Modeling the Appearance of Cloth

by

Carl Richard Feynman

B.S., Massachusetts Institute of Technology
(1983)

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE
OF

MASTER OF SCIENCE

IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1986

©Massachusetts Institute of Technology 1986

Signature of Author

....

Artificial Intelligence Laboratory
May 9, 1986

Certified by

....

....
David Zeltzer
Assistant Professor of Computer Graphics
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

ARCHIVES
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 28 1987

LIBRARIES

Modeling the appearance of cloth

by

Carl Feynman

Submitted to the Department of Electrical Engineering and Computer Science on May 9, 1986 in partial fulfillment of the requirements of the degree of Master of Science.

Abstract

A computational model for the mechanical behavior of textiles sufficiently accurate for computer synthesis of realistic scenes was implemented. The model computes static configurations of cloth draped over rigid objects by gravity. The mechanical properties of the cloth which are simulated include elasticity, bending strength, density and resistance to buckling. By changing parameters of the program, various types of cloth can be simulated. The output of the program is a shape description which can be used as input to a variety of rendering programs. The algorithm employed to simulate the cloth is a variation of the multigrid full approximation scheme.

Thesis Supervisor: David Zeltzer
Title: Assistant Professor of Computer Graphics

Contents

1 The problem **4**

2 The approach **6**
2.1 Cloth is simulated by minimizing its energy 6
2.2 Representation of the Cloth 7
2.3 Methods of Minimization 9
2.4 The Energy of the Cloth 10
2.5 Properties of the Relaxation Method 12
2.6 Hanging up the Cloth 13
2.7 Rendering 14

3 Relations to Other Work **14**

4 Energy Minimization **17**
4.1 Relaxing One Point 17
4.2 Relaxation Order 26

5 The Energy Expression **28**
5.1 The Total Energy 30
5.2 Gravity 31
5.3 Bending 32
5.4 Strain 38
5.5 Buckling 45
5.6 Problems with the energy expression 47

6 The Multigrid Method **51**
6.1 The Schedule of Relaxation 52
6.2 The Coarsening and Refinement Operators 55
6.3 Problems with the Multigrid Method 58

7 External constraints **61**
7.1 Fixed Points 61
7.2 Solids 62

8 Directions for the Future **67**

9 Acknowledgments **69**

A The Program **72**

List of Figures

1 20
2 23
3 24
4 24
5 25
6 Row-major and red-black relaxation orders 27
7 Parameters of the energy expression 29
8 Evaluating the curvature at a point 33
9 A flat piece of paper 34
10 A bent piece of paper 35
11 36
12 A stretched piece of cloth 39
13 E and σ in action 40
14 42
15 Cloths with different elasticities 44
16 The energy of a plate as a function of compression. 46
17 Energy as a function of length for long and short buckling springs. 48
18 A variety of relaxation schedules 53
19 A square of cloth supported at eight points along one edge. 60
20 Cloth draped over a sphere 64
21 A piece of cloth supported at three points. 65
22 A square piece of cloth compressed along its upper edge and crumpled on the floor below. 66

1 The problem

In the last few years, advances in algorithms have combined with the accelerating growth of number-crunching power to permit the simulation of scenes with good realism and considerable complexity. The objects in these scenes are, however, restricted to a small subset of the things we see around us. Rocks, trees, water, mirror balls and marble vases may suffice for Victorian gardens, but they are not what most of us want to make movies about. Fortunately for the ambitious computer artist, the range of objects we know how to simulate is steadily being extended. This thesis is part of that extension.

The goal of this thesis is to model the shape of textiles with sufficient accuracy that people are willing to suspend disbelief. More precisely, the goal is to simulate the shape of thin, flexible membranes under the influence of force fields and rigid bodies. These includes tablecloths, flags, and t-shirts. In this thesis, only stationary cloth will be considered, though the techniques might eventually be extended to support animation. Our requirements for realism are slack: anything that looks like cloth to the untutored observer is a success.

Some parts of the behavior of cloth are ignored in this thesis. This is either because they are unimportant or because I didn't know how to model them. Clothing, which is the type of textile which is needed most in animation and computer graphics, consists of a number of pieces of material, of a wide variety of shapes, sewn together along seams. This thesis models only square pieces of cloth, isolated from other cloth. The techniques described herein could, however, be easily extended to model complex garments, as discussed in 8. All cloth in this thesis is considered to be perfectly flat in its relaxed state. Thus, I am unable to model clothing which contains creases even in its relaxed state. Unfortunately, the simulation of skin is beyond the techniques described in this thesis. The behavior of skin is largely determined by the flesh behind it. Without a model for the flesh, a model of skin is useless. For this same reason, I am unable to model fabric backed by soft filling, as might be found in a couch or winesack.

Since the mathematical techniques described in this thesis are new to the field of computer graphics, it was felt that an attempt to produce a package actually usable by an animator would be premature. Hence, no attempt has been made to produce a software package suitable for general

use. The software described in this thesis was written in Common Lisp on a Symbolics 3600 computer.

2 The approach

This section is an overview of the entire program, intended to provide the background neccesary to understand the more detailed description of the later sections.

2.1 Cloth is simulated by minimizing its energy

Real physical cloth is stationary when its energy is minimized. It can be analyzed by considering a function mapping configurations of the cloth into energies. A local minimum of this function represents a stationary state of cloth. This is because at a local minimum, any small movement of the cloth results in an increase of energy, and hence will not occur. This is strictly true only in the absence of friction, viscosity and other effects that can convert motion into heat. Such effects will be ignored in this thesis. Thus, from a mathematical point of view, the problem of producing realistic shapes of cloth can be described as finding mappings of a segment

of 2-space (the cloth) into 3-space (object space) which are local minima of a function from such mappings into the real numbers (the energy function) subject to certain constraints (the effects of the world on the cloth). A shape which is such a local minimum is a solution of our problem. Clearly, it is not necessary to find an exact solution; all we need is something that is close enough to reality that people are willing to accept it as cloth. This is fortunate, since finding an exact solution is usually impossible.

Note that it is not necessary to find the global minimum of the energy, but only a local minimum. This is because cloth usually spends its time in a local minimum. Indeed, a global minimum often looks less "realistic" than a local minimum. For example, the global minimum configuration for a napkin lying on a table is to lie perfectly flat. A crumpled, balled-up napkin is in a local minimum, but is much more likely to actually occur in reality than the perfectly flat state.

2.2 Representation of the Cloth

The similarity of the model to real cloth is determined by the similarity between the energy function of the model and the energy function of real

cloth. It is clearly impossible to model the energy expression of a real piece of cloth in its full complexity, since this would involve simulating the quantum-mechanical wave function of the cloth. Fortunately, we have to model the energy expression only to the extent that it produces results that look like real cloth. This permits large simplifications in the representation of cloth and the form of the energy expression. The choice of the correct energy expression is discussed in section 5.

The cloth is approximated by a grid of points spread across its surface. In a flat, unstretched piece of cloth these points are arranged in a square grid of uniform spacing. The configuration of the cloth is described by the positions of these points. The finer the grid, the more accurately the shape of the cloth can be described. Bumps in the cloth smaller than about twice the grid spacing cannot be adequately described, because they will too easily "fall between" the sample points. Thus, we would like the grid spacing to be very fine. On the other hand, the complexity of the calculation increases as more points are added. The problem of balancing these opposing pressures will occupy much of the remainder of this thesis.

2.3 Methods of Minimization

We want to find a local minimum of the energy with respect to small changes in position. We can do this by changing the shape of the cloth in the direction of decreased energy. As long as the energy keeps going down, we will eventually come as close as we wish to a local minimum. We can decrease the energy of the cloth by deforming a small piece of the cloth to a position which decreases the energy of the whole cloth. Since we are approximating the cloth as a grid of points, the way to move a small piece of the cloth is to move one of the grid points. By moving each of the grid points in turn toward a position which decreases the energy of the whole cloth, we will move toward a configuration of minimum energy. This process is known as *relaxation*, since it changes the shape of the cloth in the direction of a more relaxed configuration.

The energy expression of the model is a function which takes the approximate representation of the cloth's shape and computes the energy that a real piece of cloth would have if it were put into such a shape. There are several desirable properties such an energy function should have. First, it should be close enough to the real energy expression that a configuration of

discrete cloth which minimizes the approximate energy expression should be close to a shape of the actual cloth which minimizes the real energy expression. This will ensure that the output of the program will be close to the shapes that a real cloth will adopt when relaxed. Second, it should be relatively insensitive to the grid size adopted for modeling the cloth. The reason for this will become clear in the discussion of the multigrid method below. Third, it should be quick to compute the change in energy produced by moving a single point on the cloth to a new position, since this step is the "inner loop" of the simulation, and must be performed many times for each point in the cloth.

2.4 The Energy of the Cloth

There is a considerable body of literature on energy expressions for cloth and similar materials. The theoretical behavior of plates of elastic material is well understood [7], [8]. In addition, the empirical behavior of textiles under a variety of forces has been studied extensively [5], [1]. The theoretical and empirical analyses agree that the energy of real textiles is determined by two effects internal to the cloth, as well as the forces placed

upon the cloth by its environment. The internal effects are, first, the strain energy, produced by compressing or stretching the cloth in its own plane, and, second, the bending energy produced by bending the cloth out of its plane. Other forces that impinge on the cloth from without must also be considered in the potential energy calculation. For example, a cloth in a gravitational field will decrease its potential energy as it moves downward. In this thesis, gravity is assumed to be the only external force.

In real cloth, each tiny piece of cloth makes its own contribution towards the total energy. Widely separated parts of the cloth have no effect on each other's energies. We can take advantage of this property to simplify the computation of the approximate energy expression. If we approximate real cloth as a grid of points, the energy of the entire cloth depends only on the distances and angles between points nearby in the grid. Hence, the evaluation of the change in the energy of the entire cloth produced by moving a single point requires looking at only the point and a few of its nearest neighbors, since the energy of more distant parts of the cloth will not be affected by the change in position of the point.

2.5 Properties of the Relaxation Method

The simplest way to approach the relaxation problem is to relax every point in the cloth in turn, until the cloth as a whole comes adequately close to a state of minimum energy. Unfortunately, this approach takes far too long on a cloth of any size.

A deviation from the correct solution whose elimination requires a concerted movement of many nearby points in the same direction can take a very long time to occur. Features of the final solution whose size is n grid points take order n^2 relaxation sweeps to appear. The relaxation of a single point, with its neighbors held fixed, will remove only those errors apparent in the local neighborhood. When the error is spread over a large area, the deviation from perfection of any local area may be very small. In such a case, the motion of a point upon being relaxed will be very small.

A grid is best at modeling features whose typical size is about twice the grid spacing. Features smaller than this cannot be accurately represented because the grid is too coarse for them. Features much larger than this require excessive computation to produce.

The solution to the problem of quickly computing features on a variety

of scales is to use a variety of grids, with different interpoint spacing. A grid with the points far apart can be used to find the general shape of the relaxed cloth, and a finer grid to create small-scale features. If we use a hierarchy of grids, each twice as coarse as the next finest, every feature will have a size close to the grid spacing for one of the grids, and hence will be produced with little computational effort. Methods that use a variety of grids in this way are known as *multigrid methods.*

2.6 Hanging up the Cloth

The shapes that real textiles adopt are a compromise between the tendencies of the material and the constraints imposed by the environment. In this system, there are two types of constraint: fixed points and solids. Fixed points are points on the cloth which are forbidden to move. Solids are volumes which the cloth is forbidden to enter. Fixed points can model the hooks which suspend curtains or the frames from which hammocks hang. Solids model walls, floors, tables, the human body, and other things commonly draped by cloth. Because there is no friction, fixed points are often needed to keep the cloth from sliding off the solids it is draped over.

2.7 Rendering

After the algorithm has generated a shape for a piece of cloth under the desired conditions, there remains the problem of rendering. Rendering is not a major concern of this thesis, but without it we would be unable to see the results of our algorithm. There are two systems which I use to display the products of the algorithm. The first is a special-purpose wire-frame displayer intended to ease debugging and display intermediate results. The second is an interface to the Thinking Machines Corporation 3D Graphics Toolkit, which can display the cloth as Gouraud-shaded polygons, as well as interactively manipulate viewing parameters. The drawings of cloth in this thesis were generated by the Toolkit and printed on a QMS laser printer.

3 Relations to Other Work

The only other paper I am aware of which attempts to model the behavior of draped cloth is [11]. The approach described therein involves three different methods to produce details at three different scales. The largest scale features are created by succesive construction of catenaries. This creates

initial positions for a grid of points. These points are then relaxed to create plausible features on an intermediate scale. Finally, cubic spline interpolation creates points to fill in the grid. The relaxation stage is similar to the methods in this thesis, but because only one grid is used, creation of large fatures takes too long to do directly. The method does not explicitly involve the use of an energy expression in the relaxation. Rather, the one-point relaxer tries to satisfy mechanical constraints directly.

There are many papers on the application of computer graphics to the design of textiles, but they are all concerned with the rendering of flat pieces of cloth, usually for purposes of computer-aided "prototyping" for textile designers. Such programs would be perfectly complimentary to mine. as I do not approach the problem of rendering at all.

Multigrid methods are commonly used to solve partial differential equations on two- or three-dimensional domains. They are also applicable to the problem of finding eigenfunctions or, as in my case, minimizing the value of a functional. The use of the multigrid full approximation scheme in this thesis is a straightforward application of the principles in the literature [2], [10]. My contribution was restricted to the choice of interpolation and re-

finement operators and the choice of schedules, from the many offered in the literature.

The energy expression in this thesis was created by a combination of theory and experiment. Energy expressions for thin plates of ideal materials are well known in the literature [7], but the behavior of real cloth is not so well understood [5]. I initially tried to create an energy expression by discretizing the continous energy expression for ideal materials. Integrals were replaced by sums, derivatives with finite differences, and so on. This approach was unsuccesful. The resulting behavior was not at all clothlike. At this point, I changed strategies and simply invented a plausible energy expression. After a certain amount of adjustment, I ended up with the one described in this thesis. I was surprised to discover that mathematical analysis showed my energy expression to yield behavior similar to that of the ideal material in the simple cases analyzed in Section 5. In places where the behavior differed, the difference was in the direction of making the behavior more like the behavior of real cloth and less like that of an ideal elastic material.

4 Energy Minimization

Despite its name, relaxation is not a simulation of the way cloth actually moves when relaxing. It is a method used to approach a relaxed state of the cloth, but the method whereby the cloth approaches this state is distinctly unphysical. The relaxation algorithm may be divided into two parts: the method of relaxing one point and the way in which these one-point relaxations are combined to relax the entire cloth.

4.1 Relaxing One Point

Relaxing a single point is the process of moving it so that the energy of the cloth of which it is a part is decreased. Relaxing every point in the cloth once is called a *relaxation sweep.* As relaxation sweeps are performed on the cloth, its shape approaches a local minimum. This is guaranteed as long as each point relaxation decreases the energy of the cloth. When, below, I refer to decreasing the energy of a point, I really mean the energy of the cloth of which it is a part, but it is easier to simply talk about the energy of a point. Since the change in the energy of a point when moved depends only on the old and new positions of the point and the positions

of its neighbors, it is convenient to regard the process of relaxing a point as being dedicated to reducing the energy of that point, rather than the energy of the whole cloth.

The method used to relax a single point should be fast and should also reduce the energy of the point as much as possible. These two requirements are difficult to reconcile, since a simple one-point relaxer will run faster, but a more complex one will be more likely to find the most relaxed position for a point. The energy of the point as a function of position (with the neighbors held fixed) has a global minimum. The one-point relaxation method should come as close as possible to moving the point to this minimum. On the other hand, the more complicated the one-point relaxer, the more time it takes to perform a sweep, and the slower the program. Even with a perfect one-point relaxer, multiple sweeps are necessary. Experiments are necessary to settle on an ideal one-point relaxer.

In order to preserve modularity within the program, it seemed wise to design a one-point relaxer as independent of the force law as possible. The simplest possible interface to the force law is to pass it the locations of a point and its neighbors, and have it return the energy of the point. Thus

the one-point relaxer should, in general, evaluate the energy of its point at a variety of positions, and then guess at the location of lowest energy using only the information on positions and energies. The problem then becomes one of finding the minimum of a function in three dimensions using the smallest possible number of sample points.

The method used to relax a single point is, roughly, to first find the direction in which the point would most like to move, and then move it in that direction such a distance that its energy is minimized. The problem of searching for a minimum in three-dimensional space is thus split into two subproblems: finding an optimum direction (a two-dimensional search), and then finding a distance which minimizes the energy (a one-dimensional search.)

The direction in which the point is moved is the direction of the gradient of the energy as a function of position. This is the direction in which the energy of the point decreases fastest with position. It is also the direction of the force the point would feel if it were allowed to move freely with its nieghbors held fixed. Within an infinitesimal neighborhood of the position

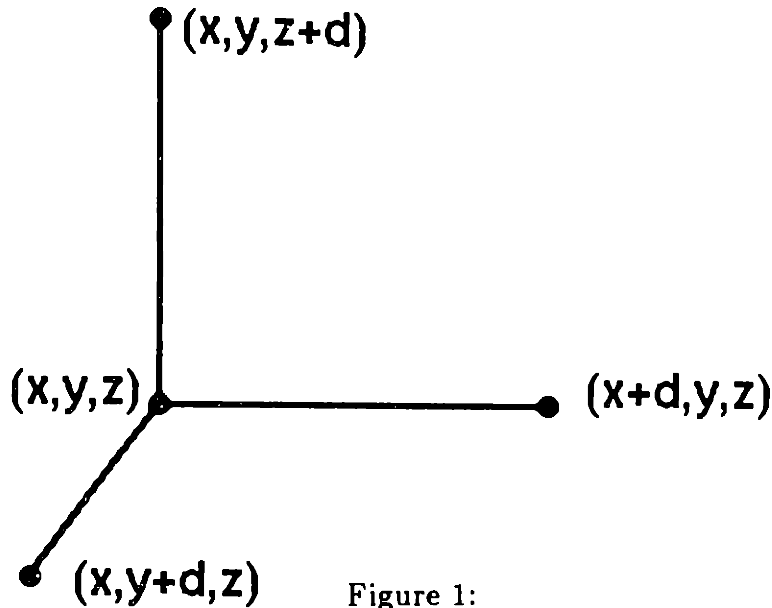


Figure 1:

of the point, the energy is a linear function of position of the form

$$E = f_x dx + f_y dy + f_z dz + c$$

. The gradient of this function is the vector (f_x, f_y, f_z). To determine the values of f_x,f_y and f_z the program evaluates the energy of the point at its original position and at positions displaced a very small distance along the coordinate axes, as shown in Fig. 1. The force vector is approximately

$$f_x = \frac{E(x + \Delta, y, z) - E(x, y, z)}{\Delta}$$

$$f_x = \frac{E(x, y + \Delta, z) - E(x, y, z)}{\Delta}$$

$$f_x = \frac{E(x, y, z + \Delta) - E(x, y, z)}{\Delta}$$

If Δ were infinitesimal, this would be exactly the force vector. If Δ is too small, however, floating-point underflow in the calculation of the energy

will produce errors. The program uses $\Delta \approx 10^{-4}$, which seems sufficiently small for all practical purposes.

Moving along the force vector is not quite the right thing to do. This problem is resolved by moving the point during the relaxation process not along the force vector but along a vector displaced from it by a random angle and direction. The problem is that there must be some way of breaking the symmetry of a flat piece of cloth. A vertically suspended square of fabric will, if compressed laterally, buckle to form a series of vertical folds. In simulated cloth, however, a point in the middle of a compressed sheet feels only the force of gravity pulling straight down. It could reduce its energy by moving out of the plane in either direction, but the force vector on the point is directed straight down. If points moved only along the force vector, symmetric pieces of cloth would never lose their symmetry. A random perturbation is added to the force vector every time a points is relaxed, even though in most cases it is not needed.

The choice of the correct mean angle for the random displacement was made experimentally. If the angle is too small, symmetric situations take too long to break into realistic asymmetry. If the angle is too large, the

relaxation process takes longer to converge and produces irregular and unrealistically bumpy results if it is not allowed to run long enough. The best value seems to be a mean deviation of about 7°.

Once the program has selected the direction along which the point is to be moved, the problem becomes one of deciding on the position along a line which minimizes the energy. The line is described by the parametric equations

$$x = x_0 + f_x\prime t$$

$$y = y_0 + f_y\prime t$$

$$y = y_0 + f_y\prime t$$

where x_0, y_0, z_0 is the original location of the point and $f_x\prime, f_y\prime, f_z\prime$ is a unit vector in nearly the direction of the force on the point. The graph of energy with respect to t looks something like the heavy line in Fig. 2. We can find the minimum of this curve by sampling it at several points, and fitting a polynomial to these points. If the polynomial is a good approximation, the minimum of the polynomial will be close to the true minimum of the curve. Experiment shows that a quadratic approximation is sufficiently

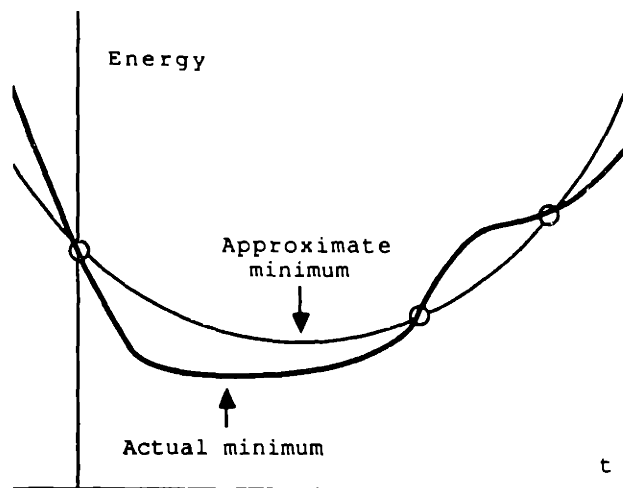


Figure 2:

accurate. A quadratic approximation requires evaluating the energy for three values of t. The sample points are shown as os in Fig. 2 and the parabola drawn through them as the thin line.

This method is vulnerable to errors. If the energy curve is not shaped at all like a parabola, the result will not be near the minimum. If the parabola is concave upward instead of downward, this method will find the maximum rather than the minimum. The method becomes sensitive to small deviations in the curve if the sample points are too close together (Fig. 3) or too far from the minimum (Fig. 4).

We can take advantage of several properties of the curve to avoid these difficult cases. First, it is known to have a negative first derivative at $t = 0$. Second, for sufficiently large values of t, the first derivative is positive. Thus

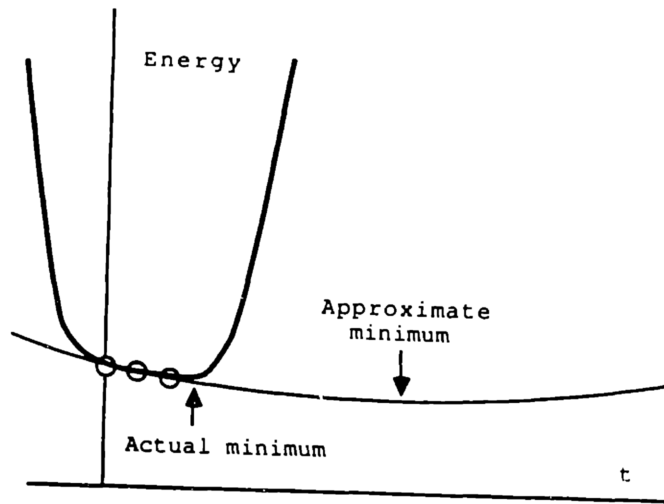


Figure 3:

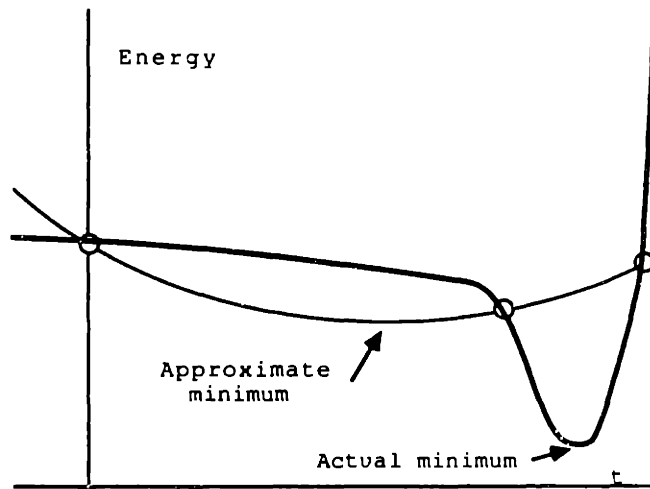


Figure 4:

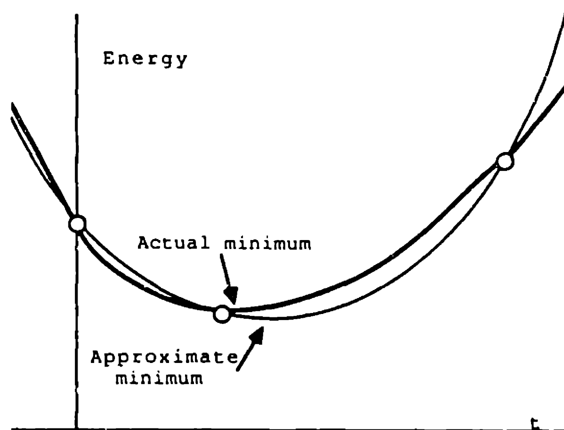


Figure 5:

we know that the minimum is somewhere between 0 and the value of t for which the energy is equal to the energy at $t = 0$. We choose the sample points to be $t = 0$, $t = k$ and $t = 3k$ for some value of k, and further require that the sample point at $t = k$ have an energy lower than that of the point at $t = 0$, and that that the sample point at $t = 3k$ have an energy higher than that of the point at $t = 0$, as shown in Fig. 5. The first sample point has already had its energy evaluated during the determination of the force vector. The choice of the other two sample points guarantees that the bowl of the parabola will be bracketed, and the program will not fall into one of the two pathological cases illustrated in Figs. 3, 4. The choice of the correct k can be made by a method of successive approximations. The initial guess of the correct value for k is based on the value of t at the

minimum of the previous point. Nearby points are similar enough that this guess is suitable about 90% of the time. If it is too large or too small, a suitable value can be found by a method of successive approximations.

4.2 Relaxation Order

The order of relaxation of points within a single sweep of all the points makes a difference to the behavior of the algorithm.

The most obvious method is to relax the points in each row in order, and then move on to the next row. This is commonly known as row-major order. This is shown for a five by five cloth on the left in in Fig. 6. This will produce asymmetries in the approach to a solution, and often produce an asymmetrical final solution. The problem is caused by asymmetrical propagation of influence. The relaxation of a single point looks only at the position of its eight neighbors. Thus it might seem that the position of point 13 at the beginning of a sweep would affect only the positions of the points in the dotted box. But if we relax in row-major order, the position of point 13 influences point 14, which in turn influences point 15. Since each point is moved before its neighbor is relaxed, the influence of point 13 can

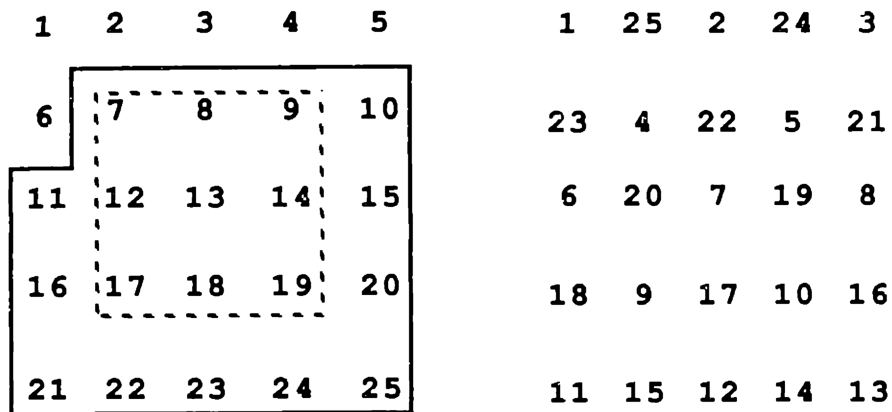


Figure 6: Row-major and red-black relaxation orders

spread to the entire area outlined in Fig. 6. Since this area is asymmetrical, the behavior of the cloth as it relaxes will be asymmetrical.

The correct solution to this problem is to move every point at once, in parallel. This requires maintaining two copies of the state of the cloth, one of the state at the beginning of the relaxation sweep, and one of the new positions generated in the current sweep. When relaxing a single point, the old positions of its neighbors are used in the determination of its new position, but the position is not updated until all the points have been relaxed. Unfortunately, this approach did not occur to me in time to include in the current implementation.

The method that was actually used, which is imperfect but better than row-major order, is "red-black" order [2, p. 243]. In this method, points

are relaxed in an order like that shown on the right in Fig. 6. The points are divided into two classes depending on the parity of the coordinates determining their positions in the grid, analogous to dividing a checkerboard into red and black squares. First, the "red" points are relaxed in row-major order, and then the "black" points are relaxed in reverse row-major order. This still results in asymmetrical propagation of influence, but the effect is much reduced.

5 The Energy Expression

The choice of energy expression is the prime determiner of the final shape of the cloth. The choice of the energy expression is a compromise between realism, speed, scale-independence and ease of debugging. Realism requires that the energy expression model the energy of real cloth as closely as possible. The energy expression must be evaluated at least six times every time a point is moved, and the program spends at least half its time evaluating energy, even with the simplest possible energy expression. The energy expression described in this section takes up about 90% of the execution time. The multigrid method requires that states of minimum energy represented

Symbol	Effect on behavior
k_s	Stretching resistance
k_b	Bending resistance
k_g	Density
h	Grid spacing
b	Buckling strength
r	Diagonal to axial strength ratio

Figure 7: Parameters of the energy expression

by different grid scales have more or less the same shape. If the shape of minimum energy is widely different for two different sizes of grid, the trick of approximating the solution to a fine grid by the solution to a coarse grid will break down. Simpler energy expressions are preferred on grounds of both speed and easy debugging, but excessively simple expressions will not be realistic.

The energy expression contains six parameters whose values determine the properties of the cloth. These are roughly described in Fig. 7 The typical values of the parameters were derived not from measurements of cloth but from tweaking until the results looked plausible. The grid spacing h changes as the algorithm proceeds. All other parameters are fixed by the user.

I will show that the discretization of the energy expression is a good

approximation to the true energy expression. First I will discuss the energy expression as a whole, and then describe the form of each of its subparts and show that they are good approximations to the continuous case.

5.1 The Total Energy

Continuous cloth is described parametrically by three functions mapping a position in the cloth coordinate system into the world coordinate system. $x(u,v)$, $y(u,v)$ and $z(u,v)$ give the coordinates in world space of the point at u,v in the cloth. The discrete model of this cloth uses the same three functions, restricted to act only on pairs of u,v coordinates which are multiples of h, the grid spacing. u and v range from 0 to u_{max} and v_{max} respectively.

The total energy of the cloth is a linear combination of three energy functions.

$$E_{total}(S) = k_s s(S) + k_b b(S) - k_g g(S) \tag{1}$$

The functions $s()$, $b()$, and $g()$ map shapes of cloth into numbers describing the extent to which the cloth is strained, bent and pulled down. The three parameters k_s,k_b and k_g control the relative strength of the three effects.

The definition of the three functions depends on a few additional, more subtle parameters, but the gross behavior of the total energy function is determined by the three parameters k_g, k_b and k_s. If k_s is large, the cloth will be very difficult to stretch. If k_b is large, the cloth will be stiff and resistant to bending. If k_g is large, the cloth will be heavy. Note that only the relative values of the three parameters matter. If all three were increased by a constant factor, the energy of the cloth would be increased proportionately but its behavior would be unchanged.

5.2 Gravity

The term $k_g g(S)$ in equation 1 corresponds to gravitational potential energy. In real cloth, the gravitational potential energy is given by

$$E_g = \rho g \int_0^{v_{max}} \int_0^{u_{max}} z(u,v) du dv$$

Where g is the acceleration of gravity and ρ the mass per unit area ("surface density") of the cloth. In the discrete case, the best approximation to this is

$$k_g g(S) = \rho g h^2 \sum_{v=0}^{v_{max}} \sum_{u=0}^{u_{max}} z(u,v)$$

Each point in the discretized cloth "stands in" for a square of cloth h on a side, and hence has a mass of ρh^2. Since k_g should be independent of h.

$$k_g = \rho g g(S) = h^2 \sum_{v=0}^{v_{max}} \sum_{u=0}^{u_{max}} z(u,v)$$

5.3 Bending

Before proceeding to the analysis of bending energy, there are some mathematical preliminaries.

Every point on a curve in the plane has a *curvature*. The curvature at a point P can be evaluated by rotating the curve until it is horizontal at P. The second derivative of the rotated curve, considered as a graph of a function, is the curvature. Consider a surface embedded in 3-space. Let the vector $\vec{n}$ be the normal to the suface at the point P Every plane lying on $\vec{n}$ intersects the surface, forming a curve. The curvature of this curve at P depends on the choice of section plane. The curvature varies smoothly as the plane is rotated. For some position of the plane, the curvature of the section is maximized, and for another position, the curvature is minimized. The maximum and minimum curvatures are referred to as the *principal curvatures* at the point. At every point on the surface, there are two prin-

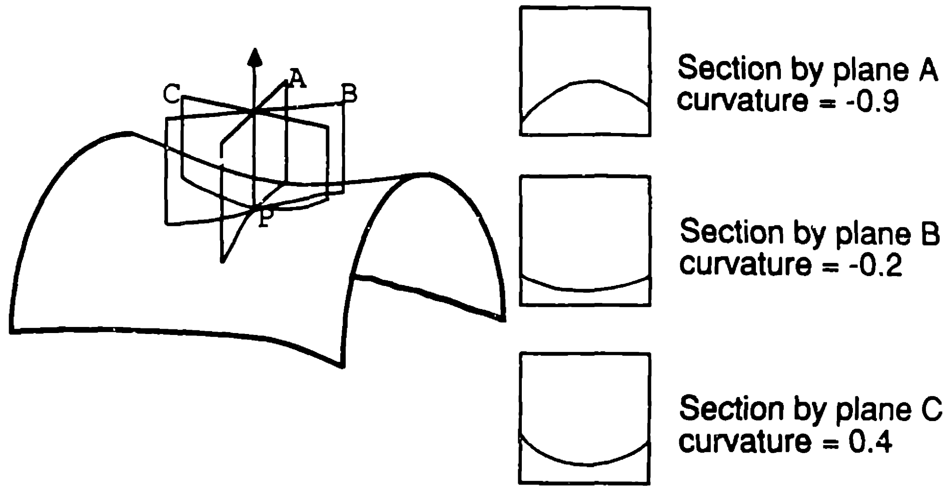


Figure 8: Evaluating the curvature at a point

cipal curvatures, κ_1 and κ_2. In the case illustrated in Fig. 8, the section by plane A produces the minimum curvature and the section by plane C produces the maximum curvature. Plane B is a section which produces an intermediate curvature. The product of the principal curvatures is referred to as the *gaussian curvature* [6]. A surface capable only of bending, but not of stretching (which changes the length of curves embedded in the surface) cannot change its gaussian curvature. An example of this is a sheet of paper. In its flat state, a sheet of paper has gaussian curvature which is everywhere 0, as illustrated in Fig. 9. Paper is not very stretchable,

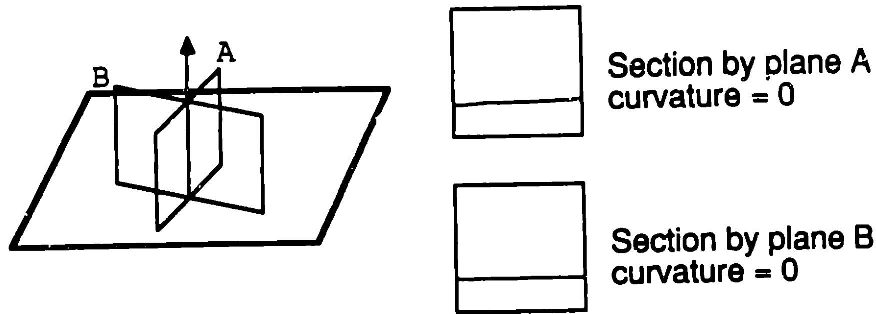


Figure 9: A flat piece of paper

and hence cannot change its gaussian curvature. No matter how it is bent, $\kappa_1\kappa_2 = 0$, and hence $\kappa_1 = 0$ or $\kappa_2 = 0$, as shown in Fig. 10. A surface without gaussian curvature is known as a developable surface.

The term $k_b b(S)$ in equation 1 corresponds to the energy of bending. The energy of bending given by the classical theory of elastic plates is

$$E_b(S) = \int_0^{v_{max}} \int_0^{u_{max}} c_1(\kappa_1 - \kappa_2)^2 + c_2(\kappa_1\kappa_2)^2 du dv$$

[7, p. 58] where c_1 and c_2 are constants describing the properties of the cloth. A relaxed cloth has no significant changes in length within its surface, hence there cannot be large areas with significantly nonzero gaussian

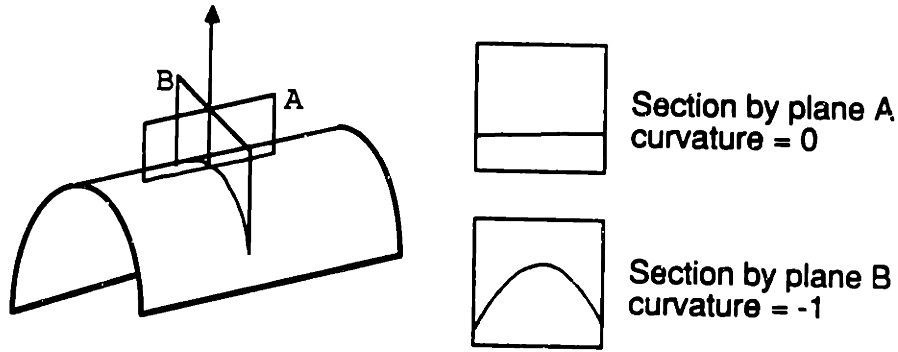


Figure 10: A bent piece of paper

curvature. Since $\kappa_1\kappa_2$ is small almost everywhere, c_2 may be ignored. In order for $\kappa_1\kappa_2$ to be close to zero, either κ_1 or κ_2 must be close to zero. If we denote by κ the nonzero principal curvature,

$$E_b(S) = \int_0^{v_{max}} \int_0^{u_{max}} c_1\kappa^2 du dv$$

I will describe the discrete approximation to bending and show it to be a discretization of the continuous case. The discrete approximation is based on the angles formed between lines connecting collinear points. The energy of bending of a single point depends on the four angles formed by the lines shown in Fig. 11. The energy at this point is

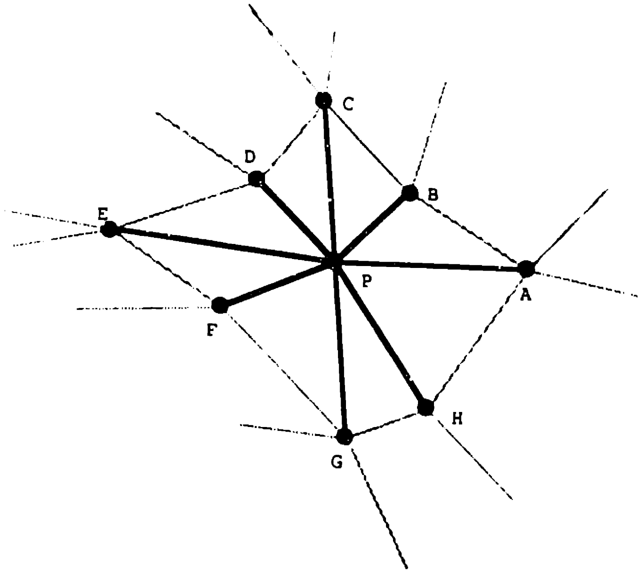


Figure 11:

$$E_b(p) = \frac{c_1}{3}((180^\circ - \angle APE)^2 + \frac{1}{2}(180^\circ - \angle BPF)^2 + (180^\circ - \angle CPG)^2 + \frac{1}{2}(180^\circ - \angle DPH)^2)$$

This is the energy that would be found if we were to attach angular springs to each angle formed by opposing lines coming into the point. Clearly the energy is zero when the cloth is flat.

I will show the equivalence of the discrete and continuouss energy measures by showing them to be equal at a typical point on a typical developable surface. This surface is typical in the sense that almost every point on a developable surface can be transformed into it by a diffeomorphism of the 3-space in which it is embedded. It is safe to consider only developable surfaces, since textiles usually have gaussian curvatures of nearly 0. The surface $z = \frac{1}{2}\kappa x^2$ has principal curvatures 0 and κ at the origin. Suppose we approximate this surface by a grid one of whose axes is inclined at an

angle θ to the y-axis. This surface is described parametrically by

$$x = u\cos\theta + v\sin\theta$$

$$y = v\cos\theta - u\sin\theta$$

$$z = \frac{1}{2}\kappa(u\cos\theta + v\sin\theta)^2$$

The energy of the point at the origin is

$$\begin{aligned} E_{0,0} = \tfrac{c_1}{3}[\quad & (2\arctan(\tfrac{h\kappa}{2}\cos^2\theta))^2 \\ + \quad & 2(\arctan(\tfrac{h\kappa}{2}\cos^2(\theta + 45^\circ)))^2 \\ + \quad & (2\arctan(\tfrac{h\kappa}{2}\cos^2(\theta + 90^\circ)))^2 \\ + \quad & 2(\arctan(\tfrac{h\kappa}{2}\cos^2(\theta + 135^\circ)))^2] \end{aligned}$$

When bending angles are small, $\arctan(y) \approx y$, and the equation reduces to

$$E_{0,0} = c_1\kappa^2 h^2$$

This is the energy that would be found in a patch of continuous cloth of area h^2. Since each point of the discrete cloth models a square patch of the real cloth which is h on a side, the two models agree to a good approximation.

Certain effects have been neglected in this analysis, but these effects are either small or only occur when the cloth is in a very deformed state,

in which case the details of the force law are unimportant as long as the general behavior is in the right direction. Since bending energy always decreases as the edges straighten out, this will generally push the cloth towards a flatter, less crumpled shape.

Nonlinearities occur when the radius of curvature approaches the grid spacing. Among other things, the approximation $\arctan(y) \approx y$ breaks down. In such a case the discrete grid is unable to accurately represent the continuous cloth. This analysis assumes that the grid lines meet at right angles when projected into the $z = 0$ plane. This is a safe assumption as long as the cloth is not severely sheared. It is unlikely that cloth could be both sheared and bent, since the shear stress would tend to pull it flat, dominating the bending strength. Moreover, the deviation from equality of the discrete and continuous energies is fourth order in the shear deformation, meaning that it is of no importance for small shear.

5.4 Strain

The analysis of the strain energy term is more complicated than the other two. In addition to the parameter k_s, the strain energy is also affected by

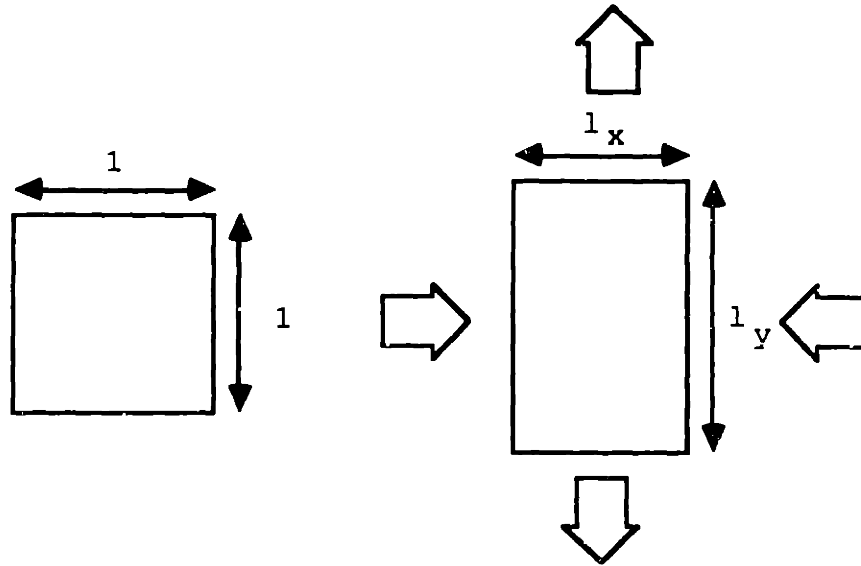


Figure 12: A stretched piece of cloth

the parameters r and b.

Consider a unit square of material subjected to changes in length in directions parallel to the x and y axes. This changes the length of its sides from 1 to l_x and l_y. The strains on this material are defined to be

$$u_{xx} = l_x - 1$$

$$u_{yy} = l_y - 1$$

as shown in Fig. 12. An ideal elastic material confined to a plane and stretched in such a fashion has an energy per unit area of

$$E_s = \frac{E}{1-\sigma^2}(u_{xx}^2 + u_{yy}^2) + \frac{2\sigma E}{1-\sigma^2} u_{xx} u_{yy}$$

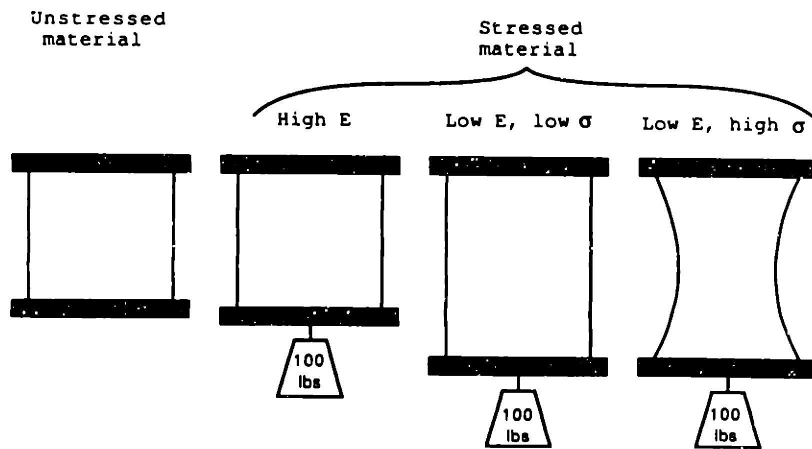


Figure 13: E and σ in action

[7, p. 53] where E and σ are physical parameters of the cloth, respectively the Young's modulus and the Poisson ratio. The Young's modulus determines how much the material stretches when subjected to a given force. The Poisson ratio determines how much the material contracts in one direction when extended in the perpendicular direction (Fig. 13). Any strain on a planar material can be resolved into a strain along perpendicular axes by rotating the coordinate frame correctly. We will consider in detail only the results of rotating the coordinates from the u, v coordinate system embedded in the cloth by 0° (placing the strain along warp and woof) and 45° (placing the strain along the bias.)

The value of E and σ for an anisotropic material such as woven or knitted cloth varies with direction. The E of woven cloth is usually much

higher along the thread than along the bias [5, p. 352] A factor of four variation is typical. This is why cloth is much easier to stretch when pulled diagonally than when pulled along the weave. On the other hand, the Poisson ratio is much higher along the bias than along the weave. [4, p. 251]. We will se below how these effects are approximated by the discrete energy expression.

The strain force law is modeled in the discrete approximation by nearly linear springs connecting points to their nearest eight neighbors. We will ignore the nonlinearity of the springs for the moment; it will be explained below. The springs connecting points to their nearest four neighbors are stronger than the springs connecting diagonally adjacent points, by the factor r. The value of r controls the degree to which the cloth is anisotropic. A small r produces a cloth which is weaker along the bias than along the threads. The energy of a spring is proportional to the square of the deviation of its length from the ideal length. Hence, the energy of the point P in Fig. 14 is

$$k_s[(h - PA)^2 + (h - PC)^2 + (h - PE)^2 + (h - PG)^2] +$$

$$k_s r[(\sqrt{2}h - PB) + (\sqrt{2}h - PB) + (\sqrt{2}h - PB) + (\sqrt{2}h - PB)]$$

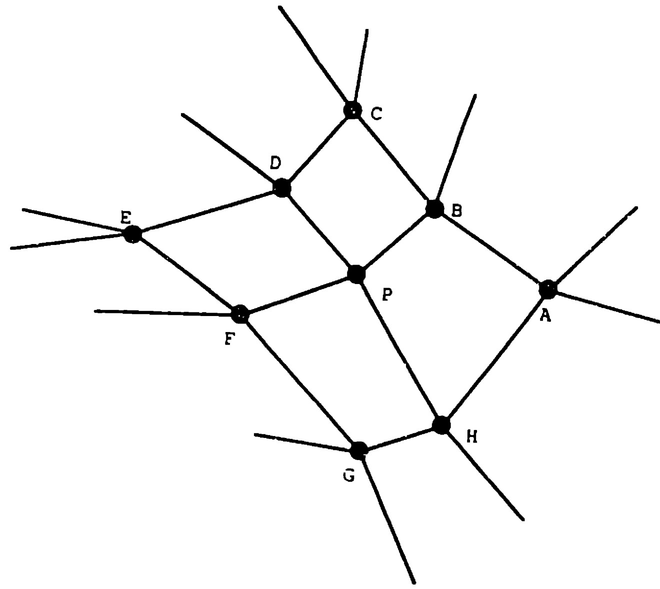


Figure 14:

where PA denotes the distance from point P to point A, and so on. What will the energy of a unit square of this discrete material be when subjected to the axial strains described in the last section? The points surrounding the point P will be moved from their positions to new positions, changing the lengths of the springs. The energy of a single point then becomes

$$(h^2 k_s + 3h^2 k_s sr)(u_{xx}^2 + u_{yy}^2) + 2h^2 k_s r u_{xx} u_{yy}$$

ignoring terms second order and higher in the strains. Since there are h^{-2} points in a unit square of the discrete cloth and the energy of a single spring is divided between two points, the energy of a unit square is equal to that of the continuous case with

$$E = k_s \frac{1 + 6r + 5r^2}{1 + 3r}$$

$$\sigma = \frac{2r}{1 - 3r}$$

Similarly, under strain along the bias, we find the discrete case to be equivalent to the continuous case with

$$E = k_s \frac{8r + 16r^2}{1 + 4r}$$

$$\sigma = \frac{1}{1 - 4r}$$

It is impossible to match the values of E and σ for all angles of a real cloth by altering only the parameters k_s and r, since this would require solving four equations in two unknowns. But we can obtain a good qualitative match for the properties of strongly anisotropic cloth by choosing a small value for r. Perfectly isotropic cloth cannot be modeled, but $r = 0.375$ gives identical Poisson ratios in the two directions, and Young's moduli which differ by only 10%.

The effect of variation in k_s is shown in Fig. 15. Each piece of cloth has twice the k_s of the piece to its left.

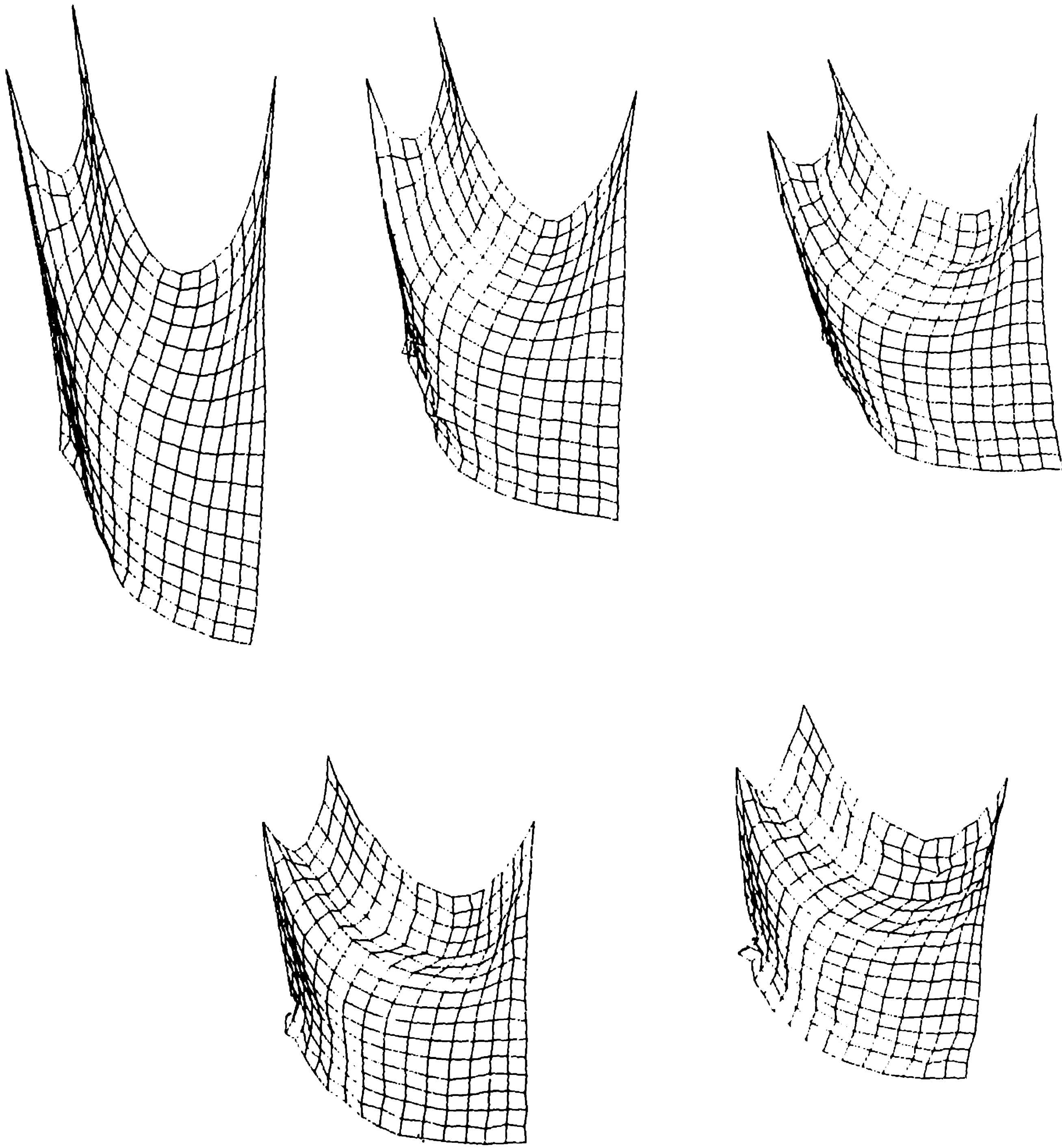


Figure 15: Cloths with different elasticities

5.5 Buckling

The above analysis is deficient because it assumes that the cloth is confined to a plane, which real cloth certainly is not.

If the springs were perfectly linear, the energy expression would fail to reproduce one of the real features of cloth: its weakness in compression. An ideal spring is just as strong in compression as in tension. This is emphatically not true of real cloth. Cloth is much weaker in compression than in tension because of buckling. When compressed, cloth grows smaller only slightly. Most of the energy of compression is taken up in buckling. The importance of buckling depends on the scale at which one views the cloth. In general, buckling is important only at scales close to or larger than the typical size of the folds in the material.

A plate subject to compression along one axis can adopt two possible configurations: it can remain in its original plane. and grow shorter, or it can bend out of its plane. and compress slightly. It will adopt whichever of these two configurations has lower energy. The potential energy of an ideal plate as a function of compression in one direction is shown qualitatively in Fig. 16 . The solid line shows the energy of the plate if it compresses and

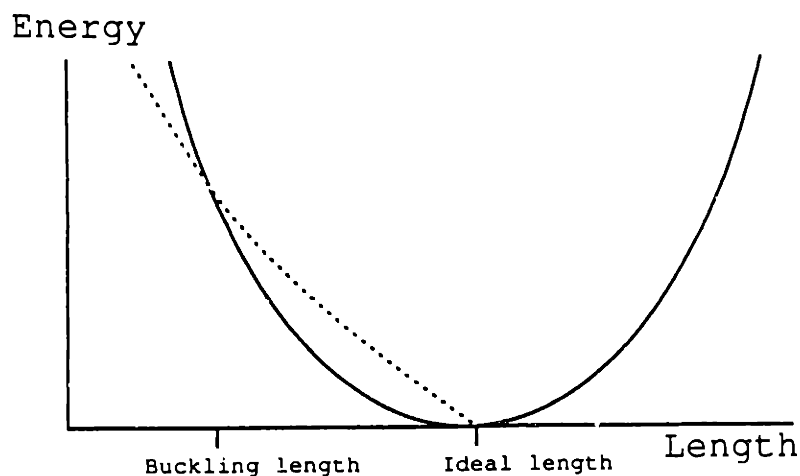


Figure 16: The energy of a plate as a function of compression.

the dotted line the energy if it bends. The actual energy of the plate will move along the lower parts of the two curves. As the plate is compressed, it will at first shorten in its own plane. and then suddenly buckle, considerably reducing the force required to hold it in position. If the plate is compressed past the point of buckling, its energy increases more slowly than it did while it was in compression.

The discrete cloth, as described above, does not reproduce the behavior of real cloth under compression. The springs which connect points in the discrete cloth should be weaker in compression than in tension. They should

duplicate the curve of energy versus length produced by a plate. The actual curve of energy versus length for a buckled plate is very difficult to calculate, involving elliptic integrals. A good approximation is to assume that energy is a linear function of compression. The constant of proportionality between energy and compression is determined by the parameter b.

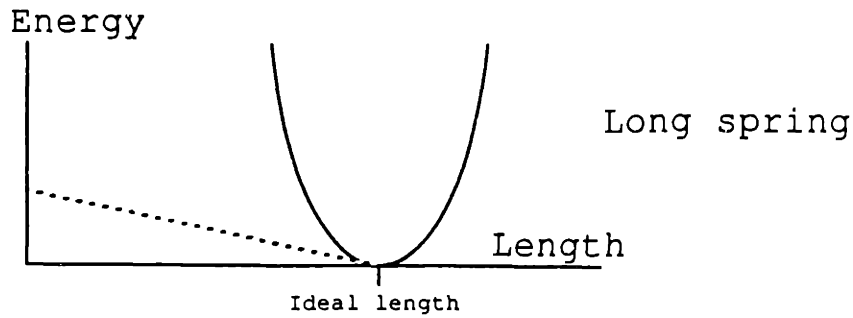
Each spring in the discrete approximation is replaced by a "buckling spring" whose energy as a function of length is given by

$$E(l) = k_s \min((l - l_i)^2, \frac{b}{l_i}(1 - l))$$

where l is the length of the spring and l_i is its ideal, uncompressed length. For diagonal springs, $l_i = \sqrt{2}h$. For warp and woof spings, $l_i = h$. Note that the relative importance of the buckling and elastic terms depends on the value of h, as shown in Fig. 17. For large h, the material has virtually no compressive strength. For small h, the material is incapable of buckling. This reproduces the behavior of real cloth.

5.6 Problems with the energy expression

The energy expresion should not be "stiff". A stiff energy expression is one in which the output is very sensitive to some of the input data and much



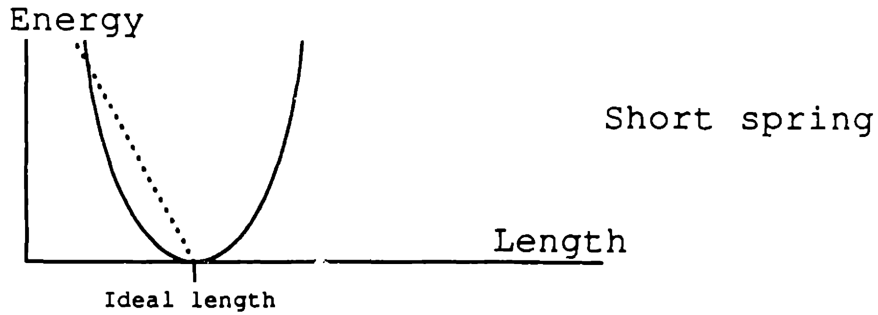


Figure 17: Energy as a function of length for long and short buckling springs.

less sensitive to other parts of the input data. (Note that stiffness of the energy expression is a mathematical property and should not be confused with stiffness of the cloth.) Applying a relaxation method to a stiff energy expression results in a disproportionately large amount of computational effort being put into optimizing the more important parts of the input and virtually no effort into optimizing the less important parts. This can defeat the whole purpose of the method if the appearance of the cloth is determined largely by parts of the input that the energy expression pays

less attention to.

For example, when k_s is large relative to k_b and k_g, the main determiner of the energy of a point is the distance between it and neighboring points. If k_s is set very high, the algorithm takes a very long time to converge, since it is spending most of its effort making sure that the distances between adjacent points are as close to h as possible. Fortunately, the problem of stiffness does not seem important in practice. The contribution of the various effects to the energy expression is well enough balanced that no effect soaks up a disproportionate share of the computation. This is achieved at the expense of perfect realism. The value of the Young's modulus used for the cloth in this thesis is unrealistically low, so the cloth is much stretchier than most real cloth, except for very loose knits.

The energy expression takes as input only the current shape of the cloth. It has no memory of how the cloth arrived at this position. It is thus incapable of modeling certain aspects of the behavior of real cloth which are known to have a large influence on its behavior, both empirically and theoretically [5, pp. 358,380]. Two such effects are hysteresis in shear and bending. Cloth tends to bend and shear more easily toward a shape

it has just been moved out of. Both of these effects have at their root the friction between the fibers composing the cloth. Friction holds the fibers in a relatively high energy state and prevents their relaxation. The representation of the cloth could be extended to include such hysteretic effects by including certain aspects of the past history of the cloth.

Unfortunately the representation of friction itself is beyond the approach described in this thesis. The energy minimization approach saves us from having to think about forces. Force appears only as the rate of change in energy with respect to position. It never explicitly enters into the calculations. This prevents us from taking into account forces not associated with changes in energy. Friction is one such force.

Friction prevents the approach to minimum energy in a complicated and highly nonlinear way. Consider a piece of cloth draped over a horizontal cylinder. If this cloth were to move toward minimum energy, it would slide off the cylinder, pulled down by its heavier side. Friction can prevent this, as long as the force of static friction of the cloth against the cylinder is greater than the force produced by the unbalanced weight of the cloth. Unfortunately, this force of static friction is not related to a change in

energy in any useful way. The energy used to fight friction goes into heat energy rather than potential energy, and cannot be taken into account.

6 The Multigrid Method

The version of the multigrid method described in this thesis is more properly called the *multigrid full approximation scheme.* It is a member of a more general family of multigrid methods. The other members of this family are suited only to linear problems. Since the problem of textile mechanics is vigorously nonlinear. we are forced to choose the full approximation scheme.

To see how such a method might work, we will examine a method that only uses two grids. We start with the cloth represented by the fine grid, in some arbitrary initial state. If we relaxed only this grid, it would take a long time to create coarser features. Instead, we relax the fine grid for only the number of steps sufficient to create plausible small-scale features. Then we coarsen the grid by retaining only every second point in each direction. This reduces the number of points fourfold. We then relax this coarse grid to a state of minimum energy. This grid is then refined into

grid of the same resolution as the original grid. The refinement combines the large-scale features of the coarse grid with the small-scale features of the original grid. This refined grid is then used as the initial state for a further process of relaxation, which eventually produces the final solution to the problem. This method works as long as the coarse-grid solution is a good approximation to the fine-grid solution. Because the slow process of producing large-scale features was done mostly on a grid half the size of the original grid, the whole algorithm was speeded up about fourfold.

The algorithm could be sped up even further by relaxing the coarse grid using the two-grid process rather than simple relaxation. By applying the two-grid method recursively, we derive the multigrid method, whereby each grid is given a good initial approximate solution by adopting larger features from a coarser grid. The recursion is grounded in the coarsest grid to show interesting features, which in most cases is only three points on a side.

6.1 The Schedule of Relaxation

The behavior of the multigrid full approximation scheme depends on a number of interacting design choices. The choice of refinement and coarsening

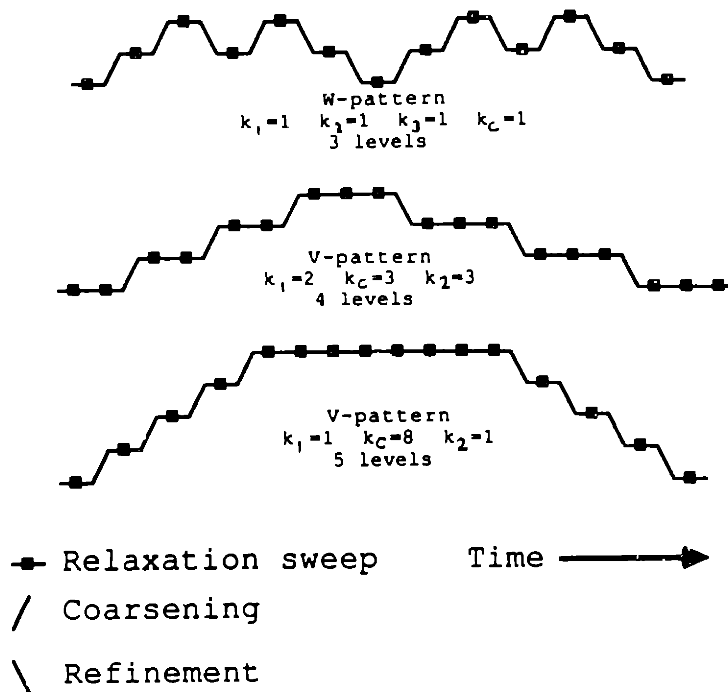


Figure 18: A variety of relaxation schedules

operators, the schedule of relaxation and the choice of grid scales all affect the behavior of the method. The schedule of relaxation is the sequence of operations of relaxation, coarsening and refinement. Several such schedules are shown in Fig. 18. These schedules are divided into two classes: V- and W-patterns. The V-patterns are generated recursively by the schedule

relax k_1 **times**
coarsen

```
if at coarsest scale
   then relax k_c times
   else recurse
refine
relax k_2 times
```

The W-patterns are generated by the schedule

```
relax k_1 times
coarsen
if at coarsest scale
   then relax k_c times
   else recurse
refine
relax k_2 times
coarsen
if at coarsest scale
   then relax k_c times
   else recurse
refine
relax k_3 times
```

The schedule is defined by specifying which of these patterns to use, the size of the coarsest scale, and the values k_c, k_1, k_2, and (if neccesary) k_3. Since one relaxation sweep of a cloth containing n points requires time $O(n)$, the V-pattern requires total time of order $n(k_1 - k_2)$, and the W-pattern requires time of order $n(k_1 + k_2 - k_3)$. This is immensely superior

to the simple relaxation method, which requires time of order n^2 to achieve comparable relaxation.

6.2 The Coarsening and Refinement Operators

The coarsening operator I_h^{2h} maps a cloth with grid spacing h into a cloth with grid spacing $2h$. The function chosen for coarsening is

$$\begin{aligned} P(u,v) = 1/8(\; & p(u-h, v-h) + 2p(u, v-h) + p(u+h, v-h) \\ & 2p(u-h, v) \quad + 4p(u,v) \quad + 2p(u+h, v) \\ & p(u-h, v+h) + 2p(u, v+h) + p(u+h, v+h)) \end{aligned}$$

where $P(u,v)$ is the location of the point at (u,v) in the coarse grid and $p()$ is the location of a point in the finer cloth. These are all 3-dimensional vectors. u and v are always multiples of $2h$. This operator, known as "full weighting" [10, p. 15] was chosen for a number of reasons. First, it is simple to compute. Second, every point in the fine grid has the same amount of influence on the coarse grid. Third, it does not alias high-frequency waves in the fine grid into low-frequency waves in the coarse grid. In this, it is much superior to the even simpler coarsening operators

$$P(u,v) = p(u,v)$$

and

$$P(u,v) = 1/4(p(u-h,v) + p(u-h,v-h) + p(u,v-h) + p(u,v))$$

which produce severe aliasing when applied to a fine grid whose features are of a size close to twice the grid spacing.

The refinement operator is somewhat more complex. The object is to use information from the relaxation of the coarse grid to move the shape of the fine grid closer to a state of minimum energy. The refinement operator finds the difference between the shapes of the coarse grid before and after relaxation. and applies this correction to the fine grid. The refinement operator thus takes two arguments: a fine grid and a coarse grid. It is defined as

$$\text{refine}(f,c) = f - I_{2h}^{h}(c - I_{h}^{2h}(f))$$

where f and c denote the fine and coarse grids respectively, the additions and subtractions are to be applied pointwise, and the operator I_{2h}^{h} is an interpolation operator which converts a coarse grid into a fine grid [2, p. 272]

The choice of interpolation operator has constraints similar to those of

the coarsening operator. The operator should be fast, evenhanded and not introduce spurious high frequency terms in the fine grid. The interpolation operator I have chosen is cubic-spline interpolation, which results in much smoother interpolation than, e.g., bilinear interpolation. [2, p. 251]

A simpler refinement operator would be to generate a new fine grid simply by interpolating the coarse grid, and use this as a basis for further relaxation. The operator would become

$$refine(f, c) = I_{2h}^{h}(c)$$

This has two drawbacks. First, it ignores the fine-scale detail generated by the fine grid relaxation before passing to a coarser scale, and forces the process to regenerate this detail. This is acceptable if the form of the small-scale detail is so dependent on the coarse features that the old details are as good as useless, but this is often not the case. Also, the aliasing produced by the interpolation operator is proportional to the signal interpolated. Since $c - I_h^{2h}(f)$ is almost always smaller than c, the error of interpolation will be smaller as well.

6.3 Problems with the Multigrid Method

The multigrid relaxation method has a potential danger not shared by the pure relaxation method: it may not move in the direction of reduced energy. It is possible that the coarse grid may relax in a direction which reduces its energy but which does not reduce the energy of the finer grids. In this case, the fine grid will have to undo the work of the coarse grid. Since the convergence of a fine grid is slower than the convergence of a coarse grid for low-frequency errors, the fine grid usually does not have time to correct the errors of the coarse grid, and the errors are frozen into the final solution. I have not implemented any general solution to this problem.

The multigrid method as applied in this program has a flaw which is clear in Fig. 19. The hanging cloth does not, as one might expect, form folds extending from the extension points at the top to the bottom hem. Instead, any given fold extends only about a fourth of the distance from the top to the bottom. These separate folds do not coalesce into longer folds because they do not have time. They can be represented only when the resolution of the grid is no more than half their width. At this grid spacing, they are each at least four grid points long. The relaxation method

is very slow at eliminating errors of this size, especially when, as in this case, they are almost correct everywhere. In order for relaxation to efficiently eliminate an error, the size of the error must be comparable to the grid spacing in both u and v directions [3, p. 7]. The speedy elimination of the error in Fig. 19 would require a grid which was not square, but rectangular, with the long axes of the rectangles directed parallel to the axes of the folds. The decision to use such a grid would have to made by the program when it saw that long, thin features were developing.

In the current program. all points in the cloth are relaxed the same number of times. This is an inefficient use of computation. since producing complicated features requires more sweeps of relaxation than producing simple ones. Portions of the cloth containing complicated folds or other features should be relaxed more times than portions which are flat.

A fundamental problem with the multigrid method as used in this thesis is that the user must decide on the relaxation schedule. In general, the more relaxation is performed. the better the result. Beyond a certain point, however, the effects of further relaxation are infinitesmal. Deciding how many relaxation sweeps to perform requires experience and extensive

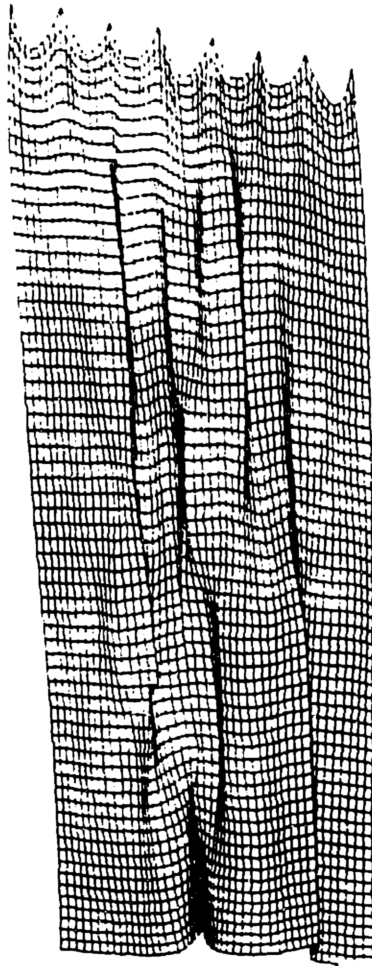


Figure 19: A square of cloth supported at eight points along one edge.

experimentation. The program should be able to decide for itself whether further relaxation is in order, on the basis of experience in past relaxations. It should also be able to decide when it would be profitable to coarsen the grid, in order to speed up the creation of large-scale features.

7 External constraints

The shape of real cloth is a compromise between the tendencies internal to the cloth and the influences of the world. In this thesis, the effects of the world are restricted to gravity, fixed points, and solids.

Gravity was discussed in the section on the energy expression. Other force fields could be taken care of in much the same way, provided that they are pure potential fields.

7.1 Fixed Points

Fixed points are points in the cloth that are constrained to have a fixed position. The only subtle aspect of fixed points is the question of what to do with them when the coarsening operator is applied. The approach used is to declare that a point in the coarse grid is fixed if any of the nine points

in the fine grid used in the calculation of the coarsening operator are fixed. However, the position of this new fixed point is the average of the positions of the nine points, just as with any other point in the coarse grid. This procedure guarantees that fixed points will not be moved by the process of coarsening and refinement.

7.2 Solids

Solids are regions of space into which the cloth is forbidden to move. If, during the relaxation process, a point is moved into a solid, it is instead moved to the nearest point outside a solid. This process of relocation is invisible to both the relaxation program, which sends sample points to the energy expression, and to the energy expression, which simply never sees sample points inside a solid. Unfortunately, this method requires that every time the energy of a point is evaluated, the program must check to see whether the position is inside a solid, and if so, find the nearest point outside this solid. Even for simple objects, this is a time-consuming operation, and it is in the inner loop of the program.

The only solids currently implemented are ellipsoids. Ellipsoids are

algebraically tractable and sufficient to construct a wide variety of objects, including as special cases disks, infinite planes, spheres and cylinders. The determination of collisions between points and ellipsoids requires only a few multiplications. The inclusion of a fast check to see if the point was within the bounding box of the ellipsoid might speed things up, but because cloth is often very close to the objects it drapes, a large fraction of points would be inside the bounding boxes.

Unfortunately, the experimental performance of this method of including solids in the program has been disappointing. When the grid is at its coarsest, the points are so far apart that objects simply slip between them. When the cloth is refined, it is distorted by the presence of the object in its midst. This can be seen in Fig. 20. The cloth lying near the sphere is unnaturally stretched. Some better approach is needed.

The cloth has no restrictions against self-intersection. This occaisonaly produces odd effects, as can be seen in the left side of the cloth in Fig. 21, and in the crumpled cloth at the bottom of 22. I decided against the inclusion of tests for self-intersection in the program since they would be difficult to program and computationally expensive.

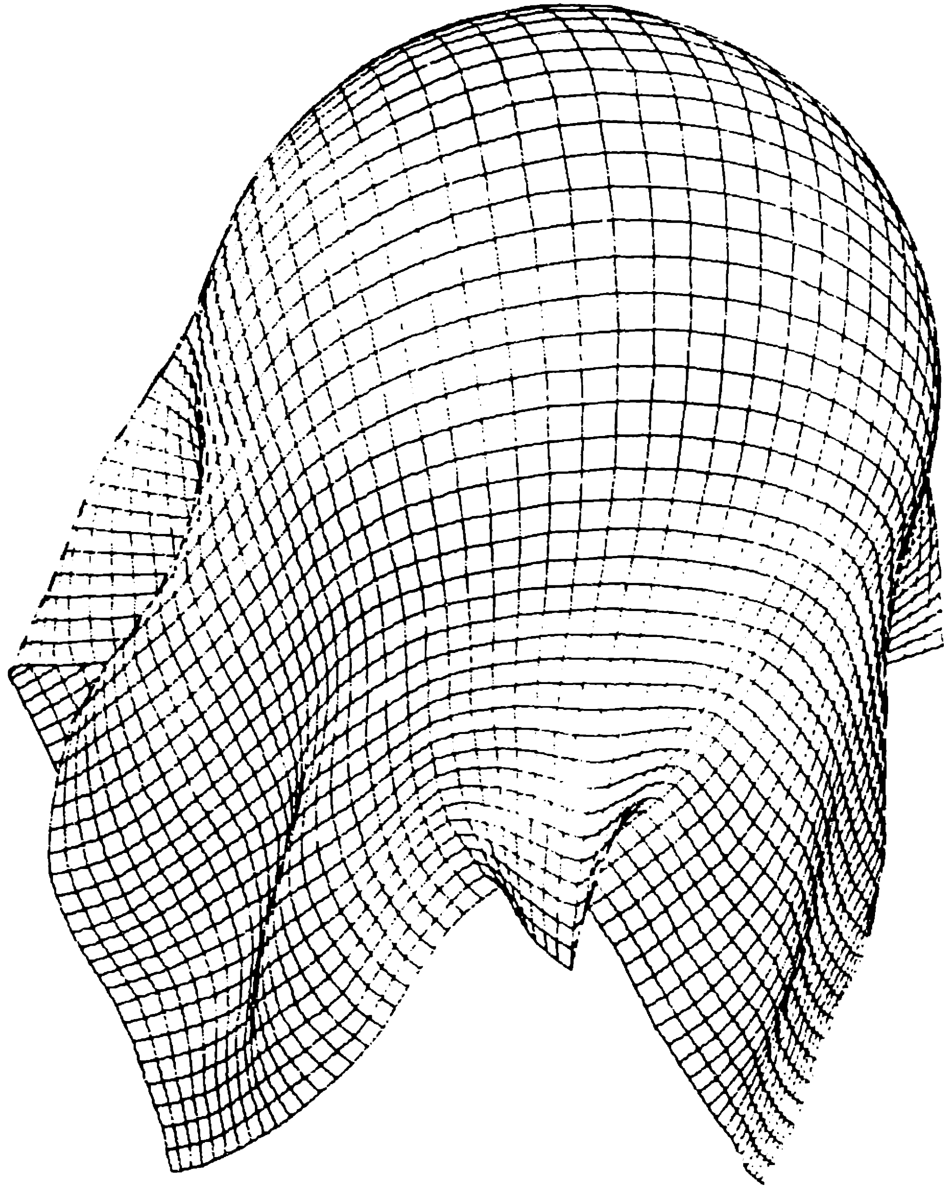


Figure 20: Cloth draped over a sphere

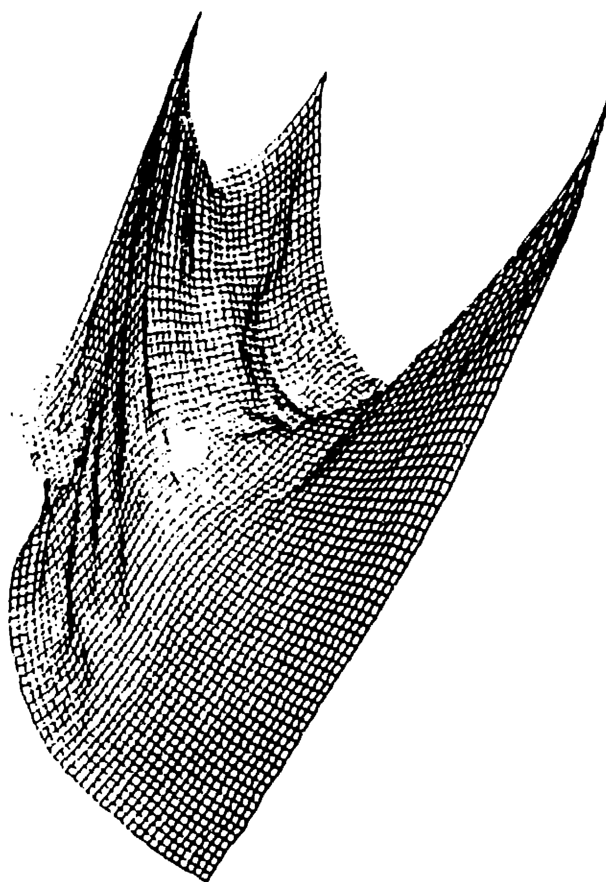


Figure 21: A piece of cloth supported at three points.

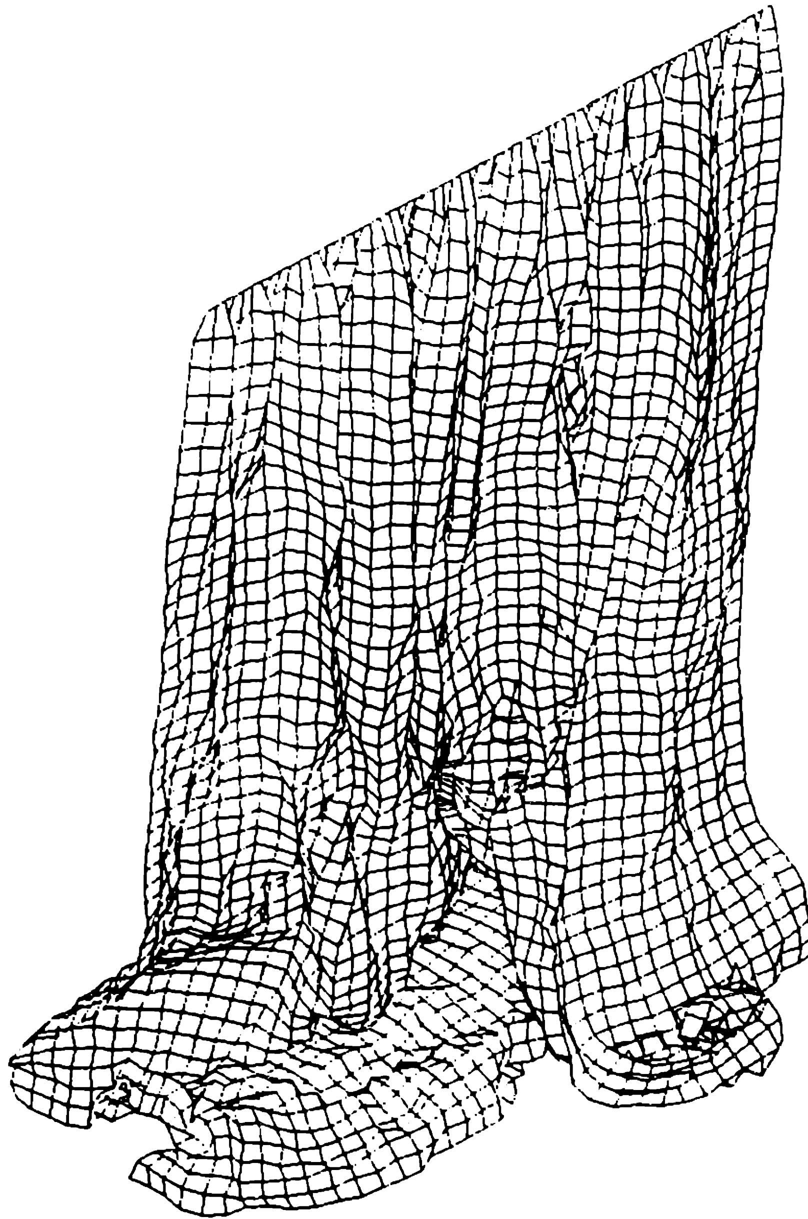


Figure 22: A square piece of cloth compressed along its upper edge and crumpled on the floor below.

8 Directions for the Future

In addtion to the improvements discussed above in sections 5.6 and 6.3, there are a number of changes neccesary to turn this program into a generally useful tool.

Clothing made out of several pieces of cloth could be simulated by constructing the garment out of ovelapping patches of flat cloth. The patches would be constrained to move identically in places where they overlapped. Each step in the multigrid schedule would be applied to all the patches in parallel. This is a standard technique for numerical solution of problems on two-dimensional manifolds.

If models of cloth could be made realistic enough to fool not just uncritical observers but discerning textile experts, one could imagine the creation of computer-assisted fashion design programs. Prototyping a garment could be done on the computer screen, instead of on a live model. The market for such a service would not be prototyping for mass production, since in such cases the construction of an actual prototype is negligible compared with the production cost. Rather, the prime application would be to the creation of unique custom garments, which could be previewed by the cus-

tomer before purchase.

One obvious extension of the program is to animation. This will not be easy. The approach of searching for stationary states of minimum energy does not apply, since stationary states are exactly what we don't want. The program will have to integrate the equations of motion, using the energy expression to calculate forces, the forces to calculate velocities, and the velocities to calculate positions. Something more efficient than simply running the algorithm on every frame will be needed. The coherence between succesive frames of an animation should be exploited to save computation. The application of multigrid algorithms to time-dependent problems is an area of current research and not as straightforward as their application to time-free problems [3].

The use of multigrid methods to simulate the mechanics of continuous objects has promise for a wide variety of problems in computer graphics. Phenomena such as billowing smoke, turbulent water, and rippling muscles have not yet appeared in computer graphics in part because they were considered too computationally expensive to simulate. The enormous increases in computational efficiency given us by multigrid methods bring

these phenomena into the domain of feasibility.

9 Acknowledgments

I would like to thank a number of people and institutions for helping me with this thesis. Resources were made available by David Zeltzer, Charles Sommer, Jim Salem, Regan Perry, the M.I.T. Media Laboratory, the M.I.T. Artificial Intelligence Laboratory and Thinking Machines Corporation. This work was supported in part by an equipment loan from Symbolics Inc., of Cambridge, Massachusetts. I profited by discussing my work with David Chen, Steve Estephanian. Charles Lieserson, Margaret Minsky, Steve Omohundro, Jim Salem, Karl Sims, Tom Trobaugh and David Zeltzer. My roommates were rendered inaudible by Robert Fripp and the members of Einstürzende Neubauten. Pink Floyd. Shreikback and Rhythm&Noise. I would like to thank my father, Richard Feynman, who convinced me to work on modeling cloth rather than baked goods, on the basis that "with cloth, you can tell when you failed", as well as providing me clearheaded advice throughout the research.

"3D Graphics Toolkit" is a trademark of Thinking Machines Corpora-

tion.

"Symbolics 3600 Computer" is a trademark of Symbolics, Inc.

References

[1] Booth, J. E. *Principles of Textile Testing* (New York: Chemical Publishing Company, 1964)

[2] Brandt, Achi, "Guide to Multigrid Development" In *Multigrid Methods*, Lecture Notes in Mathematics, No. 960. (New York: Springer-Verlag, 1982)

[3] Brandt, Achi, "Levels and Scales" In *Multigrid Methods for Integral and Differential Equations* Ed. by D. J. Paddon and H. Holstien. (Oxford: Oxford University Press, 1985)

[4] Gordon, J. *Structures, or why things don't fall down* (New York: Da Capo Press, 1982)

[5] Hearle, J., P. Grosberg and S. Backer. *Structural Mechanics of Fibers, Yarns, and Fabrics* (New York: Wiley-Interscience. 1969), Vol. I

[6] Hilbert, D. and S. Cohn-Vossen. *Geometry and the Imagination* (New York: Chelsea, 1952)

[7] Landau, L. D. and E. M. Lifshitz it Theory of Elasticity, Course of Theoretical Physics, Vol. 7 (Oxford: Pergamon, 1970)

[8] Otto, F. *Tension Structures* (Cambridge, Massachusetts: MIT Press, 1967)

[9] Steele, Guy L. *Common LISP: The Language* (Burlington, Massachusetts: Digital Press, 1984)

[10] Steüben, K. and Trottenberg, U., "Multigrid methods: fundamental algorithms, model problem analysis and applications" In *Multigrid Methods*, Lecture Notes in Mathematics, No. 960. (New York: Springer-Verlag, 1982)

[11] Weil, Jerry, "The Synthesis of Cloth Objects", to appear in *Computer Graphics*, August 1986.

A The Program

This appendix contains selected portions of the code. Not included is the code to organize file loading and compilation, the code to read and save pieces of cloth to or from the file system, or the code to interface to rendering systems. This program calls certain routines for the creation and manipulation of vectors and transformation matrices. These functions were originally part of the TMC 3D Graphics Toolkit. I took the file and translated it to Common LISP without changing the functionality. This code is not included, but the purpose of the functions should be obvious from their names.

The code consists of five files, meant to be compiled in the order they are given here. The code is a mess. *Caveat lector.*

File BASICS:

```
;;; -*- Mode: Lisp; Package: COMMON-LISP-USER; Syntax: Common-
lisp -*-

;;; Various utility functions

(defun 2d-distance (x1 y1 x2 y2)
  (sqrt (+ (expt (- x1 x2) 2)
```

```
   (expt (- y1 y2) 2))))

(defun 3d-distance (x1 y1 z1 x2 y2 z2)
  (sqrt (+ (expt (- x1 x2) 2)
   (expt (- y1 y2) 2)
   (expt (- z1 z2) 2))))

(defun vector-max-component (vector)
  (max (abs (vector-x vector))
       (abs (vector-y vector))
       (abs (vector-z vector))))

(defun multiply-vector-by-matrix (vector matrix)
  (math:multiply-matrices matrix vector))

(defun map-2d-array (function &rest arrays)
  (let* ((dimension-0 (apply #'min (map 'list #'(lambda (array)(array-
dimension array 0)) arrays)))
 (dimension-1 (apply #'min (map 'list #'(lambda (array)(array-
dimension array 1)) arrays)))
 (result (make-array (list dimension-0 dimension-1))))
    (dotimes (x dimension-0)
      (dotimes (y dimension-1)
(setf (aref result x y)
      (apply function
     (map 'list #'(lambda (array)(aref array x y))
  arrays)))))
    result))

(defun map-2d-array-for-side-effect (function &rest arrays)
  (let* ((dimension-0 (apply #'min (map 'list #'(lambda (array)(array-
dimension array 0)) arrays)))
 (dimension-1 (apply #'min (map 'list #'(lambda (array)(array-
dimension array 1)) arrays))))
    (dotimes (x dimension-0)
      (dotimes (y dimension-1)
```

```
(apply function
       (map 'list #'(lambda (array)(aref array x y))
    arrays))))))

(defmacro do-in-range ((variable start end steps) &body body)
  (let ((var (gensym)))
  `(dotimes (,var ,steps)
     (let ((,variable (rescale ,var 0 ,steps ,start ,end)))
       ,@body))))

;Cloth is a rectangular patch of an infinite square lattice. The
points
;(the geometrical object) of the lattice are points (the defstruct).
;The limits are inclusive.  These limits may change if cloth ever
;becomes nonrectangular.
(defstruct (cloth :conc-name
  (:copier copy-cloth-internal))
  points
  grid-spacing
  ;; The following slots are no longer used.  They will be elim-
inated in the future.
  max-u min-u
  max-v min-v)

;Out-of-bounds requests return nil.
(defun point-at (cloth u v)
  (point-at-integer cloth
    (round (/ u (cloth-grid-spacing cloth)))
    (round (/ v (cloth-grid-spacing cloth)))))

;Out-of-bounds requests return nil.
(defun point-at-integer (cloth u v)
  (if (or (< u 0)
  (< v 0)
  (>= u (u-array-limit cloth))
  (>= v (v-array-limit cloth)))
```

```
      nil
      (aref (cloth-points cloth) u v)))

(defun u-array-limit (cloth)
  (array-dimension (cloth-points cloth) 0))

(defun v-array-limit (cloth)
  (array-dimension (cloth-points cloth) 1))

(defun extract-array-from-cloth (function cloth)
  (map-2d-array function (cloth-points cloth)))

;;; Points are things that have a fixity, which can be :fixed,
:free, or
;;; :dummy.  These mean the following things.  :fixed can't be
moved.
;;; :free are movable.  :dummy are actually not in the cloth at
all,
;;; they're just there to make the point array rectangular.
(defstruct (point :conc-name
  (:copier copy-point-internal)
  (:print-function (lambda (point stream depth)
     (ignore depth)
     (if (point-location point)
 (format stream "#<Point ~,4f ~,4f ~,4f ~d ~d ~o>"
     (point-x point)
     (point-y point)
     (point-z point)
     (point-u point)
     (point-v point)
     (si:%pointer (point-cloth point)))
 (format stream "#<Point (no location) ~d ~d ~o>"
     (point-u point)
     (point-v point)
     (si:%pointer (point-cloth point)))))))
  location
```

```
  u v cloth ;specifies where the point is in the cloth.
  (fixity :free)
  ;; The following slots are initialized to :empty but serve to
cache nieghboring points.
  (cache-u+ :empty)
  (cache-u- :empty)
  (cache-v+ :empty)
  (cache-v- :empty))


(defparameter *all-point-fixity-types* '(:free :fixed :dummy))

(defun point-grid-spacing (point)
  (cloth-grid-spacing (point-cloth point)))

(defun create-point-xyz (x y z u v cloth fixity)
  (make-point :location (make-vector x y z) :u u :v v :cloth cloth
:fixity fixity))

(defun create-point (location u v cloth fixity)
  (make-point :location location :u u :v v :cloth
      cloth :fixity fixity))

;This only works to copy a point to the same u,v on another piece
of cloth.
(defun copy-point (point &optional cloth)
  (let ((result (copy-point-internal point)))
    (setf (point-location result)
  (copy-vector (point-location point)))
    (setf (point-cache-u+ result) :empty
  (point-cache-u- result) :empty
  (point-cache-v+ result) :empty
  (point-cache-v- result) :empty)
    (if cloth
 (setf (point-cloth result) cloth))
    result))
```

```
(defparameter *dummy-point* (make-point :fixity :dummy))
(defun dummy-point ()
  *dummy-point*)

(defun point-x (point)
  (vector-x (point-location point)))

(defun point-y (point)
  (vector-y (point-location point)))

(defun point-z (point)
  (vector-z (point-location point)))

;This is only function allowed to change point location after creation.
(defun set-point-location (point location)
  (setf (point-location point) location))

(defun offset-point (point offset)
  (set-point-location point
      (add-vectors (point-location point)
   offset)))

;These functions find the nieghboring points, given a point.  If
there is no point in that direction, it returns nil.
(defun point-to-u+ (point)
  (if (eq :empty (point-cache-u+ point))
      (setf (point-cache-u+ point)
    (point-at-integer (point-cloth point) (+ (point-u point) 1)
(point-v point)))
      (point-cache-u+ point)))

(defun point-to-u- (point)
  (if (eq :empty (point-cache-u- point))
      (setf (point-cache-u- point)
    (point-at-integer (point-cloth point) (- (point-u point) 1)
```

```
(point-v point)))
      (point-cache-u- point)))

(defun point-to-v+ (point)
  (if (eq :empty (point-cache-v+ point))
      (setf (point-cache-v+ point)
    (point-at-integer (point-cloth point) (point-u point) (+ (point-
v point) 1)))
      (point-cache-v+ point)))

(defun point-to-v- (point)
  (if (eq :empty (point-cache-v- point))
      (setf (point-cache-v- point)
    (point-at-integer (point-cloth point) (point-u point) (- (point-
v point) 1)))
      (point-cache-v- point)))

(defun point-to-u-- (point)
  (let ((p (point-to-u- point)))
    (and p
 (point-to-u- p))))

(defun point-to-u++ (point)
  (let ((p (point-to-u+ point)))
    (and p
 (point-to-u+ p))))

(defun point-to-v-- (point)
  (let ((p (point-to-v- point)))
    (and p
 (point-to-v- p))))

(defun point-to-v++ (point)
  (let ((p (point-to-v+ point)))
    (and p
 (point-to-v+ p))))
```

```
(defmacro do-points ((point cloth &optional (types-to-include '(:free)))
&body body)
  `(let (($u-limit (array-dimension (cloth-points ,cloth) 0))
 ($v-limit (array-dimension (cloth-points ,cloth) 1)))
     (dotimes ($u $u-limit)
       (dotimes ($v $v-limit)
 (let ((,point (point-at-integer ,cloth $u $v)))
   (if (memq (point-fixity ,point) ',types-to-include)
       (progn ,@body)))))))

(defmacro do-point-coordinates ((u v cloth &optional (types-to-
include ''(:free))) &body body)
  `(let (($u-limit (array-dimension (cloth-points ,cloth) 0))
 ($v-limit (array-dimension (cloth-points ,cloth) 1)))
     (dotimes (,u $u-limit)
       (dotimes (,v $v-limit)
 (if (memq (point-fixity (point-at-integer ,cloth ,u ,v)) ,types-
to-include)
     (progn ,@body))))))

;Like doing a checkerboard, first the red squares then the black
squares.  (0,0) gets done first.
(defmacro do-points-in-red-black-order ((point cloth &optional
(types-to-include '(:free))) &body body)
  `(let (($u-limit (array-dimension (cloth-points ,cloth) 0))
 ($v-limit (array-dimension (cloth-points ,cloth) 1)))
     (dotimes ($u $u-limit)
       (dotimes ($v $v-limit)
 (if (evenp (+ $u $v))
     (let ((,point (point-at-integer ,cloth $u $v)))
       (if (memq (point-fixity ,point) ',types-to-include)
   (progn ,@body))))))
     (dotimes ($u-backwards $u-limit)
       (dotimes ($v-backwards $v-limit)
 (let (($u (- $u-limit 1 $u-backwards))
```

```
        ($v (- $v-limit 1 $v-backwards)))
    (if (oddp (+ $u $v))
        (let ((,point (point-at-integer ,cloth $u $v)))
  (if (memq (point-fixity ,point) ',types-to-include)
      (progn ,@body)))))))))

(defun copy-cloth (cloth)
  (let ((result (copy-cloth-internal cloth)))
    (setf (cloth-points result)
  (make-array (array-dimensions (cloth-points cloth))))
    (do-point-coordinates (u v cloth *all-point-fixity-types*)
      (setf (aref (cloth-points result) u v)
    (copy-point (aref (cloth-points cloth) u v) result)))
    result))

(defun point-to-u+v+ (point)
  (let ((u+ (point-to-u+ point)))
    (if u+
(point-to-v+ u+)
nil)))

(defun point-to-u+v- (point)
  (let ((u+ (point-to-u+ point)))
    (if u+
(point-to-v- u+)
nil)))

(defun point-to-u-v+ (point)
  (let ((u- (point-to-u- point)))
    (if u-
(point-to-v+ u-)
nil)))

(defun point-to-u-v- (point)
  (let ((u- (point-to-u- point)))
    (if u-
```

```
(point-to-v- u-)
nil)))
```

File ENERGY:

```
;;; -*- Mode: Lisp; Package: COMMON-LISP-USER; Syntax: Common-
lisp -*-

(defparameter *density* 1)
(defparameter *elasticity* 5)
(defparameter *rigidity* 0.002)
(defparameter *gravity* :call-set-force-parameters-you-dummy)
(defparameter *buckling-energy-linear-term* :call-set-force-parameters-
you-dummy)
(defparameter *diagonal-force-ratio* :call-set-force-parameters-
you-dummy)
(defparameter *current-grid-spacing* :call-set-force-parameters-
you-dummy)

;Buckling energy is
;divided by elastic force constant because we want to save mul-
tiplications by
;multiplying by elastic force constant as late as possible.
(defun set-force-parameters (length ;l in notebook
      &optional (elasticity *elasticity*)
      (rigidity *rigidity*) ;Bs in notebook
      ;; These next ones you probably don't ever want to change.
      (density *density*)
      (diagonal-force-ratio .25))
    (setq *buckling-energy-linear-term* (/ rigidity length elasticity))
    (setq *diagonal-force-ratio* diagonal-force-ratio)
```

```
    (setq *gravity* (* density length length 9.8))
    (setq *density* density)
    (setq *elasticity* elasticity)
    (setq *rigidity* rigidity)
    (setq *current-grid-spacing* length))

(defmacro with-grid-spacing (spacing &body body)
  `(let ((old-grid-spacing *current-grid-spacing*))
     (set-force-parameters ,spacing)
     (prog1 (progn ,@body)
    (set-force-parameters old-grid-spacing))))

(defun stretching-energy (length ideal-length)
  (* (- length ideal-length)
     (- length ideal-length)))

(defun buckling-energy (length ideal-length)
  (* *buckling-energy-linear-term* (- 1 (/ length ideal-length))))

(defun strain-energy-per-line (point1 point2 ideal-length)
  (if (null point2)
      0
      (let* ((length (distance-between (point-location point1)
       (point-location point2))))
(if (< length ideal-length)
    (min (stretching-energy length ideal-length)
 (buckling-energy length ideal-length))
    (stretching-energy length ideal-length)))))

;for testing
(defun tabulate-strain-energy (grid-size &optional (step-size (*
.1 grid-size)) (start 0) (finish (* 1.5 grid-size)))
  (with-grid-spacing grid-size
    (do ((length start (+ length step-size))
 (result ())
 (last 0 energy)
```

```
 (energy))
((> length finish)
 (nreverse result))
      (setq energy
    (if (< length grid-size)
(min (stretching-energy length grid-size)
     (buckling-energy length grid-size))
(stretching-energy length grid-size)))
      (format zl:terminal-io "~%~f~20t~f~40t~f" length energy (-
energy last))
      (push (list length energy (- energy last))
    result))))

(defun strain-energy (point)
  (let* ((ideal-length (point-grid-spacing point))
 (diagonal-length (* 1.414 ideal-length)))
    (+ (strain-energy-per-line point (point-to-u+ point) ideal-
length)
       (strain-energy-per-line point (point-to-v+ point) ideal-
length)
       (* *diagonal-force-ratio*
  (+ (strain-energy-per-line point (point-to-u+v+ point) diagonal-
length)
     (strain-energy-per-line point (point-to-u+v- point) diagonal-
length))))))

(defun strain-energy-around (point)
  (let* ((ideal-length (point-grid-spacing point))
 (diagonal-length (* 1.414 ideal-length)))
    (+ (strain-energy-per-line point (point-to-u+ point) ideal-
length)
       (strain-energy-per-line point (point-to-v+ point) ideal-
length)
       (strain-energy-per-line point (point-to-u- point) ideal-
length)
       (strain-energy-per-line point (point-to-v- point) ideal-
```

```
length)
       (* *diagonal-force-ratio*
  (+ (strain-energy-per-line point (point-to-u+v+ point) diagonal-
length)
     (strain-energy-per-line point (point-to-u+v- point) diagonal-
length)
     (strain-energy-per-line point (point-to-u-v+ point) diagonal-
length)
     (strain-energy-per-line point (point-to-u-v- point) diagonal-
length))))))

(defun gravitational-energy (point)
  (point-z point))

(defun energy-of-point (point)
  (realpart (+ (* *elasticity* (strain-energy point))
       (* *gravity* (gravitational-energy point)))))


(defun energy-of-cloth-around-point (point)
  (realpart (+ (* *elasticity* (strain-energy-around point))
       (* *gravity* (gravitational-energy point)))))



(zl:comment ;this is the fast old way.

(defun strain-energy-per-line (point1 point2 ideal-length^2)
  (if (null point2)
      0
      (let ((length^2 (square-of-distance-between (point-location
point1)
  (point-location point2))))
(if (< length^2 ideal-length^2)
    (* *compressibility-ratio*
       (square (- length^2 ideal-length^2)))
```

```
    (square (- length^2 ideal-length^2))))))

(defun bending-energy (point)
  (+ (bending-energy-per-line (point-to-u- point) point (point-
to-u+ point))
     (bending-energy-per-line (point-to-v- point) point (point-
to-v+ point))))

(defparameter *stiffness* 0)
;;; Set to .5, makes rubber sheet
(defparameter *elasticity* .5)
;;; Usually 0.25
(defparameter *compressibility-ratio* 0.25)
(defparameter *diagonal-strength* .5)

(defun bending-energy-per-line (point1 point2 point3)
  (if (or (not point1)(not point3))
      0
      (square (angle-between-vectors (subtract-vectors (point-
location point1)
       (point-location point2))
     (subtract-vectors (point-location point2)
       (point-location point3))))))




) ;end comment
```

File RELAXATION:

```
;;; -*- Syntax: Common-lisp; Package: USER -*-

(defun apex-of-parabola (x1 y1 x2 y2 x3 y3)
  (if (or (= x1 x2)
  (= x1 x3)
  (= x2 x3))
      x2 ;kluge
      (let ((a (/ (+ (* (- y3 y1)
(- x2 x1))
     (* (- y1 y2)
(- x3 x1)))
  (* (- x3 x2)
     (- x3 x1)
     (- x2 x1)))))
(if (zerop a)
    x2 ;kluge
    (- (/ (- (/ (- y2 y1)
(- x2 x1))
     (* a (+ x2 x1)))
  (* 2 a)))))))

(defun test-apex-of-parabola (a b)
  (print (- (/ b 2 a)))
  (let ((x1 (random 10))
(x2 (random 10))
(x3 (random 10))
(c (random 10)))
    (apex-of-parabola x1 (+ c (* b x1) (* a x1 x1))
      x2 (+ c (* b x2) (* a x2 x2))
      x3 (+ c (* b x3) (* a x3 x3)))))

;simple one-point-relaxer
(defun slide-along-vector (point)
  (let ((vector (nearly-force-vector point))
(distance 5e-4)
(last-distance 0))
```

```
    (block got-it
      (loop (if (< (energy-of-point-moved-along-vector-by-distance
point vector last-distance)
   (energy-of-point-moved-along-vector-by-distance point vector
distance))
(return-from got-it
  (offset-point point (scale-vector vector last-distance))))
    (setq last-distance distance)
    (setq distance (* 2 distance))))))

;slightly more complicated one-point relaxer, but should still
work.
;Uses last three measurements for three-point parabolic approximation.
;Unfortunately, experiment shows that it only speeds things up
by a few
;percent per relaxation cycle.  (While using forcible relaxation)
But
;since SLIDE-ALONG-VECTOR is inefficient in that it evaluates point
;energy twice where it dosen't really have to.  So let's use this
one.
;Eventually we should be smart about guessing where to start the
search
;instead of starting it at 2.5e-4.
(defun search-along-vector (point)
  (let* ((vector (nearly-force-vector point))
 (distance 5e-4)
 (last-distance 2.5e-4)
 (last-last-distance 0)
 (energy (energy-of-point-moved-along-vector-by-distance point
vector distance))
 (last-energy (energy-of-point-moved-along-vector-by-distance point
vector last-distance))
 (last-last-energy (energy-of-point-moved-along-vector-by-distance
point vector last-last-distance)))
    (offset-point point
  (scale-vector vector
```

```
(block got-it
  (loop (if (< last-energy energy) ;energy started going up as
we move out
    (return-from got-it
      (apex-of-parabola distance energy
last-distance last-energy
last-last-distance last-last-energy)))
(shiftf last-last-distance
last-distance
distance
(* 2 distance))
(shiftf last-last-energy
last-energy
energy
(energy-of-point-moved-along-vector-by-distance point vector distance)))))))

(defvar *best-relaxation-distance* 5e-4)

(defun search-along-vector-starting-smart (point)
  (let* ((vector (nearly-force-vector point))
 (hi (* 6 *best-relaxation-distance*))
 (lo (* .75 *best-relaxation-distance*))
 (hi-energy (energy-of-point-moved-along-vector-by-distance point
vector hi))
 (lo-energy (energy-of-point-moved-along-vector-by-distance point
vector lo))
 (zero-energy (energy-of-point-moved-along-vector-by-distance point
vector 0))
 (best-distance
   (block got-it
     (loop (cond ((and (< lo-energy zero-energy)
       (>= hi-energy zero-energy))
  ;; We're in parabolic domain
  (return-from got-it
    (apex-of-parabola hi hi-energy
      lo lo-energy
```

```
      0 zero-energy)))
  ((< lo 1e-4) ;prevents flonum underflow in pathological cases
   (return-from got-it hi))
  ((>= lo-energy zero-energy)
   ;; Too far out
   (shiftf hi lo (/ lo 2))
   (shiftf hi-energy lo-energy (energy-of-point-moved-along-vector-
by-distance point vector lo)))
  ((< hi-energy zero-energy)
   ;; Too far in- still in bowl
   (shiftf lo hi (* hi 2))
   (shiftf lo-energy hi-energy (energy-of-point-moved-along-vector-
by-distance point vector hi)))
  (t
   (ferror "This should never happen-- return 0 to continue if it
does.")))))))
    (setq *best-relaxation-distance* (/ (+ *best-relaxation-distance*
   *best-relaxation-distance*
   *best-relaxation-distance*
   best-distance)
4))
    (set-point-location point
(push-point-out-of-all-ellipsoids (add-vectors (point-location
point)
       (scale-vector vector best-distance))))))

(defparameter *force-vector-perturbation-strength* .2)

;a unit vector pointing in nearly the direction of fastest en-
ergy decrease
(defun nearly-force-vector (point)
  (make-unit-vector (v+ (make-random-vector *force-vector-perturbation-
strength*)
(scale-vector (make-unit-vector (force-vector point))
      -1))))
```

```
(defun force-vector (point)
  (let* ((current-energy (energy-of-cloth-around-point point)))
    (make-vector (- (energy-of-point-if-offset point (make-vector
1e-3 0 0))
    current-energy)
 (- (energy-of-point-if-offset point (make-vector 0 1e-3 0))
    current-energy)
 (- (energy-of-point-if-offset point (make-vector 0 0 1e-3))
    current-energy))))

(defun energy-of-point-moved-along-vector-by-distance (point vec-
tor distance)
  (energy-of-point-if-point-is-moved-to-location point
 (push-point-out-of-all-ellipsoids (add-vectors (point-location
point)
(scale-vector vector distance)))))

(defun move-point-tabulating (point vector step-size start finish)
  (do ((distance start (+ distance step-size))
       (result ())
       (last 0 energy)
       (energy))
      ((> distance finish)
       (nreverse result))
    (setq energy (energy-of-point-moved-along-vector-by-distance
point vector distance))
    (format zl:terminal-io "~%~f~20t~f~40t~f" distance energy (-
energy last))
    (push (list distance energy (- energy last))
  result)))

;sucky one-point relaxer
(defun perturb-in-a-random-direction (point distance)
  (let* ((offset (make-random-vector (* 2 distance)))
 (new-location (add-vectors (point-location point)
    offset)))
```

```
    (if (<= (energy-of-point-if-point-is-moved-to-location point
new-location)
    (energy-of-point point))
(progn (set-point-location point new-location)
       (vector-max-component offset))
;; If you didn't reduce the energy, you're probably going too far.
(* .99 distance))))

(defun energy-of-point-if-point-is-moved-to-location (point location)
  (let ((old-location (point-location point)))
    (setf (point-location point) location)
    (prog1 (energy-of-cloth-around-point point)
   (setf (point-location point) old-location))))

(defun energy-of-point-if-offset (point offset)
  (energy-of-point-if-point-is-moved-to-location point
 (add-vectors (point-location point)
      offset)))

(defun relax-cloth-once (cloth one-point-relaxer)
  (with-grid-spacing (cloth-grid-spacing cloth)
    (do-points-in-red-black-order (point cloth)
      (funcall one-point-relaxer point))
    ;;for testing
    (draw-cloth cloth)))

(defun uniformly-distributed-random-vector (&optional (length 1))
  (do ((try (make-random-vector 1)
    (make-random-vector 1)))
      ((< (vector-magnitude try)
  1)
       (scale-vector try (/ length (vector-magnitude try))))))

(defun relax-cloth-keeping-records (cloth one-point-relaxer &op-
tional (number-of-cycles 100))
  (let ((result ()))
```

```
    (dotimes (n number-of-cycles)
      (push (print (list n (energy-of-whole-cloth cloth)))
    result)
      (draw-cloth cloth)
      (relax-cloth-once cloth one-point-relaxer))
    result))

(defvar foo)

(defun drop-banner (&optional (cloth (transform-whole-cloth (make-
canvas 5 .9 .9 t t)
    (make-rotate-matrix 0 90 0))))
  (setq foo cloth)
  (relax-cloth-keeping-records cloth #'slide-along-vector 5))


;two copies?



(defun tension-energy (point)
  (apply #'+ (remove-if #'null (list (if (point-to-u+ point)
 (square (pt-to-pt-distance point (point-to-u+ point))))
    (if (point-to-u- point)
 (square (pt-to-pt-distance point (point-to-u- point))))
    (if (point-to-v+ point)
 (square (pt-to-pt-distance point (point-to-v+ point))))
    (if (point-to-v- point)
 (square (pt-to-pt-distance point (point-to-v- point)))))))

(defun pt-to-pt-distance (point1 point2)
  (if (or (eq (point-fixity point1) :dummy)
  (eq (point-fixity point2) :dummy))
      0
      (distance-between (point-location point1)
(point-location point2))))
```

```
(defun energy-of-whole-cloth (cloth)
  (let ((energy 0))
    (dotimes (u (array-dimension (cloth-points cloth) 0))
      (dotimes (v (array-dimension (cloth-points cloth) 1))
(incf energy (energy-of-point (aref (cloth-points cloth) u v)))))
    energy))

(deff energy-of-cloth #'energy-of-whole-cloth)

(defun interpolate-cloth (cloth0 cloth1 parameter)
  (let ((result (copy-cloth cloth0)))
    (do-point-coordinates (u v cloth0 *all-point-fixity-types*)
      (set-point-location (point-at-integer result u v)
  (linearly-interpolate-vectors parameter
(point-location (point-at-integer cloth0 u v))
(point-location (point-at-integer cloth1 u v)))))
    result))

(defun relax-cloth-forcibly (cloth)
  (let* ((old (copy-cloth cloth))
 (new (relax-cloth-once cloth #'search-along-vector))
 (even-newer (interpolate-cloth old new 3))
 (new-energy (energy-of-cloth new))
 (even-newer-energy (energy-of-cloth even-newer))
 (place-to-interpolate-to (apex-of-parabola 0 (energy-of-cloth
old)
    1 new-energy
    3 even-newer-energy)))
    (interpolate-cloth old new
       (if (< place-to-interpolate-to 7)
   place-to-interpolate-to
   (apex-of-parabola 1 new-energy
     3 even-newer-energy
     place-to-interpolate-to (energy-of-whole-cloth
       (interpolate-cloth old new
```

```
  place-to-interpolate-to)))))))
```

File SOLIDS:

```
;;; -*- Mode: Lisp; Package: COMMON-LISP-USER; Syntax: Common-
lisp -*-

(defstruct (ellipsoid :conc-name
      (:constructor make-ellipsoid-internal))
  transform
  inverse
  transform-with-translation
  x-axis
  y-axis
  z-axis)

(defun make-ellipsoid (x-length y-length z-length &optional (x-
rot 0) (y-rot 0) (z-rot 0) (center-x 0) (center-y 0) (center-z
0))
  (let ((transform (math:multiply-matrices (make-scale-matrix (/
2 x-length) (/ 2 y-length) (/ 2 z-length))
   (make-rotate-matrix x-rot y-rot z-rot))))
    (make-ellipsoid-internal :transform transform
     :inverse (math:invert-matrix transform)
     :transform-with-translation (math:multiply-matrices transform
 (make-translate-matrix (- center-x)
(- center-y)
(- center-z)))
     :x-axis (multiply-vector-by-matrix (make-vector 1 0 0) transform)
     :y-axis (multiply-vector-by-matrix (make-vector 0 1 0) transform)
     :z-axis (multiply-vector-by-matrix (make-vector 0 0 1) transform))))
```

```
;Actually makes a ball 10 km in diameter.
(defun make-floor (location-to-pass-through normal-vector)
  (let ((the-center (v+ location-to-pass-through (scale-vector
(make-unit-vector normal-vector) -5000))))
    (make-ellipsoid 10000 10000 10000 0 0 0 (vector-x the-center)(vector-
y the-center)(vector-z the-center))))

;;; Given a vector and a point a which to base the vector, computes
;;; whether the vector intersects a unit sphere centered at the
origin.
;;; If it does, returns the min and max values of the parameter.
(defun sphere-intersection (base-point vector)
  (let* ((p*v (dot-product base-point vector))
 (v*v (dot-product vector vector))
 (discriminant (- (* p*v p*v)
  (* v*v
     (- (dot-product base-point base-point) 1)))))
    (if (plusp discriminant)
(let ((sqrt-discriminant (sqrt discriminant)))
  (list (/ (- (- p*v) sqrt-discriminant)
   v*v)
(/ (+ (- p*v) sqrt-discriminant)
   v*v)))
nil)))


(defun ellipsoid-intersection-distance (ellipsoid base-point vector)
  (sphere-intersection (multiply-vector-by-matrix base-point (ellipsoid-
transform-with-translation ellipsoid))
       (multiply-vector-by-matrix vector (ellipsoid-transform ellipsoid))))

;returns nil if no intersection.
(defun ellipsoid-intersection-point (ellipsoid base-point vector)
  (let ((distance (remove-if-not #'plusp (ellipsoid-intersection-
distance ellipsoid base-point vector))))
```

```
    (if distance
(v+ base-point (scale-vector vector (apply #'min distance)))
nil)))

; What?  You say you don't understand this code?  Well, it only
took me an
; hour to figure it out.  That counts as intuitive in my book.
(defun normal-to-ellipsoid (ellipsoid point)
  (let ((transformed-base-point (multiply-vector-by-matrix point
(ellipsoid-transform-with-translation ellipsoid))))
    (make-vector (dot-product transformed-base-point
      (ellipsoid-x-axis ellipsoid))
 (dot-product transformed-base-point
      (ellipsoid-y-axis ellipsoid))
 (dot-product transformed-base-point
      (ellipsoid-z-axis ellipsoid)))))

;returns point on surface of ellipsoid close to POINT.  If point
is not inside ellipsoid, returns NIL.
(defun push-point-out-of-ellipsoid (ellipsoid point)
  (ellipsoid-intersection-point ellipsoid point (make-unit-vector
(normal-to-ellipsoid ellipsoid point))))

(defun draw-normals-on-ellipsoid (ellipsoid &optional (radius 1)(times-
to-do-it 989999))
  (dotimes (n times-to-do-it)
    (let* ((point (make-random-vector radius))
   (vector (make-random-vector radius))
   (int (ellipsoid-intersection-distance ellipsoid point vector)))
      (when int
(draw-vector-based-at-location (scale-vector (make-unit-vector
(normal-to-ellipsoid ellipsoid
    (v+ point
(scale-vector vector (car int)))))
     .2)
       (v+ point (scale-vector vector (car int)))
```

```
(draw-3d-point-at-vector (v+ point (scale-vector vector (car int)))))))

(defun draw-normals-on-all-ellipsoids (&optional (times-to-do-
it 989999))
  (dolist (obj *all-objects*)
    (draw-normals-on-ellipsoid obj 1 times-to-do-it)))

(defparameter *all-objects* nil)

(defun clear-all-objects ()
  (setq *all-objects* nil)
  )

(defun add-objects (&rest objects)
  (setq *all-objects* (append objects *all-objects*))
  )

(defun push-point-out-of-all-ellipsoids (point)
  (dolist (obj *all-objects*)
    (let ((new-point (push-point-out-of-ellipsoid obj point)))
      (if new-point
  ;debug
  (progn ;(draw-3d-line-between-vectors point new-point *highlight-
color*)
 ;(draw-3d-dot-at-vector point .7)
 (setq point new-point)))))
  point)

(defun test-push-point (radius)
  (set-display-parameters radius (- radius) radius (- radius) ra-
dius (- radius))
  (clear-all-objects)
  (add-objects (make-floor (make-vector 0 0 -.5)
   (make-vector 0 0 1))
       (make-ellipsoid .7 .5 1 0 0 0 0 0 0))
  (let ((base (make-random-vector radius))
```

```
(end (make-random-vector radius)))
    (do-in-range (d 0 1 20)
      (let* ((s (linearly-interpolate-vectors d base end))
     (e (push-point-out-of-all-ellipsoids s)))
(if (not (< (vector-magnitude (v- s e)) 1e-6))
    (progn (draw-3d-dot-at-vector s .7)
   (draw-3d-dot-at-vector e .7 *highlight-color*)
   (draw-3d-line-between-vectors s e)))))))
```

File MULTIGRID:

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10
-*-

(defun coarsen (cloth)
  (let ((result (make-cloth-with-point-array (ceiling (/ (u-array-
limit cloth) 2))
     (ceiling (/ (v-array-limit cloth) 2))
     (* 2 (cloth-grid-spacing cloth)))))
    (flet ((average-sideways (point)
     (if (and (point-to-u- point)
      (point-to-u+ point))
 (weighted-mean 1/4 (point-to-u- point)
1/2 point
1/4 (point-to-u+ point))
 (point-location point))))
      (dotimes (u (u-array-limit result))
(dotimes (v (v-array-limit result))
  (let ((point (point-at-integer cloth (* u 2) (* v 2))))
    (set-point-location (point-at-integer result u v)
(if (and (point-to-v- point)
 (point-to-v+ point))
    (weighted-mean 1/4 (average-sideways (point-to-v- point))
```

```
   1/2 (average-sideways point)
   1/4 (average-sideways (point-to-v+ point)))
    (point-location point)))
    (setf (point-fixity (point-at-integer result u v))
  (if (or (eql (point-fixity point) :fixed)
  (and (point-to-u+ point)
       (eql (point-fixity (point-to-u+ point)) :fixed))
  (and (point-to-v+ point)
       (eql (point-fixity (point-to-v+ point)) :fixed))
  (and (point-to-u- point)
       (eql (point-fixity (point-to-u- point)) :fixed))
  (and (point-to-v- point)
       (eql (point-fixity (point-to-v- point)) :fixed))
  (and (point-to-u+v+ point)
       (eql (point-fixity (point-to-u+v+ point)) :fixed))
  (and (point-to-u+v- point)
       (eql (point-fixity (point-to-u+v- point)) :fixed))
  (and (point-to-u-v+ point)
       (eql (point-fixity (point-to-u-v+ point)) :fixed))
  (and (point-to-u-v- point)
       (eql (point-fixity (point-to-u-v- point)) :fixed)))
      :fixed
      :free)))))
      result)))

;Takes a list of alternating weights and (point or vector or NIL)s.
Averages together locations and vectors but not NILs.
(defun weighted-mean (&rest alternating-weights-and-points)
  (do ((list alternating-weights-and-points
     (cddr list))
       (total-vector (make-vector 0 0 0)
     (if (second list)
 (v+ (scale-vector (if (typep (second list) 'point)
       (point-location (second list))
       (second list))
   (first list))
```

```
    total-vector)
 total-vector))
       (total-weight 0
     (if (second list)
 (+ (first list) total-weight)
 total-weight)))
      ((null list)
       (scale-vector total-vector (/ 1 total-weight)))))


;uses bilinear interpolation.  This is not the right thing.  It
should use cubics.
(defun refine (cloth &optional (u-size (- (* 2 (u-array-limit cloth))
1))
     (v-size (- (* 2 (v-array-limit cloth)) 1)))
  (flet ((set-point-location-safely (point-or-nil loc)
   (if point-or-nil (set-point-location point-or-nil loc))))
    (let ((result (make-cloth-with-point-array u-size v-size
       (* 1/2 (cloth-grid-spacing cloth)))))
      (dotimes (u (u-array-limit cloth))
(dotimes (v (v-array-limit cloth))
  (set-point-location-safely (point-at-integer result (* 2 u)(*
v 2))
     (copy-vector (point-location (point-at-integer cloth u v))))
  (set-point-location-safely (point-at-integer result (+ 1 (* 2
u))(* v 2))
     (weighted-mean 1/2 (point-at-integer cloth u v)
    1/2 (point-at-integer cloth (+ 1 u) v)))
  (set-point-location-safely (point-at-integer result (* 2 u)(+
1 (* v 2)))
     (weighted-mean 1/2 (point-at-integer cloth u v)
    1/2 (point-at-integer cloth u (+ 1 v))))
  (set-point-location-safely (point-at-integer result (+ 1 (* u
2)) (+ 1 (* v 2)))
     (weighted-mean 1/4 (point-at-integer cloth u v)
    1/4 (point-at-integer cloth u (+ 1 v))
```

```
     1/4 (point-at-integer cloth (+ 1 u) v)
     1/4 (point-at-integer cloth (+ 1 u) (+ 1 v))))))
       result)))

(defun refine-with-cubic-spline (cloth &optional (u-size (- (*
2 (u-array-limit cloth)) 1))
       (v-size (- (* 2 (v-array-limit cloth)) 1)))
  (flet ((set-point-location-safely (point-or-nil loc)
   (if point-or-nil (set-point-location point-or-nil loc))))
    (let ((result (make-cloth-with-point-array u-size v-size
       (* 1/2 (cloth-grid-spacing cloth)))))
      (dotimes (u (u-array-limit cloth))
(dotimes (v (v-array-limit cloth))
  (set-point-location-safely (point-at-integer result (* 2 u)(*
v 2))
     (copy-vector (point-location (point-at-integer cloth u v))))
  (set-point-location-safely (point-at-integer result (+ 1 (* 2
u))(* v 2))
     (cubic-spline-points (point-at-integer cloth (- u 1) v)
  (point-at-integer cloth u v)
  (point-at-integer cloth (+ u 1) v)
  (point-at-integer cloth (+ u 2) v)))
  (set-point-location-safely (point-at-integer result (* 2 u)(+
1 (* v 2)))
     (cubic-spline-points (point-at-integer cloth u (- v 1))
  (point-at-integer cloth u v)
  (point-at-integer cloth u (+ v 1))
  (point-at-integer cloth u (+ v 2))))))
      (dotimes (u (floor (/ u-size 2)))
(dotimes (v (floor (/ v-size 2)))
  (set-point-location-safely (point-at-integer result (+ 1 (* 2
u)) (+ 1 (* v 2)))
     (cubic-spline-points (point-at-integer result (- (* 2 u) 2)
(+ 1 (* v 2)))
  (point-at-integer result (* 2 u) (+ 1 (* v 2)))
  (point-at-integer result (+ (* 2 u) 2) (+ 1 (* v 2)))
```

```
  (point-at-integer result (+ (* 2 u) 4) (+ 1 (* v 2)))))))
      result)))

;Actually a highly specialized spline.  Takes 4 points with spline
;parameters of -1.5, -.5, .5, and 1.5 and returns the value for
the
;point with parameter 0.  If some of the points are NILs, it uses
a
;lower-order fit.
(defun cubic-spline-points (point1 point2 point3 point4)
  (cond ((and point1 point2 point3 point4)
 (v+ (scale-vector (point-location point1) -1/16)
     (scale-vector (point-location point2) 9/16)
     (scale-vector (point-location point3) 9/16)
     (scale-vector (point-location point4) -1/16)))
((and (not point1) point2 point3 point4)
 (v+ (scale-vector (point-location point2) 3/8)
     (scale-vector (point-location point3) 3/4)
     (scale-vector (point-location point4) -1/8)))
((and point1 point2 point3 (not point4))
 (v+ (scale-vector (point-location point1) -1/8)
     (scale-vector (point-location point2) 3/4)
     (scale-vector (point-location point3) 3/8)))
((and point1 point2 (not point3) (not point4))
 (v+ (scale-vector (point-location point1) -1/2)
     (scale-vector (point-location point2) 3/2)))
((and (not point1) (not point2) point3 point4)
 (v+ (scale-vector (point-location point3) 3/2)
     (scale-vector (point-location point4) -1/2)))
((and (not point1) point2 point3 (not point4))
 (v+ (scale-vector (point-location point2) 1/2)
     (scale-vector (point-location point3) 1/2)))
(t
 (ferror "CUBIC-SPLINE recieved a configuration of missing points
it didn't know how to handle."))))
```

```
(defun copy-fixities (into-cloth cloth-to-copy-fixities-from)
  (dotimes (u (u-array-limit into-cloth))
    (dotimes (v (v-array-limit into-cloth))
      (setf (point-fixity (point-at-integer into-cloth u v))
    (point-fixity (point-at-integer cloth-to-copy-fixities-from
u v)))
      (if (eql :fixed (point-fixity (point-at-integer cloth-to-
copy-fixities-from u v)))
  (setf (point-location (point-at-integer into-cloth u v))
(point-location (point-at-integer cloth-to-copy-fixities-from u
v)))))))

(defun cloth-difference (cloth0 cloth1)
  (let ((result (copy-cloth cloth0)))
    (do-point-coordinates (u v cloth0 *all-point-fixity-types*)
      (set-point-location (point-at-integer result u v)
  (v- (point-location (point-at-integer cloth1 u v))
      (point-location (point-at-integer cloth0 u v)))))
    result))

(defun cloth-sum (cloth0 cloth1)
  (let ((result (copy-cloth cloth0)))
    (do-point-coordinates (u v cloth0 *all-point-fixity-types*)
      (set-point-location (point-at-integer result u v)
  (v+ (point-location (point-at-integer cloth1 u v))
      (point-location (point-at-integer cloth0 u v)))))
    result))

;side-effects the cloth.
(defun push-cloth-out-of-all-objects (cloth)
  (do-points (pt cloth)
    (set-point-location pt
(push-point-out-of-all-ellipsoids (point-location point)))))

(defun relax-at-coarsest-level (cloth times-for-coarsest)
  (dotimes (n times-for-coarsest) ;do this a lot cause it's so
```

```
fast.
    (setq cloth (relax-cloth-once cloth #'search-along-vector-
starting-smart)))
  cloth)

(defun coarsest-level-p (cloth coarsest)
  (<= (min (u-array-limit cloth)
  (v-array-limit cloth))
     coarsest))

(defun interpolate-correction (cloth-to-modify coarse-begin coarse-
relaxed)
  (cloth-sum cloth-to-modify (refine-with-cubic-spline (cloth-
difference coarse-begin coarse-relaxed))))


(defparameter *relaxation-steps* 2)

;amazingly enough, we should do equal amounts of relaxation be-
fore & after multigrid.
;energy difference between 2-and-2 and 0-and-4 is -0.601 and -
0.606.
;Timing info: 293 sec for 9x9 grid.  (using V-pattern)  relax-
at-coarsest-level takes about 20 sec for 10 passes.
(defun multigrid-relax (cloth pattern &optional (coarsest 3)(times-
for-coarsest 10))
  (if (coarsest-level-p cloth coarsest)
      (setq cloth (relax-at-coarsest-level cloth times-for-coarsest))
      (progn (dotimes (n (car pattern))
       (setq cloth (relax-cloth-once cloth #'search-along-vector-
starting-smart)))
     (dolist (item (cdr pattern))
       (let ((coarse (coarsen cloth)))
 (setq cloth (interpolate-correction cloth (copy-cloth coarse)(multigrid-
relax coarse pattern))))
       (dotimes (n item)
```

```
 (setq cloth (relax-cloth-once cloth #'search-along-vector-starting-
smart))))))
  cloth)

(defvar *permanent-record* ())

(defun clock-multigrid (pattern)
  (let ((now (get-universal-time))
(cloth (multigrid-relax (make-hammock 9 .7) pattern)))
    (push (list (- (get-universal-time) now)
(energy-of-cloth cloth)
pattern
cloth)
  *permanent-record*)))

;;; Here's the permanent record:
;;; This uses #'search-along-vector for the one point relaxer (11
march 86)
;;;((319 -0.58848315 (0 1 2))
;;; (286 -0.6052529 (1 3))
;;; (443 -0.5859825 (1 1 2))
;;; (303 -0.6007395 (2 2))
;;; (380 -0.57208633 (1 1 1))
;;; (145 -0.6013322 (0 2))
;;; (170 -0.58449453 (1 1)))

;;; Switching from #'search-along-vector to
;;; #'search-along-vector-starting-smart speeds up (clock-multigrid
'(2 2))
;;; from 303 to 208 seconds, or 145%.  Whee!

;;; Using REFINE with cubic instead of bilinear interpolation slows
down
;;; from 183 to 193 sec, 5% slowdown but energy gets slightly bet-
ter and
;;; result is smoother.
```

```
;;; Changed the point-to-u+ etc functions to use caches, speeded
up
;;; clock-multigrid to 168 seconds.

;;; Changed grid-spacing from a rational to a flonum, sped up to
96
;;; seconds.  Infuckingcredible!

;;; Changed to new strain energy expression, sped up to 56 seconds.

;;; Changed to new energy-of-cloth-around-point, sped up to 41
seconds.

;;; Took out draw-cloth for debugging in relax-cloth-once, now
runs in 21 seconds.
```

File TEST-SAMPLES:

```
;;; -*- Mode: Lisp; Package: COMMON-LISP-USER; Syntax: Common-
lisp -*-
;;; A few test examples

; Sets up points with fixity :free and no location
(defun make-point-array (cloth u-size v-size)
  (let ((result (make-array (list u-size v-size))))
    (dotimes (u u-size)
      (dotimes (v v-size)
(setf (aref result u v)
      (make-point :u u
  :v v
  :cloth cloth))))
    result))
```

```
(defun make-cloth-with-point-array (u-size v-size grid-spacing)
  (let ((result (make-cloth :grid-spacing (float grid-spacing))))
    (setf (cloth-points result)
  (make-point-array result u-size v-size))
    result))

(defun make-saddle (points-per-edge)
  (let ((point-array (make-array (list points-per-edge points-
per-edge)))
(cloth (make-cloth :grid-spacing (/ 1 (- points-per-edge 1))))
(p00 (make-vector -1 0 -1))
(p01 (make-vector 0 -1 1))
(p10 (make-vector 0 1 1))
(p11 (make-vector 1 0 -1)))
    (dotimes (u points-per-edge)
      (dotimes (v points-per-edge)
(let ((u-fraction (/ u (- points-per-edge 1)))
      (v-fraction (/ v (- points-per-edge 1))))
  (setf (aref point-array u v)
(cond ((= u-fraction 0)
       (create-point (linearly-interpolate-vectors v-fraction p00
p01)
     u v cloth
     :fixed))
      ((= u-fraction 1)
       (create-point (linearly-interpolate-vectors v-fraction p10
p11)
     u v cloth
     :fixed))
      ((= v-fraction 0)
       (create-point (linearly-interpolate-vectors u-fraction p00
p10)
     u v cloth
     :fixed))
      ((= v-fraction 1)
```

```
       (create-point (linearly-interpolate-vectors u-fraction p01
p11)
     u v cloth
     :fixed))
      (t
       (create-point-xyz 0 0 0 u v cloth :free)))))))
    (setf (cloth-points cloth) point-array)
    cloth))

(defun make-prestretched-saddle (points-per-edge)
  (let ((point-array (make-array (list points-per-edge points-
per-edge)))
(cloth (make-cloth :grid-spacing (/ 1 (- points-per-edge 1))))
(p00 (make-vector -1 0 -1))
(p01 (make-vector 0 -1 1))
(p10 (make-vector 0 1 1))
(p11 (make-vector 1 0 -1)))
    (dotimes (u points-per-edge)
      (dotimes (v points-per-edge)
(let ((u-fraction (/ u (- points-per-edge 1)))
      (v-fraction (/ v (- points-per-edge 1))))
  (setf (aref point-array u v)
(create-point (linearly-interpolate-vectors u-fraction
    (linearly-interpolate-vectors v-fraction p00 p01)
    (linearly-interpolate-vectors v-fraction p10 p11))
     u v cloth
     (if (or (= u 0)
     (= v 0)
     (= u (- points-per-edge 1))
     (= v (- points-per-edge 1)))
 :fixed
 :free))))))
    (setf (cloth-points cloth) point-array)
    cloth))

(defun make-canvas (points-per-edge u-stretch v-stretch &optional
```

```
(prestretched? t)(corners-only? nil))
  (let ((point-array (make-array (list points-per-edge points-
per-edge)))
(cloth (make-cloth :grid-spacing (/ 1 (- points-per-edge 1))))
(p00 (make-vector 0 0 0))
(p01 (make-vector 0 v-stretch 0))
(p10 (make-vector u-stretch 0 0))
(p11 (make-vector u-stretch v-stretch 0)))
    (dotimes (u points-per-edge)
      (dotimes (v points-per-edge)
(let ((u-fraction (/ u (- points-per-edge 1)))
      (v-fraction (/ v (- points-per-edge 1)))
      (at-edge-p (if corners-only?
     (and (= u 0)
  (or (= v 0)
      (= v (- points-per-edge 1))))
     (or (= u 0)
 (= v 0)
 (= u (- points-per-edge 1))
 (= v (- points-per-edge 1))))))
  (setf (aref point-array u v)
(create-point (if (or at-edge-p prestretched?)
  (linearly-interpolate-vectors u-fraction
(linearly-interpolate-vectors v-fraction p00 p01)
(linearly-interpolate-vectors v-fraction p10 p11))
  (make-vector 0 0 0))
      u v cloth
      (if at-edge-p
  :fixed
  :free))))))
    (setf (cloth-points cloth) point-array)
    cloth))

(defun make-warped-cloth (points-per-edge)
  (let ((point-array (make-array (list points-per-edge points-
per-edge)))
```

```
(cloth (make-cloth :grid-spacing (/ 2 (- points-per-edge 1)))))
    (dotimes (u points-per-edge)
      (dotimes (v points-per-edge)
(let ((u-fraction (rescale u 0 (- points-per-edge 1) -1 1))
      (v-fraction (rescale v 0 (- points-per-edge 1) -1 1)))
  (setf (aref point-array u v)
(create-point-xyz u-fraction v-fraction (+ (expt u-fraction 3)
   (expt v-fraction 2))
  u v cloth :free)))))
    (setf (cloth-points cloth) point-array)
    cloth))

;Stretches along vector ALONG-VECTOR by proportion LENGTHWAYS.
perp to vector by SIDEWAYS.
(defun stretch-cloth (cloth along-vector lengthways sideways)
  (let ((rotation (untranslate-matrix (make-view-matrix (make-
vector 0 0 0)
along-vector))))
    (transform-whole-cloth cloth
   (multiply-many-matrices rotation
   (make-scale-matrix sideways sideways lengthways)
   (math:invert-matrix rotation)))))

(defun transform-whole-cloth (cloth matrix)
  (do-points (pt cloth (:free :fixed))
    (setf (point-location pt)
  (math:multiply-matrices matrix (point-location pt))))
  cloth)

(defun make-banner (length width)
  (transform-whole-cloth (make-canvas 5 length width t t)
     (make-rotate-matrix 0 90 0)))

(defun energy-of-banner (length)
  (let ((cloth (transform-whole-cloth (make-canvas 5 length .9
t t)
```

```
     (make-rotate-matrix 0 90 0))))
    (draw-cloth cloth)
    (energy-of-whole-cloth cloth)))

(defun make-hammock (points-per-side compression)
  (let ((result (make-flat-cloth points-per-side points-per-side
compression compression (/ 1 (- points-per-side 1)))))
    (setf (point-fixity (point-at result 0 0))
  :fixed)
    (setf (point-fixity (point-at result 1 0))
  :fixed)
    (setf (point-fixity (point-at result 0 .5))
  :fixed)
    (transform-whole-cloth result (make-rotate-matrix 0 0 36))))

(defun make-flat-cloth (u-size v-size u-compression v-compression
grid-spacing)
  (let ((result (make-cloth-with-point-array u-size
     v-size
     grid-spacing))
(u-spacing (* u-compression grid-spacing))
(v-spacing (* v-compression grid-spacing)))
    (do-point-coordinates (u v result)
      (set-point-location (point-at-integer result u v)
  (make-vector (* u u-spacing)
       (* v v-spacing)
       0)))
    result))

(defun make-shower-curtain (points-per-side compression &optional
(hooks points-per-side))
  (let ((result (make-flat-cloth points-per-side points-per-side
1 compression (/ 1 (- points-per-side 1)))))
    (dotimes (h hooks)
      (setf (point-fixity (point-at-integer result 0 (round (rescale
h 0 (- hooks 1) 0 (- points-per-side 1)))))
```

```
    :fixed))
    (transform-whole-cloth result (make-rotate-matrix 0 90 0))))

(defun drop-over-ball (points-per-edge ball-diameter)
  (clear-all-objects)
  (add-objects (make-ellipsoid ball-diameter ball-diameter ball-
diameter 0 0 0 .5 .5 (* -.5 ball-diameter)))
  (let ((cloth (make-flat-cloth points-per-edge points-per-edge
1 1 (/ 1 (- points-per-edge 1)))))
    (setf (point-fixity (point-at cloth .5 .5))
  :fixed)
    cloth))

(defun two-bumps (points-per-edge)
  (clear-all-objects)
  (add-objects (make-floor (make-vector 0 0 0)(make-vector 0 0
1))
       (make-ellipsoid .5 .5 .5 0 0 0 .2 .2 0)
       (make-ellipsoid .4 .4 .4 0 0 0 .6 .5 0))
  (let ((cloth (transform-whole-cloth (make-flat-cloth points-
per-edge points-per-edge .9 .9 (/ 1 (- points-per-edge 1)))
      (make-translate-matrix 0 0 .25))))
    (setf (point-fixity (point-at cloth (/ .2 .9)(/ .2 .9)))
  :fixed)
    (setf (point-location (point-at cloth (/ .2 .9)(/ .2 .9)))
  (make-vector .2 .2 .25))
    (setf (point-fixity (point-at cloth (/ .6 .9)(/ .5 .9)))
  :fixed)
    (setf (point-location (point-at cloth (/ .6 .9)(/ .5 .9)))
  (make-vector .6 .5 .2))
    cloth))

(defun make-and-draw-parabola (angle points-per-edge)
  (set-display-parameters 1.7 -1.7 1.7 -1.7 3 -3)
  (draw-cloth (let ((point-array (make-array (list points-per-
edge points-per-edge)))
```

```
(cloth (make-cloth :grid-spacing (/ 1 (- points-per-edge 1)))))
    (dotimes (u points-per-edge)
      (dotimes (v points-per-edge)
(let ((u-fraction (rescale u 0 (- points-per-edge 1) -1 1))
      (v-fraction (rescale v 0 (- points-per-edge 1) -1 1)))
  (setf (aref point-array u v)
(create-point-xyz (+ (* u-fraction (cos angle))
     (* v-fraction (sin angle)))
  (- (* v-fraction (cos angle))
     (* u-fraction (sin angle)))
  (* .8 (expt (- (* v-fraction (cos angle))
   (* u-fraction (sin angle)))
2))
  u v cloth :free)))))
    (setf (cloth-points cloth) point-array)
    cloth))
  (draw-3d-line -1.9 0 0 1.9 0 0)
  (draw-3d-line 0 -1.9 0 0 1.9 0)
  (draw-3d-line 0 0 0 0 0 2.2))
```