end point of elastica 7
- Between the start point of elastica 8 and the end point of elastica 18
- Between the start point of elastica 19 and the end point of elastica 27

The long elasticas are constructed by the ConstructLongElasticaValues() procedure. The
deformation of long elasticas is governed by the same procedures as the short elasticas. They
use the same data structures but with the addition of a new type \( \text{ElasticaL} \) that wraps the
Elastica type and holds extra information about the mapping between long and short elasticas.

Type ElasticaL

<table>
<thead>
<tr>
<th>Elastica</th>
<th>*el</th>
<th>The long Elastica</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>elMapT[25]</td>
<td>t-values defining mapping points</td>
</tr>
<tr>
<td>float</td>
<td>elDist[25]</td>
<td>mapping distance</td>
</tr>
<tr>
<td>Elastica</td>
<td>*elMap[25]</td>
<td>Pointers to underlying Elasticas</td>
</tr>
<tr>
<td>int</td>
<td>elnr</td>
<td>Number of underlying Elasticas</td>
</tr>
</tbody>
</table>

Type Thread

The type Thread is updated to include the addition of the following entries.

| ElasticaL | *longels[3] | long elasticas for layered model |
| Solution  | *longsls[3] | solutions of the long elasticas |

The Elastica contained with in a ElasticaL is initialised with the following values.

\[
type3=type4=1\]

The elastica is set to buckle with negative coordinates
Initially undeformed.

firstp is assigned a copy of the point (ElasticaPt) of starting elastica. lastp is assigned a copy of the point (ElasticaPt) of ending elastica.

dist=length=Length(firstp, lastp)

The plane equation of the long elastica \((a, b, c, d)\) is set to be the same as that of the first short elastica.

The rest of the values will be calculated by the deformation process, For each long elastica a Solution is needed so the Thread type has an additional array of three Solution called longsls[3]. For each long Elastica, this is initially given the values:

\[
\begin{align*}
\text{pnr} & = 11; & \text{11 points will be used to model the elastics shape. A number larger than 11 would be preferable for long elastics, but Nikitas's implementation restricts all elastics to 11 points.} \\
\text{t1} & = 2 & \text{Fixed-fixed boundary conditions} \\
\text{t2} & = 2 & \text{Vertical orientation} \\
\text{P} & = 45 & \text{Loading force} \\
\text{R} & = 0 & \text{Resistance} \\
\text{W} & = 0 & \text{Weight} \\
\text{G} & = 0 & \text{Guess for initial curvature}
\end{align*}
\]

Nikitas's procedure ProcessElastica(el, sl); copies the values listed above to the equivalent values in the elastica data structure. The shape of the elastica is then calculated by a call to DeformLongThreadElasticas(t); which is the same as Nikitas's DeformThreadElasticas(t); but adjusted for the slightly different data structure.

The method for mapping long elastics to short elastics as described in Chapter 3 was implemented at the end of procedure ConstructLongThread().

It was at this point that I discovered that Nikitas hadn’t implemented the Inverse Dynamics procedure for calculating the shape of an elastica given the positions of its end-points and the angles of the elastica at those end-points. This is needed by the DeformThreadElasticas(t); function and is a critical part of the model. What is in fact done is the guess value is used as the actual solution value for the angle of the elastica at the start point. The differential equation of the elastica is then solved once by the SolveDifEqn() function, with the length of the elastica set to the distance between the end points. The guess value is then provided explicitly for each elastica on each surface, in the files control1 - control10.c. This means each elastica has to be told its solution so the model has to be set up for a single static image only. This is not sufficient for an implementation of the layered model as this needs to be dynamic to be of any use.

It was at this point I had to give up attempting to implement the layered model as at the beginning of the project I had considered the mechanics of elastics beyond the scope of the project, assuming that all I needed had already been implemented. By the time I had discovered this problem, there was not enough time to complete an implementation of the Inverse Dynamics algorithm myself.
4-46

Figure 4-1 - Long threads of the outer layer

**Generation of animations**

The `ControlDeformation()` procedure was originally only called once from the main procedure, and it is this procedure that calls the `BendThreads()` function to deform the surface. The program then allows you to view the surface, but further deformation and hence the generation of animations was not possible.

I added the ability for the user to use the mouse to control further deformations. When ‘Bend’ is selected from the mouse mode menu, then mouse actions over the view window are interpreted as follows.

**Mouse Click**

When a mouse button is pressed the position of the mouse is recorded in the global variables `origx` and `origy`.

**Mouse Movement**

Within the `mouseMove` function the section of code is called:

```c
if (bobsidle) {
    bobsidle=0;
    obj_bend=(y-origy)/200.0;
    ControlDeformation(obj_bend);
    ReconstructSurfaces();
    glutPostRedisplay();
}
```

The `bobsidle` flag is set to true only when the system is idle (this is implemented by registering an idle function with glut that sets this variable to true). This prevents action being taken over the mouse events that are queued in the period when the model is redrawing.

The change in `y` coordinate of the mouse caused by the drag motion is used to define the angle of bending `obj_bend`. This angle is then passed to the `ControlDeformations()` procedure where the surface is deformed further. The polygonal representation of the surface is then reconstructed by Nikitas’s `ReconstructSurfaces()` procedure, finally glut is instructed to redisplay the surface.
Chapter 5 - Results

Regions
The patterns program allows you to specify the number of regions that the surface is divided into. Each region contains one diamond in its width so for a sleeve, the number of regions is equal to the number of diamonds in the circumferential direction. This number could be calculated from the thickness and diameter of the sleeve according to equation 3-2, but by allowing it to be specified we gain more control over the surface. In Figure 6-1 we show the results obtained from 3, 4 and 5 regions, with the default buckling wavelength that gives us 7 diamonds along the length of a thread. The number of regions on the front and back panels of the jumper and the trousers can also be specified. With a sleeve, the elasticas solutions are conditioned uniformly over the surface so changing the number of regions does not require rewriting the elastica conditioning code, but with the trousers and shirt front and back panels this is not possible. For this reason, I have stuck to experimenting with sleeves only, particularly the LeftSleeve surface.

![Figure 5-1](image)

(a) 3 regions (5520 polygons)
(b) 4 Regions (7410 polygons)
(c) 5 Regions (9300 Polygons)
(d) 3 Regions : Ucurves, Vcurves and Diamonds

Figure 5-1 (d) shows the longitudinal and circumferential isoparametric curves of the parameter space (u,v) of the surface. On top of this, the diamonds that have been traced on
the surface are shown in red. Note how the diamond vertices lie on the u, v curve intersections. The deformed and rendered version of this is shown in (a), similarly (b) and (c) show 4 and 5 regions respectively. The number of regions should reflect the type of material we are trying to simulate. Three regions give the most pleasing results for the sleeve of a jumper. For the front and back panels of the shirt Nikitas also uses 3 regions.

Figure 5-2 - Three regions on the front panel of a shirt

The number of Threads and hence the number of Polygons increases linearly with the number of regions. As surface reconstruction and elastica deformation is of linear time complexity this means that the complexity of the simulation is linear with the number of regions.

Crease Sharpness

We can alter the ratio of the length of elasticas that lie over a fold to those that lie between. This allows an element of control over the sharpness of creases. In the following, I present the visual results from experimenting with this ratio.
For all these simulations, the number of diamonds is kept constant and only the ratio of lengths of elastics is allowed to change. Changing this parameter only has a visual effect and does not affect the complexity of the simulation. In Figure 5-3 (a) the ratio is small leading to sharp creases as might be found on leather materials, in (c) smooth creases are shown as might be found on silk or leather garments.

**Figure 5-3 - Crease Sharpness**

Buckling Wavelength/Thickness

Changing the thickness of the surface changes the buckling wavelength and hence the number of diamonds that appear along the length of the sleeve. In the following, I present the visual results from thickness’ ranging from 0.001 to 0.0004. All other parameters have been kept to their default values.
(a) Thickness = 0.001
(b) Thickness = 0.00075
(c) Thickness = 0.0005
(d) Thickness = 0.0004

(e) Threads : Thickness = 0.001

Figure 5-4 - Thickness
Progressive bending through negative angle

In figure 5-5 I show a sequence of frames showing a sleeve progressively bending. Each frame is generated by deforming the previous frame with a bend of 0.3 radians. In sequence (a) Nikitas’s single BendThreads() function is used with the bend point specified by the 11th Elastica of the 1st Thread. In sequence (b) I present the results of experimenting with using multiple bend points; here, for each frame the BendThreads function is called three times, with bend points on the 9th, 11th and 13th Elastica of the 1st Thread. Each single bend is through an angle of 0.1 radians so that the resultant total bend angle is 0.3 radians. In both animation sequences the LeftSleeve surface is used, with 3 regions and thickness 0.00075.
We can see that the bending causes unwanted stretching of the surface, the multiple bending in sequence (b) reduces this, but by the 5th frame the stretching begins to become noticeable.

**Figure 5-5 - Bending at a joint - comparison of old and new approach**

We can see that the bending causes unwanted stretching of the surface, the multiple bending in sequence (b) reduces this, but by the 5th frame the stretching begins to become noticeable.
Progressive bending through a Positive angle
This experiment is the same as the previous but a negative bending angle used, causing the surface to compress rather than stretch.
In sequence (a) the surface self-intersects after the first frame, due to the elasticas on the inside of the bend being forced to compress further that their length and in effect folding over. With the multiple bend points in sequence (b) the same compression is distributed over three elasticas so the joint can bend three times further before causing self-intersection. In sequence (a) we also get a noticeably sharp crease at the bend point. This unwanted effect is reduced in sequence (b) by the fact that there are in effect three smaller creases at neighbouring Elasticas.

Timings
So far we have discussed the time complexity of the model, to summarise, this is linear in the number of diamonds on the surface. In the following simulations, I use a sleeve and the full jumper model, with typical parameters, that is, a thickness of 0.00075 and 3 regions per surface (the front and back panels of a jumper are treated as separate surfaces). Because of the linear time complexity it is easy to extrapolate timings for different parameters.

Intuitively, it is clear from the simulations that the model is fast and that it is the rendering that is the major bottleneck. For the case of a sleeve with 5520 polygons, on my system it is possible to achieve 11 frames a second in the simplest graphics mode (wireframe with no texture mapping or lighting) however enabling lighting, smooth shading and texture mapping reduces the frame rate to 1 frame/second. This is mainly because I am using a software implementation of OpenGL, I expect hardware accelerated OpenGL to be capable of fully rendering the 5520 polygons in real-time frame rates.
What is important is the time taken by the deformation and surface reconstruction procedures. To investigate this I used the profiling capability of gcc. The following table shows the important parts of the profile of the Deformations program when bending LeftSleeve (5520 polygons) progressively 9 times.

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>23.29</td>
<td>0.56</td>
<td>0.17</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.73</td>
<td>0.00</td>
</tr>
</tbody>
</table>

We are interested in the far right hand column, this is the average time taken by the procedure and its sub-procedures for one call. The function that performs the deformation is ControlDeformation(), which calls BendThreads() to position the endpoints of the elasticas then DeformThreads() to find the solutions to the Elasticas given their new endpoints. DeformThreads() calls DeformThreadElasticas() for each Thread which in turn calls DeformCurve() for each elastica to find the solution of each Elastica. We can see that 46% of the execution time for the deformation process is spent in DeformCurve() and its sub procedures. The BendThreads() procedure itself accounts for 4% of the total the deformation process which takes 14422us (0.014... seconds) in total so approx. 70 frames/second should be possible excluding surface reconstruction and rendering. The rest of the time taken by the ControlDeformation() procedure is mostly accounted for by the transformations between local and world coordinates for each Elastica.

The surface reconstruction procedure ReconstructSurfaces() reconstructs the polygonal representation of the surface. The data structures currently require this to be called after each deformation but I believe this is not necessary. Once we have constructed a mapping between elastica vertices and polygon vertices, this mapping should not change, only the vertex positions should change as the elasticas deform. Unfortunately, the current implementation does not take advantage of this, and fixing this would have required changing data structures that would have meant rewriting large sections of code.

The DisplayPolygonalSurfaces() function takes the reconstructed surface representation and effectively sends the description of the 5520 polygons to OpenGL. Although this has not been implemented in the most efficient way possible, this only takes 0.017 seconds per frame, excluding rendering time. The time for rendering itself depends on the OpenGL drivers, and will mostly be performed in hardware, thus not taking too much processing time away from the deformations procedure.

For this implementation it currently takes a total of 17000+444444.44+14422.44 =75866.88us to deform, reconstruct and transfer the surface to OpenGL. If rendering were to take place in parallel then theoretically 13 frames/sec should be possible. By eliminating the need for the reconstruction step, theoretically frame rates of around 30 f/s should be possible.

For multiple bending the BendThreads() function takes 9% of deformation time. This is still insignificant compared to the total time taken to produce a frame.

For a complete jumper (31500 polygons) for each frame we get the following timings in the case of bending each sleeve with multiple bends, and reconstructing every Elastica.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlDeformation()</td>
<td>112079.04</td>
</tr>
<tr>
<td>ReconstructSurfaces()</td>
<td>174444.44</td>
</tr>
<tr>
<td>DisplayPolygonalSurfaces()</td>
<td>67000.00</td>
</tr>
<tr>
<td>Total</td>
<td>353523.48us</td>
</tr>
</tbody>
</table>
This equates to 3 frames a second, or 6 f/s if we can eliminate `ReconstructSurfaces()`.

It should be noted that the timings quoted here do not use the Inverse Dynamics algorithm that is necessary for a complete simulation of the model as described by Nikitas. This is likely to increase the time taken by `DeformCurve()` by a factor equal to the number of iterations required by the Inverse Dynamics algorithm to converge to its solution. However, if (as Nikitas suggests) a database of elastica solutions is implemented, we should be able to actually reduce the time taken by the `DeformCurve()` procedure significantly.

**Visual Results**

It is difficult to compare the visual results of this model with those of other techniques due to the difficulty of finding example animations from other cloth modeling techniques. However, I will now attempt to give an intuitive evaluation of the models realism from comparison with how real clothes dynamically deform.

The folds on the surface as generated by the patterns program appear too uniform, in order to attempt to address this, in section 4-?? I introduced a degree of randomization into the model. Figure 5-7 shows the model with (a) no randomization, (b) Uniform randomization with multiplication factor 0.2 (c) factor 0.4 (d) factor 0.6 (e) Gaussian randomization with factor 0.6. All randomization was performed on the `LeftSleeve` with thickness 0.00075 and 3 regions.
Figure 5-7 - Randomisation

Visually, these results definitely show a slight improvement by removing some of the uniformity of the model.
Chapter 6 - Conclusions, Evaluation and Further Work

Achievements

The aim of the report is to answer the question, ‘is Nikitas’s model capable of real-time or interactive-time animation of clothing objects?’ A full evaluation of this would have required an implementation of his layered model and Inverse Dynamics algorithm, which unfortunately was not possible. Provided here are the timings of key parts of the code and analysis of the complexity of the algorithm which enables us to answer this question.

The size and complexity of Nikitas’s model makes it difficult to understand quickly, one has to read over 100 pages of his thesis to get a full understanding of how it works. This report aims to provide a reasonable condensed explanation of Nikitas’s model. Some of the detail has been omitted, but extra explanation is given in some areas, making it more accessible to those without an engineering background.

I am grateful to Nikitas for providing his source code (via Mel Slater) for use in this project. A complete reimplementation of the model would have been difficult and very time consuming. The source code obtained from Nikitas is massive, the deformations program for example, is spread over 33 source files and totals over 15000 lines of code. Nikitas’s code came with no explanation and was virtually uncommented. Should anyone wish to develop this code further this report provides information on how Nikitas’s code is structured, how to use it, and furthermore an OpenGL version of the code with additional features is given. The beginnings of an implementation of the Layered model are also found in the directory DeformLayer. Some of the areas where the efficiency of the code can be improved have been identified and in some cases implemented.

Improvements to the model and code have been made with the addition of the following features;

- Faster rendering through use of OpenGL graphics library
- Semi-randomisation of hinge placement
- Bending of joints using multiple bend points
- Interactive deformation implemented with up to 10 f/s
- Texture mapping
- Additional viewing options added including interactive viewing
- Additional UI controls

Obstacles

Working with such a large piece of undocumented code has been the major challenge of this project. I would estimate at least half of my time was spent examining and experimenting with his code to find out how it works, and a further quarter of my time was spent fixing bugs and getting Nikitas’s code to compile on my machine. Not being from an engineering background, from the outset I decided the detailed treatment of the theory behind elastica deformation to be beyond the scope of this project choosing to rely on the code that Nikitas had already written. This unfortunately caused problems when I was trying to implement a simplified version of the layered model for the bending of a sleeve as I discovered a key component of the model, the Inverse Dynamics procedure had not been implemented. In figure 5.11 of his thesis Nikitas shows the progressive collapse of elasticas which he claims to be generated by the Inverse Dynamics procedure, I have the source code for this image in the Elasticas directory which I used to reproduce this image for Figure 2-9. However, when I came to implement the deformation of long elasticas I found difficulties in getting the elastica deformation code to do what I was expecting. This lead me to examine the Elastica deformation code in detail and I discovered that figure 5.11 did not use the inverse dynamics procedure. The Inverse dynamics procedure is supposed to search for the boundary conditions for an elastica such that its deformed shape meets the required conditions at its end-points. Instead what has been implemented is that the boundary conditions for each elastica have been explicitly given. In figure 5.11 these have been adjusted to give the look of a progressive deformation. In the main deformations program, I found that the elasticas of the are also explicitly given their boundary
conditions in the control4-control10.c files. This meant the deformations program had been set up for static images for the pre-defined set of input surfaces. When I tried to use the Deformations package with files generated by the Patterns program with different surface parameters (the numbers of regions or thickness) the control procedures needed rewriting to reflect the different number and configuration of Threads and Elasticas on the surface. In the case of a sleeve, all the elasticas solutions are uniform so this did not matter. However for the body of the shirt and the trousers, the Elastica solutions are not uniform over the surface leading to the variable crease size features we see on those surfaces. When producing dynamic simulations, the Inverse Dynamics procedure is necessary for the model to be complete as the animator can not be expected to provide the boundary conditions for every elastica in each frame. Although I attempted to implement the Inverse Dynamics procedure myself, I ran into many problems like how to ‘adjust the boundary conditions accordingly’ and what to do if a solution does not exist. Unfortunately, time did not permit the completion of this.

**Performance**

Without a full implementation of Nikitas’s layered kinematics model for animating complex deformations it has not been possible to evaluate this model to its full potential but from the timings of the current code it is possible to estimate the performance of such a model. A great advantage of this model over pure physical methods is its speed. There are no large systems of differential equations to solve, we only need to solve independent equations for each elastica. This means there is potential for a massively parallel implementation of this model, and timings indicate that currently real-time frame rates are possible without parallelisation. We have seen that 6 frames a second is possible for the complete simulation of a jumper.

For a complete layered model, the time taken to deform the short-elasticas will be the same but first we have to deform a simpler outer layer. In the simplest case, the outer layer is deformed using similar inverse dynamics procedures as the inner layer, the displacements for the long-elasticas end-points coming from the movements of the key-curves tied to a synthetic actor. The process of mapping long to short elasticas involves finding a point on an elastica followed by the simple vector addition of a multiple of the normal at that point. This requires little computation as the point on the elastica and its normal can be easily found from interpolating the elastica points generated in the deformation process. This should require less computation than the simple bending of threads so all the extra cost of the layered approach will be due to the deformation of the outer layer. As this will be much simpler than deforming the underlying short-threads, I would expect in total the layered model to be at least half as fast as the simpler case. Theoretically this means at least 3 frames a second should be possible from the timings of the simpler model. These timings however are based on an implementation of the model that does not use the Inverse Dynamics method. This would multiply the cost of elastica deformation by a factor roughly equal the average number of iterations required by the inverse dynamics procedure to converge. However, Nikitas suggests a database of elastica solutions can be built, removing the need for an Inverse Dynamics procedure altogether. If this were implemented then we can expect much faster timings than those described in the previous section even for the layered model. Thus a reasonable estimate for the possible frame rate achievable by the layered model is at least 5 frames a second, for deforming a jumper of 31500 polygons. This gives the model a big advantage over physically based models, which are still struggling to attain these kinds of frame rates for such complex models.

**Visual Performance**

In the case of bending a sleeve, the deformation of the surface only occurs with the elasticas on the bend point, the rest of the surface remains undeformed and is merely transformed by a rotation or a translation. This leads to the surface appearing rigid in areas away from the deformation point. The layered approach should not suffer from this problem as much, depending on how the upper layer is deformed.

Nikitas’s model is based on the collapse of rigid cylindrical and square tubing under axial loads. This leads to the patterns of folds that appear unrealistically uniform. Folds only appear in a direction perpendicular to the curvilinear axis of the surface, unlike real garments where folds can appear in any direction. Deformations due to twisting of the surface, which are very common, cannot be simulated with this model.
For realism, physically based models offer much better results, and the mass-spring model is approaching real-time frame rates. For a mass-spring based model to be able to accurately capture fine wrinkling of a surface the mesh divisions need to be small compared to the size of the wrinkles. This leads to large mesh sizes that are currently too large to be real-time. Nikitas explicitly models these creases geometrically, so the divisions in the grid elasticas need only be of the same size as the desired wrinkles.

Collision detection is not implemented by this model, if we allow small self-intersections, then the layered model may well be able to get away without collision detection, if collision detection were to be implemented fully, the model will begin to lose its speed advantage.

**Taking further**
An obvious place to start further development would be to implement the Inverse Dynamics procedure and Layered kinematics model. (I would recommend a complete rewrite of the code with efficiency in mind). Furthermore an investigation into the possibility of implementing a database of elasticas would be beneficial. With this done a complete evaluation of the model should be possible. The limitations of this model make it only really suitable for modelling simple trousers and shirts. However, the fact that the model is reasonably fast and massively parallelisable, makes it particularly suitable for clothing synthetic actors in interactive-time, where realism is not of primary concern.

**Conclusion**
In conclusion, I believe Nikitas’s model is capable of real time and interactive-time animation. However the model is limited to only simple tube-like surfaces and suffers from poor visual results. I believe one possible application for Nikitas’s model would be the dressing of simple virtual actors where external collision detection is not important. The speed of the model may mean it is possible to dress many virtual humans at the same time.
Chapter 7 – Code listing

This is the display.c file from the DeformTex package. All the functions are my own work, some were based on the original functions of Tsopelas.

#include "headers.h"
#include "xgr.h"
#include "grlib.h"
#include "GL/glut.h"

extern InitialiseZ_buffer();
extern ShadePolygon();
extern hmg_division();
extern clippolygon();
extern XXPolygon();
extern float InnerProduct();
extern PolygonPoint *PolygonPointAlloc();

void initGL(void);
void draw(void);
void projection(void);
void initMenu(void);
void menu(int id);
void menu_draw(int id);
void menu_colour(int id);
void menu_mode(int id);
void idle(void);
void mouseMove(int x, int y);
void mouse(int b, int s, int x, int y);
void load_texture(char* file, int alpha);
void reshape(int w, int h);
void GLInitialise(int argc,char** argv);

int idx,Shade_Surface,PolygonsNumber;
hcoord ToLightSource,LightSource;

GLint screen_width, screen_height;
int origx, origy;
/* Enumerate names for menus */
enum {drawsurface, quit, smooth, flat, wireframe, colour, reset, dispsurface, dispthreads, deformcurves};
enum {red, darkred, green, darkgreen, blue, darkblue, yellow, darkyellow, customcolour};
enum {lightpos, lightdir};
enum {scale, translate, rotate, bend};
GLuint mousemode;
GLuint texName[1];
int texwidth, texheight;
GLuint* texture;

GLfloat view_r1, view_r2, view_scale, view_tx, view_ty, obj_bend;
GLfloat view_tx2, view_ty2, view_r12, view_r22, obj_bend2;
GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat mat_shininess[] = {50.0};
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
int bobsidle, displaythreads;
int dispsurface=1;
HandleEvents(int argc,char** argv)
{
    int    i,j;
    int    selection,xpos,ypos,WorkFinished;
    long   mask;
    Window drawWindow;

    printf("HandleEvents\n");
    GLInitialise(0,NULL);
    glutMainLoop();
}

Display3DSystem(win)
Window  win;
{
    glBegin(GL_LINES);
    glColor3f(1.0,1.0,1.0);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(2.0,0.0,0.0);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(0.0,2.0,0.0);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(0.0,0.0,2.0);
    glEnd();
}

DisplayPolygonalSurfaces(void)
{
    int            i;
    for(i=0; i<SurfacesNr; i++){
        if (displaysurface) DisplayPolygonalSurface(Surfaces[i]);
        if (displaythreads) DisplayThreads(Surfaces[i]);
    }
}

DisplayThreads(sf)
Surface *sf;
{
    int        i,j,l;
    ElasticaPt **pts;
    hcoord     *pt1,*pt2;
    Elastica   **els,*el;

    glDisable(GL_LIGHTING);
    glDisable(GL_LIGHT0);
    glDisable(GL_COLOR_MATERIAL);
    for(i=0; i<sf->ThreadsNr; i++){
        els = sf->Threads[i]->els;
        for(j=0; j<sf->Threads[i]->elsnr; j++){
            el = *(els+j);
            pts = el->elpts;
            glBegin(GL_LINES);
if(el->type3 == 1) glColor3f(1,1,0);
else if(el->type3 == 2) glColor3f(0,1,1);
for(l=0; l<el->elptsnr-1; l++){
    pt1 = (*(pts+l))->pt;
    pt2 = (*(pts+l+1))->pt;
    glVertex3f(pt1->x, pt1->y, pt1->z);
    glVertex3f(pt2->x, pt2->y, pt2->z);
}
glEnd();
}
glFlush();
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_COLOR_MATERIAL);
}

/****************************************************************************
/
DisplayPolygonalSurface(sf)
Surface *sf;
{ /* Displays the polygons of the surface */
    int            i,j,l;
    GLfloat        x,y,z,w, nx,ny,nz,nw, tx,ty;
    ElasticaPt     **epts;

    /* perform some initial operations before displaying the first surface */
    if(sf->index == Surfaces[0]->index){
    }

    /* draw the surface either as wireframe or shaded */
    glEnable(GL_TEXTURE_2D);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    PolygonsNumber = 0;
    for(j=0; j<sf->ThreadsNr; j++){
        i = 0;
        while(sf->Tiles[i][j] != NULL){
            /* construct a polygon */
            epts = sf->Tiles[i][j]->v_list;
            glBegin(GL_POLYGON);
            for(l=0; l<sf->Tiles[i][j]->sides; l++){
                x = (*(epts+l))->pt->x;
                y = (*(epts+l))->pt->y;
                z = (*(epts+l))->pt->z;
                nx = (*(epts+l))->nm->x;
                ny = (*(epts+l))->nm->y;
                nz = (*(epts+l))->nm->z;
                tx = TexMul*((*(epts+l))->tex->x);
                ty = TexMul*((*(epts+l))->tex->y);
                glTexCoord2f(tx,ty);
                glNormal3f(nx,ny,nz);
                glVertex3f(x,y,z);
            }
            PolygonsNumber++;
            glEnd();
            i++;
        }
    }
}

}
glFlush();
printf("Polygons:%i\n",PolygonsNumber);
}

/***************************************************************
*/
TransformThread(idx,t,a,b,c,angle)
int     idx;
Thread  *t;
float   a,b,c,angle;
{
    int     i,j;
    ElasticaPt  **pts;
hcoord   *pt,*nm;
    Elastica **els,*el;

    /*printf("TransformThread\n"); */
    els = t->els;
    for(i=0; i<t->elsnr; i++){
        el = *(els+i);
        pts = el->elpts;
        for(j=0; j<el->elptsnr; j++){
            pt = (*(pts+j))->pt;
            nm = (*(pts+j))->nm;
            switch(idx){
                case 1:
                    Rotate_X(pt,angle);
                    Rotate_X(nm,angle);
                    break;
                case 2:
                    Rotate_Y(pt,angle);
                    Rotate_Y(nm,angle);
                    break;
                case 3:
                    Rotate_Z(pt,angle);
                    Rotate_Z(nm,angle);
                    break;
                case 4:
                    printf("not implemented yet\n");
                    exit(121);
                    break;
                case 5:
                    TranslatePoint(pt,a,b,c);
                    break;
                case 6:
                    ScalePoint(pt,a,b,c);
                    /* implement Al Barr's paper for computing
                        the normals of a scaled surface */
                    break;
            }
        }
    }
}

/***************************************************************
*/
Finally, I decided to load all the Polygon3D data structures in a temporary Polygon data structure. This quite simpler data structure is passed to the projection, clipping, and shading routines.

In order to apply various graphics transformations to the polygonal surface representation, we need to transform the thread points using either the Threads or the Tiles data structure. In this way, we lose the efficiency provided by the B-spline representation, where we just need to transform the control mesh of the surface. However, we don’t need to recompute the B-spline surface again, using the transformed control points.

```c
void GLInitialise(int argc, char** argv)
{
    char** poo;
    screen_width=800;
    screen_height=800;
    view_r1=0;
    view_r2=0;
    view_scale=2.0;
    view_tx=1.8;
    view_ty=1.0;
    obj_bend=0.0;

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(screen_width, screen_height);
    glutInitWindowPosition (100, 200);
    glutCreateWindow("Elasticas");
    glutDisplayFunc(draw);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutIdleFunc(idle);
    glutMotionFunc(mouseMove);

    initMenu();

    /* Setup lighting */
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
    /* Perform Normalization of my normals */
    glEnable(GL_NORMALIZE);
    /* Enable z-buffering */
    glEnable(GL_DEPTH_TEST);

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glShadeModel(GL_SMOOTH);
    glClearColor (0.0, 0.0, 0.0, 0.0);

    /* Setup Texture */
    load_texture("./cloth/CLOTH027.rgb", 160);
}```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glGenTextures(1, texName);
glBindTexture(GL_TEXTURE_2D, texName[0]);

GLuint texName[1];

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texwidth, texheight, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture);

/* Setup 3D viewing parameters */
}

/*******************************
** Set up transform to rotate the object */

void mouse(int b, int s, int x, int y) {
origx = x;
origy = y;
view_tx2=view_tx;
view_ty2=view_ty;
view_r12=view_r1;
view_r22=view_r2;
obj_bend2=obj_bend;
}

void mouseMove(int x, int y) {
switch(mousemode) {
case scale:
view_scale=y/100.0;
glutPostRedisplay();

}
break;
case rotate:
    view_r1=view_r12-(y-origy)/2.0;
    view_r2=view_r22-(x-origx)/2.0;
    glutPostRedisplay();
    break;
case translate:
    view_tx=view_tx2-(x-origx)/400.0;
    view_ty=view_ty2-(origy-y)/400.0;
    glutPostRedisplay();
    break;
case bend:
    if (bobsidle) {
        obj_bend=(y-origy)/200.0;
        ControlDeformation(obj_bend);
        ReconstructSurfaces();
        glutPostRedisplay();
        bobsidle=0;
    }
    break;
}
/***************************************************************************/
void draw(void) {
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    projection();
    glClear (GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    DisplayPolygonalSurfaces();
    glFlush();
    glutSwapBuffers();
}
/***************************************************************************/
void initMenu(void) {
    int sub1, sub2, sub3, sub4;

    sub1=glutCreateMenu(menu_draw);
    glutAddMenuEntry("Smooth Shaded",smooth);
    glutAddMenuEntry("Flat Shaded",flat);
    glutAddMenuEntry("Wireframe",wireframe);
    glutAddMenuEntry("========",999);
    glutAddMenuEntry("Surface",dispsurface);
    glutAddMenuEntry("Threads",dispthreads);

    sub2=glutCreateMenu(menu_colour);
    glutAddMenuEntry("Red",red);
    glutAddMenuEntry("Dark Red",darkred);
    glutAddMenuEntry("Green",green);
    glutAddMenuEntry("Dark Green",dargreen);
    glutAddMenuEntry("Blue",blue);
    glutAddMenuEntry("Dark Blue",darkblue);
    glutAddMenuEntry("Yellow",yellow);
    glutAddMenuEntry("Dark Yellow",darkyellow);
    glutAddMenuEntry("Custom Colour",customcolour);

    sub3=glutCreateMenu(menu_light);
    glutAddMenuEntry("Position Light",lightpos);
    glutAddMenuEntry("Direction Light",lightdir);

    sub4=glutCreateMenu(menu_mode);
    glutAddMenuEntry("Zoom",scale);
glutAddMenuEntry("Translate", translate);
glutAddMenuEntry("Rotate", rotate);
glutAddMenuEntry("Bend", bend);

glutCreateMenu(menu);
glutAddSubMenu("Draw Mode", sub1);
glutAddSubMenu("Colour", sub2);
glutAddSubMenu("Light", sub3);
glutAddSubMenu("Mouse Mode", sub4);
glutAddMenuEntry("Reset", reset);
glutAddMenuEntry("DeformCurves", deformcurves);
glutAddMenuEntry("--------------", 999);
glutAddMenuEntry("Quit", quit);
glutAttachMenu(GLUT_MIDDLE_BUTTON);
}
***************************************************************************/

void menu(int id) {
  if (id==quit) exit(1);
  if (id==reset) {
    view_r1=0;
    view_r2=0;
    view_scale=1.0;
    view_tx=0.0;
    view_ty=0.0;
    obj_bend=0.0;
    draw();
  }
}
/***************************************************************************/

void menu_draw(int id) {
  if (id==smooth) {
    glShadeModel(GL_SMOOTH);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    draw();
  }
  if (id==flat) {
    glShadeModel(GL_FLAT);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    draw();
  }
  if (id==wireframe) {
    glShadeModel(GL_FLAT);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    draw();
  }
  if (id==dispsurface) {
    displaysurface=1-displaysurface;
    draw();
  }
  if (id==dispthreads) {
    displaythreads=1-displaythreads;
    draw();
  }
}
/***************************************************************************/

void menu_colour(int id) {
  GLfloat r,b,g;
  switch (id) {
    case red:
glColor3f(1.0,0.0,0.0);
break;
case darkred:
    glColor3f(0.5,0.0,0.0);
    break;
case green:
    glColor3f(0.0,1.0,0.0);
    break;
case darkgreen:
    glColor3f(0,0.5,0.0);
    break;
case blue:
    glColor3f(0.0,0.0,1.0);
    break;
case darkblue:
    glColor3f(0.0,0.0,0.5);
    break;
case yellow:
    glColor3f(1.0,1.0,0.0);
    break;
case darkyellow:
    glColor3f(0.5,0.5,0.0);
    break;
case customcolour:
    printf("provide RGB values [r, g, b], 0.0-1.0\n");
    scanf("%f, %f, %f", &r, &g, &b);
    glColor3f(r,g,b);
    break;
}
draw();
}
/***************************************************************************/
void menu_light(int id) {
    GLfloat x,y,z,w;
    if (id == lightpos) w=1.0; else w=0.0;
    printf("provide x, y, z values [x, y, z]\n");
    scanf("%f, %f, %f", &x, &y, &z);
    light_position[0]=x;
    light_position[1]=y;
    light_position[2]=z;
    light_position[3]=w;
    draw();
}
/***************************************************************************/
void menu_mode(int id) {
    mousemode=id;
}
/***************************************************************************/
void idle(void) {
    bobsidle=1;
}
/***************************************************************************/
void load_texture(char* file, int alpha)
{
    FILE *File;
    int size, x;

if((File=fopen(file, "r"))==NULL) {
    fprintf(stderr,"Error loading texture file\n");
    exit(-1);
}

/* get the width and height of the texture from the first 4 bytes of the file */
texwidth=fgetc(File)*256+fgetc(File);
texheight=fgetc(File)*256+fgetc(File);
texture = (GLubyte *)malloc(4*texwidth*texheight*sizeof(GLubyte));

size=texwidth*texheight;
/* read the rest of the file into an array of pixel values */
fread(texture, size*4, 1 , File);
fclose(File);

/* It might be necessary to alter the alpha values */
for (x=0; x<texwidth*texheight;x++) {
    texture[3+4*x]=alpha;
}

/***************************************************************************/
This is the layer.c file, this file contains the functions that I wrote to build an outer layer over the underlying layer of elastics.

#include <math.h>
#include "headers.h"
#include "xgr.h"
extern float Length3();
extern coord *CoordAlloc();
extern tcoord *TCoordAlloc();
extern hcoord *HCoordAlloc();
extern Elastica *ElasticaAlloc();
extern ElasticaPt *ElasticaPtAlloc();
extern Thread *ThreadAlloc();
extern Solution *SolutionAlloc();
coord PN,RP;

ElasticaL *ElasticaLAlloc()
{ return(ElasticaL *)malloc(sizeof(ElasticaL)); }

ConstructLongElasticas(sf,ff1,ff2)
Surface *sf;
float *ffl,ff2;
{
    int i, j, idx;
    Thread *t;
    Elastica **els;

    for(i=0; i<sf->RegionsNr; i++)
        for(j=0; j<sf->InNr*2+3; j++){
            idx = i*(sf->InNr*2+3)+j;
            t = sf->Threads[idx];
            ConstructLongThread(t,ff1,ff2);
        }

    ElasticaPt *ElasticaPtCopy(e)
    ElasticaPt *e;
    {
        ElasticaPt *t;
        t=ElasticaPtAlloc();
        t->pt=HCoordAlloc();
        t->pt->x=e->pt->x;
        t->pt->y=e->pt->y;
        t->pt->z=e->pt->z;
        t->pt->w=e->pt->w;
        if (e->nm !=NULL ) {
            /* end points dont have calculated normal yet so ignore them */
            t->nm=HCoordAlloc();
            t->nm->x=e->nm->x;
            t->nm->y=e->nm->y;
            t->nm->z=e->nm->z;
            t->nm->w=e->nm->w;
        }
        return(t);
    }

}
InterpPoints(hcoord* res, hcoord* pt1, hcoord* pt2, float s) {
    res->x=(1-s)*pt1->x + s*pt2->x;
    res->y=(1-s)*pt1->y + s*pt2->y;
    res->z=(1-s)*pt1->z + s*pt2->z;
}

/************************************************************************/
/* Given an 11 point elastica, this returns
the point and normal, t of the way along the elastica, */

ElasticaPt *InterpolateElastica(el, t)
Elastica *el;
float t;
{
    int i, j;
    float s;
    ElasticaPt *res;
    
    if (t==0.0) {
        return(ElasticaPtCopy(el->elpts[0]));
    } else if (t==1.0) {
        return(ElasticaPtCopy(el->elpts[10]));
    } else {
        res=ElasticaPtAlloc();
        res->pt=HCoordAlloc();
        res->nm=HCoordAlloc();
        s=10.0/(el->elptsnr-1.0);
        j=rint(floor(t*s));
        InterpPoints(res->pt,(el->elpts[j])->pt,(el->elpts[j+1])->pt,t*s-j);
        InterpPoints(res->nm,(el->elpts[j])->nm,(el->elpts[j+1])->nm,t*s-j);
    }
    return(res);
}

/************************************************************************/
float DotProduct(v1, v2)
hcoord *v1, *v2;
{
    return(v1->x*v2->x + v1->y*v2->y + v1->z*v2->z);
}

/************************************************************************/
ConstructLongThread(t, ff1, ff2)
Thread *t;
float ff1, ff2;
{
    int i, j;
    float nr;
    int l1, l2, l3, l4;
    Elastica *el;
    ElasticaL *longel;
    float dist, length, length2;
    hcoord *p;
    ElasticaPt *elpt;

    t->longels[0]=ElasticaLAlloc();
    t->longels[1]=ElasticaLAlloc();
    t->longels[2]=ElasticaLAlloc();
    t->longsls[0]=SolutionAlloc();
    t->longsls[1]=SolutionAlloc();
    t->longsls[2]=SolutionAlloc();
}
/* find the elastica numbers of the key curves */
nr=t->elsnr;
l1=0;
l2=rint(ff1*nr);
l3=rint(ff2*nr);
l4=nr;

/* Construct the long elasticas */
ConstructLongElasticaValues(t->longels[0],t->longsls[0],t,l1,l2-1);
ConstructLongElasticaValues(t->longels[1],t->longsls[1],t,l2,l3-1);
ConstructLongElasticaValues(t->longels[2],t->longsls[2],t,l3,l4-1);
DeformLongThreadElasticas(t);
ConstructLongElasticaMapping(t->longels[0],t,l1,l2-1);
ConstructLongElasticaMapping(t->longels[1],t,l2,l3-1);
ConstructLongElasticaMapping(t->longels[2],t,l3,l4-1);
}
/**************************************************************************/

ConstructLongElasticaValues(longel,sl,t,a,b)
ElasticaL *longel;
Solution *sl;
Thread *t;
int a, b;
{
int i, j;
Elastica *el;
Elastica **shortels;
ElasticaPt *elpt;
hcoord *p;
float length, length2, dist;
p=HCoordAlloc();
longel->el=ElasticaAlloc();
el=longel->el;
shortels = t->els;
el->type3=1;
el->type4=1;
el->deformed=0;
el->firstp=ElasticaPtCopy((*(shortels+a))->firstp);
el->lastp=ElasticaPtCopy((*(shortels+b))->lastp);
el->length=Length3(el->firstp->pt, el->lastp->pt);
el->dist=el->length;

/* the long elastica will deform in the same plane as the underlying short elasticas */
el->a=(*)(shortels+a)->a;
el->b=(*)(shortels+a)->b;
el->c=(*)(shortels+a)->c;
el->d=(*)(shortels+a)->d;

/* set up the solution for the long elastica */
sl->pnr = 11;
sl->t1 = 2;
sl->t2 = 2;
BryCond(sl,45.0,0.0,0.0,0.0,-0.0);

/* Find the initial solution */
ProcessElastica(el,sl);
longel->elnr=b-a+1;

} ConstructLongElasticaMapping(longel, t, a, b) ElasticaL *longel;
Thread *t;
int a,b;
{
int i,j;
Elastica *el;
hcoord *p;
ElasticaPt *elpt;
float length, length2, dist;
/* construct mapping between long and short elasticas */
el=longel->el;
/* calculate the total length of the thread */
length=0;
for (j=0; j<longel->elnr; j++) {
    longel->elMap[j]=t->els[j+a];
    length+=longel->elMap[j]->length;
}
/* here we calculate the point on the elastica and the distance along the bend plane normal from that point that intersects the endpoint of the underlying elasticas. This implementation is only approximate, and may cause small changes to the configuration of the surface, but these should not be visually noticeable */
length2=0;
p=HCoordAlloc();
for (j=0; j<longel->elnr; j++) {
    longel->elMapT[j]=length2/length;
    /* p is now set to the vector between the point on the long elastica and the corresponding first point of the short elastica */
    elpt=InterpolateElastica(el,longel->elMapT[j]);
    p->x=longel->elMap[j]->firstp->pt->x - elpt->pt->x;
    p->y=longel->elMap[j]->firstp->pt->y - elpt->pt->y;
    p->z=longel->elMap[j]->firstp->pt->z - elpt->pt->z;

    /* we now find the length of the vector p in the direction of the bend plane normal */
    dist=DotProduct(p, elpt->nm);
    /*printf("dist = %f\n",dist);*/
    longel->elDist[j]=dist;
    length2=length2+longel->elMap[j]->length;
}
/* Special case for when we use the last point of the last elastica */
j=longel->elnr-1;
longel->elMapT[j+1]=1.0;
elpt=InterpolateElastica(el,longel->elMapT[j+1]);

7-75
p->x=longel->elMap[j]->lastp->pt->x - elpt->pt->x;
p->y=longel->elMap[j]->lastp->pt->y - elpt->pt->y;
p->z=longel->elMap[j]->lastp->pt->z - elpt->pt->z;
dist=DotProduct(p, elpt->nm);
printf("dist end = %f\n",dist);
longel->elDist[j+1]=dist;
}
Chapter 8 - Bibliography

1 Chu, C.c., M.M. Platt, W.J. Hamburger
“Investigation of the factors Affecting the Drapeability of Fabrics”

2 Ng, Hing N, R.L. Grimsdale
“Computer Graphics Techniques for modelling cloth”

3 Weil, J.
“The synthesis of cloth objects”

4 Hsu, M. B.
“An iterative graphics program for equilibrium shape determination for tensioned fabric structures”

5 Terzopoulos, D., J. Platt, A. Barr, K. Fleischer
“Elastically Deformable Models”

6 Terzopoulos, D., K. Fleischer
“Modeling Inelastic Deformations: Viscoelasticity, Plasticity, Fracture”

7 Terzopoulos, D., A. Witkin
“Physically-based models with rigid and deformable components”
Proc, Graphics Interface ’88, June 1988

8 Breen, D.E., D.H. House, P.H. Getto
“A physically-based particle model of woven cloth”

9 Breen, D.E., House, D.H., Wozny
“A Particle-Based Model for Simulating the Draping Behaviour of Woven Cloth”

10 Breen, D.E., House, D.H., Wozny, M.J.
“Predicting the Drape of Woven Cloth Using Interacting Particles”

11 Kawabata, S.
“The Standardization and Analysis of Hand Evaluation”
The Textile Machinery Society of Japan, 1980

“A particle-Based Model for Simulating the Draping Behaviour of Woven Cloth”

13 Eberhardt, B., Weber, A., Strasser, W.
“A fast, flexible particle-system for cloth draping”
IEEE computer graphics and applications, Vol 16, pp. 52059, 1996

14 Baraff, D., Witkin, A.
“Large Steps in Cloth Simulation”
15 Hadap, S., Bangerter, E., Volino, P., Magenat-Thalmann
“Animating Wrinkles on Clothes”
MIRALab, university of Geneva, 1999

16 Shinohara, A, Q. Ni, M. Takatera
“Geometry and mechanics of the buckling wrinkle in fabrics.”
Part I: Characteristics of the buckling wrinkle
Part II: Buckling model of a woven fabric cylinder in axial compression

17 Grosberg, P., Swani, N.M.
“The Mechanical Properties of Woven Cloth, Part II: The buckling of Woven Fabrics”

18 Aono, M., P. Denti, D.E. Breen and M.J. Wozny