Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle using the Finite Element Method

by
David Tzu-Wei Chen
MSEE, Stanford University (1983)
BSEE, University of Illinois (1981)

Submitted to the Media Arts and Sciences Section, School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
at the
Massachusetts Institute of Technology
February 1992

Author ______________________________
David Chen
Media Arts and Sciences Section
October 18, 1991

Certified by [illegible]
David Zeltzer
Associate Professor of Computer Graphics
Media Arts and Sciences Section

Accepted by [illegible]
Stephen A. Benton
Chairperson, Departmental Committee on Graduate Students

Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle using the Finite Element Method

by
David Tzu-Wei Chen

Submitted to the Media Arts and Sciences Section,
School of Architecture and Planning
on October 18, 1991
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

This thesis examines muscle function through the process of making computer animation and developing interactive graphics applications. Muscle is the fundamental "motor" that drives all animal motion and so is an appropriate place to begin investigations relevant to the goal of modeling human characters. The major supposition of the thesis is that the shape changes generated by a contracting muscle will be reproduced by accurately simulating the forces involved. To examine the hypothesis, a novel computational model of skeletal muscle is presented. The geometry and underlying material properties of muscle are captured using the finite element method (FEM). A biomechanical model of muscle action is used to apply non-linear forces to the finite element mesh nodes. The techniques that are developed for fast graphical display and interactive manipulation of finite element simulations can be used both to design computer animations and directly incorporated into new kinds of applications—such as surgery simulation systems—made possible by the ever increasing power of computer workstations. Results presented indicate that the twin goals of realistic computer animation and valid biomechanical simulation of muscle can be met using the methods presented herein and can be a foundation both for animators wishing to create anatomically based characters and biomechanical engineers interested in studying muscle function.

Thesis Supervisor: David L. Zeltzer
Title: Associate Professor of Computer Graphics
This work was supported by NHK and by equipment grants from Hewlett-Packard.

Thesis Committee

Chairman [illegible]

David Zeltzer
Associate Professor of Computer Graphics
MIT Media Arts and Sciences Section

Member [illegible]

Joe Rosen
Associate Professor in Plastic and Reconstructive Surgery
Dartmouth Medical School

Member [illegible]

John Williams
Associate Professor
MIT Department of Civil Engineering

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The specific goal of this thesis is to construct a biomechanical model for simulating the changes in shape that a muscle undergoes during contraction. Posed in this way, the thesis topic falls roughly into the domain of physically-based modeling for generating computer animation. The analysis of the problem is formulated using the displacement-based finite element method (FEM) from mechanical engineering. Thus, a large part of our work will involve developing and codifing techniques to apply the FEM to making animation, and in the *dual* case, to working out computer graphics methods that allow the effective visualization of the results of finite element simulation. Lastly, in the role of media technologist, an effort will be made to evolve computer tools that are highly integrated and interactive; that is, tools to enable one to easily design specific instances of structures to be simulated, to set simulation parameters in a straightforward way, and that accommodate quick viewing of the results that are obtained. In this light, the ultimate goal for the biomechanical muscle model becomes not only computer animation, but to have it "take on a life of its own" and become an *autonomous* tool for clinicians and physiologists interested in studying muscle *function* as well as computer animators wishing to make anatomically accurate animations of muscled characters.

The general problem, as stated, is to model elements of the human body through physical simulation. Modeling and animation of the human form has long been considered a significant research area in the computer graphics field. In natural communication, our bodies transmit information about ourselves and we take cues, both narrative and physical, from what is seen of others. The human shape is an important and ubiquitous expressive tool that we would like to use in computer animations. This is also a hard problem. The human body is the antithesis of shapes that is easy to model with computer graphics. The body does not have a rigid form that can be treated in the same way as "flying logos". The skeletal motions of the body are subtly complex and well coordinated. Next to the skeleton are muscles that change shape due to the complex dynamic interactions of contraction and contact. On the surface, realistic modeling of soft, living layers of skin and the ways in which it interfaces to the outside world is also daunting.

Komatsu [Kom86] has put forth a minimum set of conditions for a computer graphics based human figure model,

1. The shape of the model must be smooth everywhere.
2. The three-dimensional form of the body is defined by the skeleton, which must change flexibly at the connecting angles between bones.
3. The shape must reflect the motions of the skeleton in accord with action, and the shape must always keep its smoothness.
4. A local change in the skeleton must affect only a small part of the shape.
5. Change of shape like the swelling of muscles must be expressed.

To meet the goals of character animation, many people have designed systems that greatly simplify the complete problem to make it possible to represent entire moving figures. Typically, the geometry is modeled to achieve the *effect* of how the underlying anatomy looks at the surface. The difference in this thesis is that we take a detailed look at the anatomy by modeling the important underlying structures (ie. muscles and bone), rather than stressing the ability to represent at the current time, a whole figure. The immediate goal of the thesis is to accurately model individual three-dimensional muscles. This is a "bottom up" approach, but because a ***physical*** model is created, our results can be applied both to produce realistic-looking animations of human characters and to help create new bioengineering applications in which computer graphics display is vital. As computer workstations become more powerful, and rendering and computation times drop, it will be possible to extend the results from the thesis in a straightforward way. The eventual goal then is not just character animation but to take a step towards the creation of an *artificial person* that can repond convincingly as its simulated muscles are activated.

1.1 Previous Work

The approaches taken to computer modeling the shape of the human figure include,

1. *geometric*—rigid limbs, like bones, in which only the static geometry is specified
2. *kinematic*—geometry of whole limb changes due to kinematic position of underlying skeleton
3. *elastic*—modeling parts of the body as non-linear visco-elastic-plastic composite materials that change shape due to the action of forces

For the geometric case, the three methods most commonly used to represent the geometry are polygonal meshes, volume primitives and surface patches. Fetter [Fet82] used contours derived from biosterometric data to generate the polygons in his "Fourth Man and Woman". Human forms from standard volume primitives include Badler's Bubbleman [BOT79], made from spheres, or Ginsberg and Maxwell's "cloud" figure [Max83], based on ellipsoids. In these examples, the shape of the limbs does not change as the character moves.

Examples of kinematic models include that of Komatsu, who parameterized spline patch control points to simulate a contracting biceps as the elbow is bent [Kom86]. Komatsu used four major spline patch surfaces to cover the head, chest, abdomen and legs of a skeleton. Chadwick et.al. in [CHP89], generalized this approach by using a "layered" technique based on free-form deformations (FFDs) to apply muscle effects onto a skeleton. His model derives the shape of a whole limb from the kinematic skeletal state. Abstract muscles are parameterized as two sets of FFDs. These FFDs are controlled by the skeleton position to simulate the gross effects of muscle contraction at the body's surface. A simple elastic model based on discretized mass points joined by Hookean springs can be added on top of this to allow for automatic squash and stretch of the face or whole limbs.

Elastic[1] models comprise one form or another of displacement analysis of an elastic continuum. This analysis can be characterized as static or dynamic, linear or non-linear, isotropic or anisotropic, and so on. The particular shape of a deformation is a function of both the internal stresses and strains within the elastic object and the external forces applied to it. Examples of computer graphics researchers modeling parts of the human body with elastic analysis include Gourret [GMTT89], who described a system for modeling the human hand with a finite element volume meshed around bone. He formulates and solves a set of statics equations for skin deformation based on bone kinematics and hand/object contact points in a grasping task. While bending and flexing of the hand flesh is nicely simulated,

[1]ie. visco-elastic-plastic models

no muscle effects or changes in the underlying shape are calculated. Pieper [Pie92], has developed a *surgical simulation* system that can be used both to create animation of the face and to simulate surgical reconstructions of the face. He performs a finite element analysis of the skin arranged as three different layers of material.

1.2 Elastic objects

Here, *single muscle masses* will be modeled as visco-elastic, deformable bodies that are subjected to non-linear forces calculated from what is known about the biomechanics of the situation. The hope is that if these forces are simulated correctly, then the correct changes in the muscle shape will be automatically produced and propagated to the surface. The shape of muscle groups will be found through the constrained interaction of an ensemble of individual muscles.

The technique used to define the dynamics of deformation for an elastic object is the FEM. More specifically, stiffness and mass matrices derived from finite element meshes of twenty-node brick elements will be developed to yield differential equations that control the displacement of the mesh nodal points. The resulting second-order matrix equations are decoupled using the *modal* technique so that dynamic simulations can be run relatively rapidly on a workstation-size computer.

Because the focus of the thesis is centered around computer graphics-based applications, the geometric shape of an elastic object will be described in a standard polyhedral format that is well suited to rendering by a graphics workstations. This emphasis leads to the following particular problems that will need to be addressed in the course of the thesis work. First, is to show how a FEM mesh can be used to approximate the elastic volume defined by an input set of polyhedral geometric data. This is a *discretization* process. Second is to show how the FEM mesh is used to derive the dynamic equilibrium equations from both the shape and the material properties of the elastic volume. Third, for computer graphics, is to show how the FEM mesh defines a *free-form deformation*[2] that can appropriately warp the computer graphic models as the mesh changes shape in response to external forces. This can aid in visualizing the simulated deformations. Lastly, forces will be developed and applied to the FEM mesh that model muscle contraction, and other effects appropriate to computer animation such as gravity, point-to-point attachment, and collisions between objects.

1.3 Goals and Contributions

To sum up, the goals and contributions of the thesis are

1. To make a 3D, dynamic, biomechanically valid model of muscle that can simulate both muscle force and muscle shape.
2. Develop techniques that will allow the finite element method to be used for making computer animation of elastically deforming objects, while not compromising its ability to approximate the dynamics of real physical structures.
3. To engineer a software testbed system that will be a useful, simple, interactive design tool for creating new simulations and animations.

While the idea of using computer graphics techniques to produce animations of moving human figures is not novel, the approach taken to the problem is. The development of an accurate force based computational model to simulate muscle shape has not been attempted in the past. The desire is to make a physically realistic model of muscle and muscle function.

[2]based on twenty-node brick elements

Why go to all this trouble? The primary application area is making computer animation, but, if the simulation is done with a sufficient degree of accuracy, then we also have a tool that can be applied in other areas. Suppose a medical researcher wants to compute the effects of a tendon transfer surgery on a muscle's ability to generate force to drive the skeleton. He needs to have a physical model of the structures in question in order to make a clinical analysis. The model developed in the thesis should be able to make such predictions, while also providing a way for the clinician to visualize the results of a simulated procedure.

1.4 Thesis Organization

The rest of the thesis is organized as follows, Chapter 2 discusses muscle anatomy and the sources of force generation within a whole muscle. Chapter 3 goes over aspects of using finite elements to simulate elastic materials. Chapter 4 concerns the complications introduced by considering the musculoskeletal system rather than isolated muscles. Chapter 5 discusses the *3d* software system implemented to do the thesis work. Chapter 6 develops a force-based finite element model of muscle that simulates both muscle force and muscle shape, and describes experiments performed using the model.

Chapter 2

Modeling a Single Muscle

This chapter contains a quick overview of skeletal muscle anatomy and dynamics [Kel71] [McM84] [VSL75] [CW74], presents a method from Zajac [ZTS86] [Zaj89] for computing the amount of force generated in a muscle and discusses sources of input data that will be used for defining the rest shapes of muscle masses. Skeletal muscle makes up from 40 to 45 percent of the total human body weight, so we expect that our emphasis on modeling muscle will allow us to simulate shape throughout most of the body.

2.1 Basic Muscle Anatomy

Muscle connects to bone through tendons, which are bundles of connective tissue. These connective tissues are composed largely of collagen, a fibrous protein found throughout the body. The center part of the muscle can be called the "belly", which is surrounded by a connective tissue sheath called the *epimysium.* The tendons are actually continuations of these connective tissue sheaths that hold the muscle together. The whole muscle is held in place within the body by extensive connective tissue layers called *fascia.*

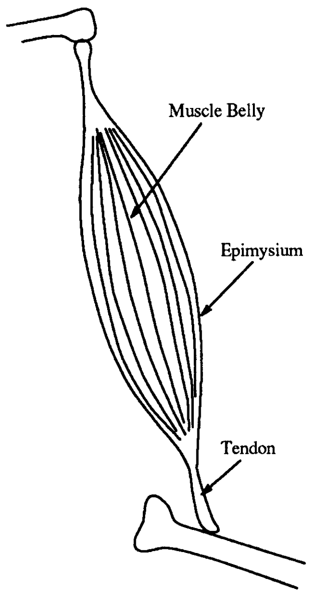


Figure 2.1: A whole muscle *adapted from* [Kel71]

The connective tissue also penetrates the muscle and divides it longitudinally into groups of muscle fibers known as *fasciculi.* It is at this level of differentiation that the muscle is supported by capillaries, veins and nerve fibers. Muscle fibers come in a wide variety of lengths—sometimes stretching the whole length of a muscle—and are usually 10 to 100 microns in diameter. The muscle fibers are composed of still smaller elements called *myofibrils* that run the whole length of the fiber. Each myofibril is about 1 to 2 microns thick. A single muscle fiber contains on the order of hundreds to thousands of myofibrils.

It is at the level of myofibril that a discussion of the contractile mechanism for a muscle usually begins. The myofibril is made up of *sarcomeres* arranged in a repeating pattern along its length. This repeating pattern is responsible for the striations or banding pattern often observed on skeletal muscles. The sarcomere is the actual functional unit of contraction for the muscle. Sarcomeres are short sections—only about 1 to 2 microns long—that contract upon suitable excitation, developing tension along their longitudinal axis. The

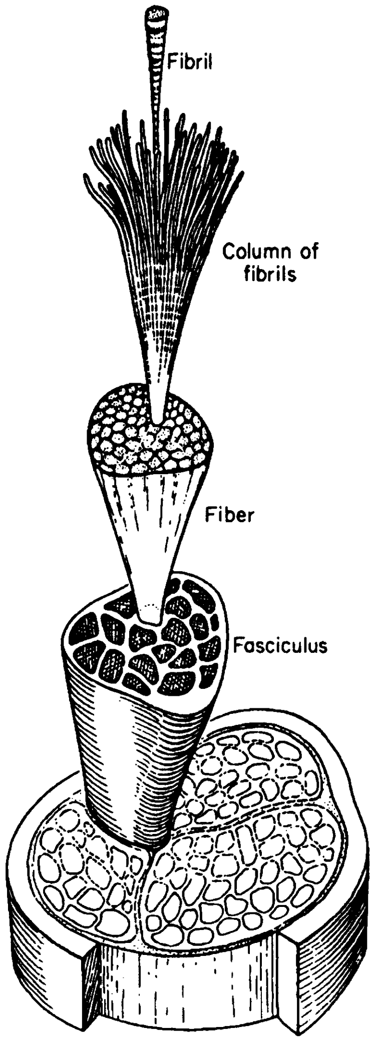


Figure 2.2: Section of muscle [Kel71]

shortening of a single muscle fiber then is due to the effect of many sarcomeres shortening in series. A bundle of muscle fibers can be thought to be many of these force generators arranged in parallel. Finally, the combined tension produced by bundles of muscle fibers is transmitted to the bones through the network of connective tissue and muscle tendons.

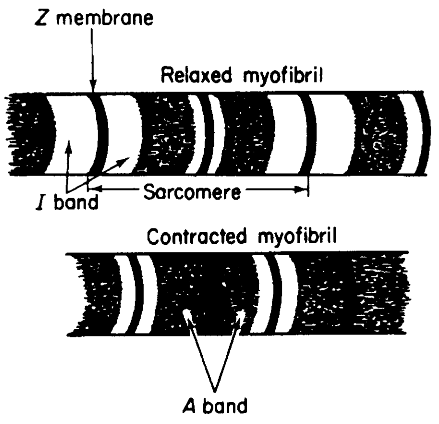


Figure 2.3: The myofibril in relaxed and contracted states [Kel71]

2.2 Muscle Force

For a given muscle, the arrangement of muscle fibers relative to its tendon attachments will determine the amount of shortening during contraction and thus the amount of force generated. In general, there are two different kinds of fiber arrangements, longitudinal and penniform. The fibers in a longitudinal muscle run parallel to each other along the entire length of the muscle. In a penniform muscle the fibers terminate at an angle relative to the tendon. There are several variations of penniform muscles. Unipennate muscles have their fibers arranged obliquely to a tendon only on one side. In a bipennate configuration, the fibers converge onto the tendon from both sides. Multipennate muscles are a combination of both unipennate and bipennate fiber arrangements.

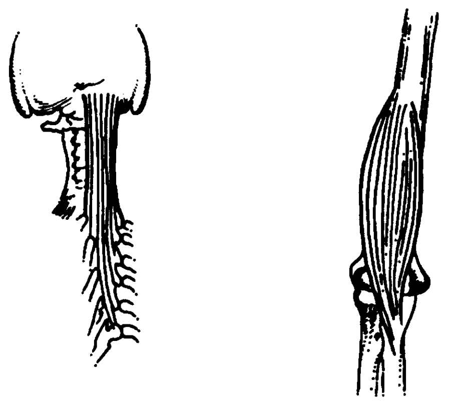


Figure 2.4: Longitudinal muscles [Kel71]

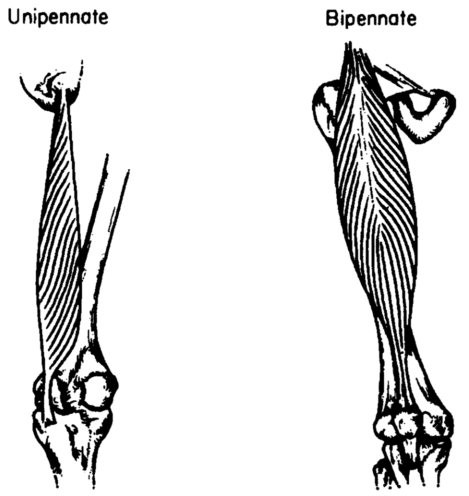


Figure 2.5: Peniform muscles [Kel71]

When the fibers of a longitudinal muscle shorten by an amount F_s, the muscle as a whole shortens an amount $M_s = F_s$. For a penniform muscle, the amount of muscle shortening depends on F_s and the pennation angle α. From Figure 2.6 it is clear that $M_s = F_s \cos\alpha$. This relationship also shows that the penniform fiber arrangement approaches the longitudinal as α approaches zero.

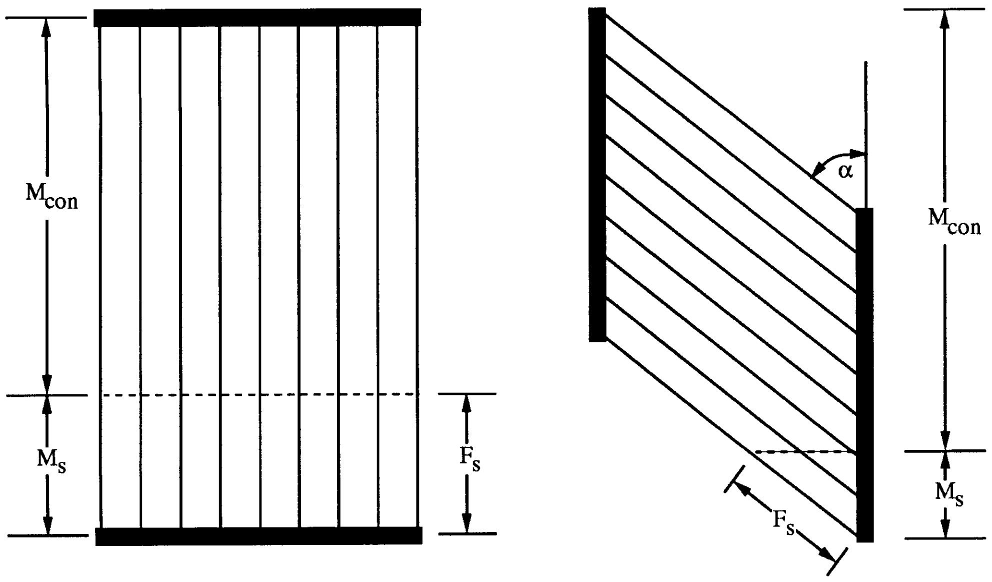


Figure 2.6: Comparison of longitudinal and penniform shortening. M_{con} is the length after contraction. *adapted from* [Kel71]

The amount of force developed by a shortening muscle depends on, among many factors, the number of contracting muscle fibers, the size of each fiber and the internal fiber arrangement. The product of the number of fibers and the fiber size is itself an important measure and is called the ***physiological cross section***, which, for a longitudinal muscle, can be determined by making a transverse cut through the belly of the muscle. For a pennate muscle, the results of such a cut would depend on the length of the muscle and α. From Figure 2.7 it can be seen that the area defined by a cut at X determines the physiological cross section for the muscle on the left, but would not include all the fibers for the unipennate muscle on the right. To determine the cross section for the unipennate muscle requires summing the area from cuts at A, B and C.

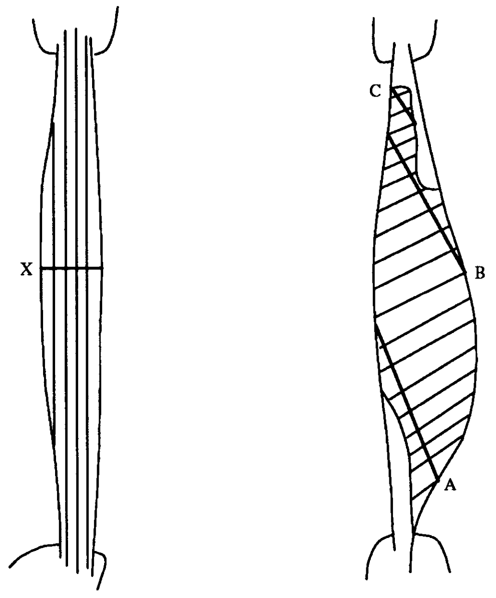


Figure 2.7: Physiological cross section *adapted from* [Kel71]

In general then, a penniform fiber arrangement trades off a lesser amount of muscle shortening for a greater number of fibers that can be involved in contraction. But pennation angle effects the final force that a muscle can develop in yet another way. Figure 2.6 also shows that if a muscle fiber generates a force in the direction of F_s, the amount of force produced along the muscle's line of action must again be scaled by $\cos\alpha$.

2.2.1 Sliding Filament Theory of Contraction

The discussion above has concerned the more or less macroscopic properties of muscle force generation. Here we will briefly touch upon the contractile mechanism within a single sarcomere. Muscles have been differentiated into at least eight separate protein structures, of which four play a role in contraction. The two most important of these are *actin* and *myosin*.

These two contractile proteins form filaments within the sarcomere, and, when viewed in cross section, can be seen to be packed hexagonally with six thin filaments surrounding each thick filament. The thick myofilaments are made of myosin, the thin myofilaments are made of actin.

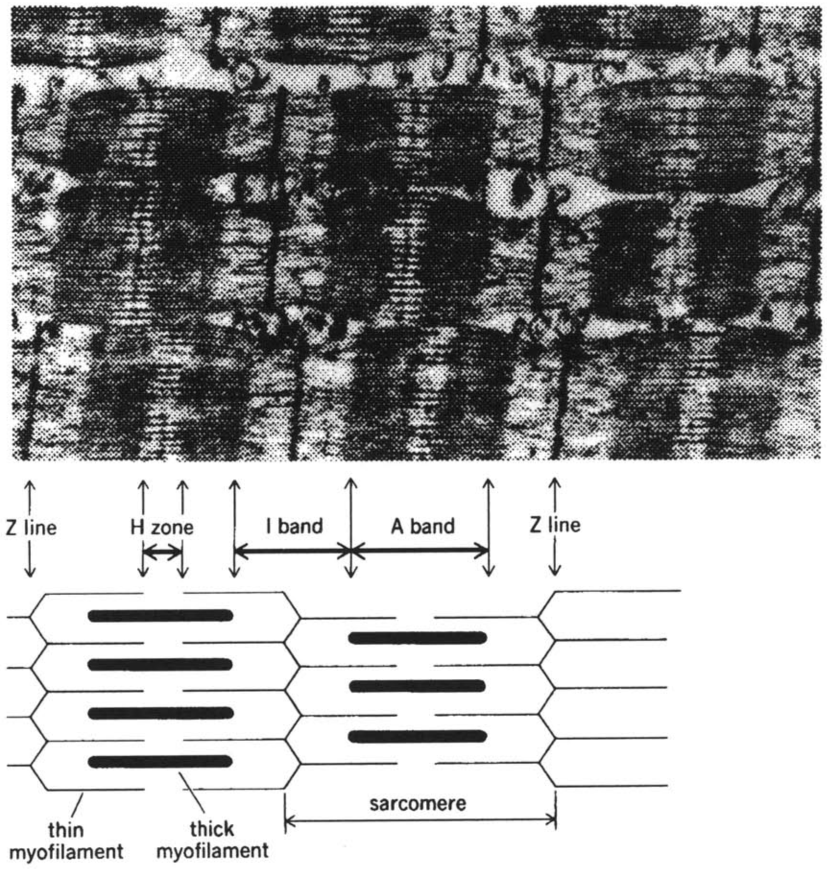


Figure 2.8: Electron micrograph showing three myofibrils in a single muscle fiber [VSL75]

The idea behind the sliding filament theory of muscle contraction is that as a muscle fiber shortens, the thin and thick myofilaments do not themselves get shorter, rather they slide across each other. This idea was presented concurrently in the same issue of Nature by A. F. Huxley and H. E. Huxley in 1954 [HN54] [HH54]. A. F. Huxley developed an interference microscope that allowed him to watch changes in the banding pattern of isolated frog muscle fibers under various circumstances. As the fiber was subjected to a purely passive stretch or shortening, the I-bands became either longer or shorter, but the length of the A-bands remained about the same. Under an isometric contraction with both fiber ends fixed, the banding was essentially invariant. Under isotonic contraction, it was again the I-bands that took up the resulting change in length. Because it was known at the time from electron microscope studies by H. E. Huxley that the A-bands were defined by birefringent rodlets of myosin and that the thin actin myofilaments extended through the I-bands into the A-bands, A. F. Huxley was able to conclude that during a muscle fiber contraction, the actin filaments were drawn into the A-bands, between the myosin rods. Furthermore, he proposed that the known dependence of the isometric tension produced in a muscle fiber to the fiber length is caused by the change in overlap between actin and myosin within a sarcomere.

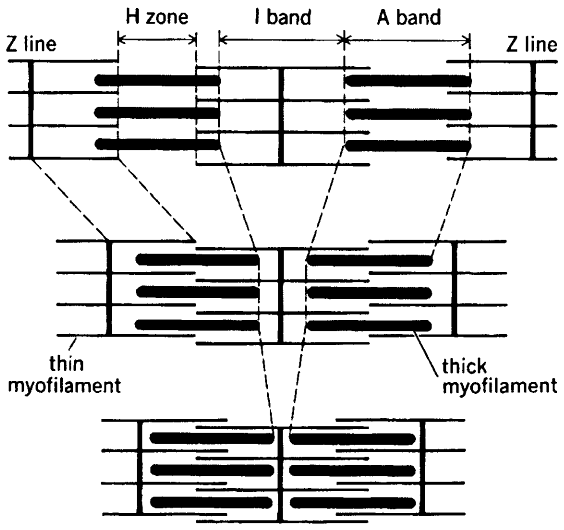


Figure 2.9: Changes in fiber banding pattern due to contraction [VSL75]

2.2.2 Hill's Force Model

One of the simplest kinds of experiments that can be done to a prepared, isolated whole muscle is to measure the force output as the muscle is stretched through a number of constant lengths. If this is done with no stimulation, then the resulting plot of force or tension to length is said to represent the *passive* elastic properties of the muscle. This passive tension-length curve has an exponential shape in which the curve gets steeper and steeper the more the muscle is elongated. This behavior is very similar to a rubber band in which the material can be pulled very easily until it is all "stretched out", and then the rubber band can feel very stiff.

If the same kind of tension-length plot is then made with the muscle fully stimulated, then a different curve is produced that has components from both *active* and passive force components. This curve, of course, should always be greater than the passive force-length plot by itself and is called the total tension-length curve. Finally, the tension-length curve that represents only the active muscle force is found by subtracting the passive curve from the total curve as in Figure 2.10. The length dependence of the developed force is wholly consistent with the sliding filament theory of muscle contraction.

Futhermore, from measurements on human subjects, A. V. Hill proposed that there is also a velocity dependent force component that counteracts the contraction force. That is, *the force exerted by the muscle decreases as the speed of shortening increases* [GH24]. It was thought at first that this phenomenon depended on an automatic regulatory mechanism within the central nervous system, but Gasser and Hill showed through quick-release experiments on isolated frog muscle that this damping effect was part of the "fundamental character" of the muscle itself. Two experiments point out this result.

First, a muscle held isometrically was suddenly allowed to shorten to a new length against no applied load. The force that was recorded fell below the amount that corre-

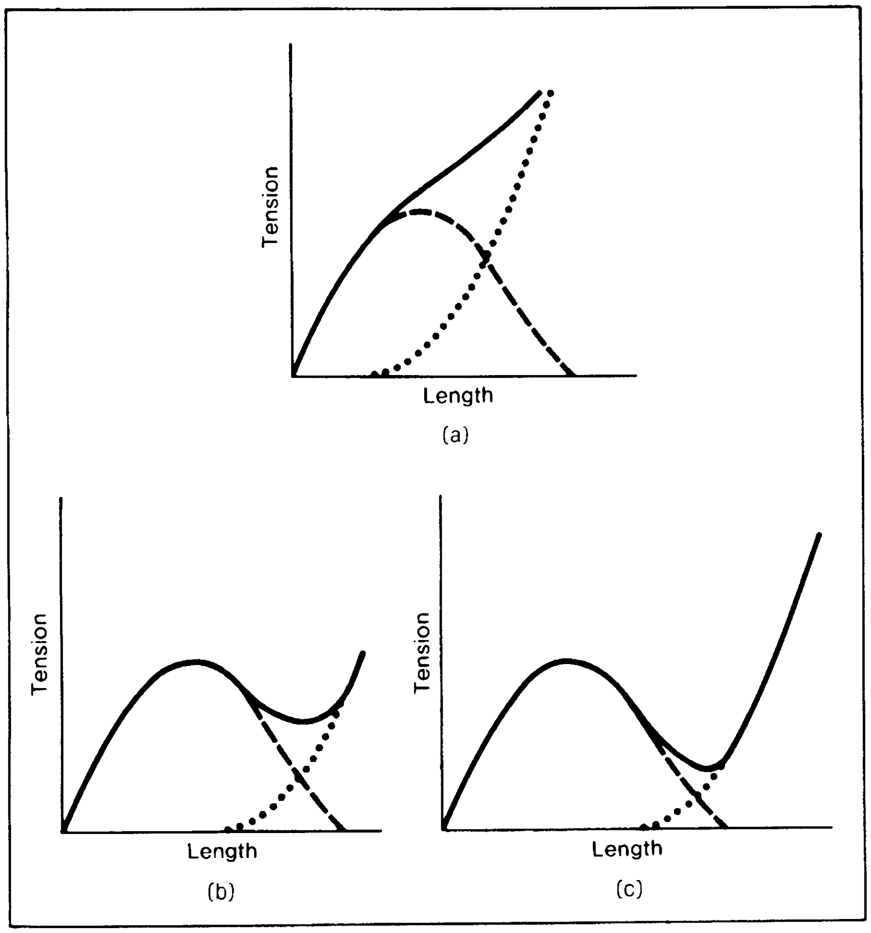


Figure 2.10: Tension-length curves. The dotted line shows the length-tension curve of the resting muscle. The total force recorded on tetanizing the muscle is shown by the solid line. The extra force developed on stimulation is shown by the dashed line. The progression from (a) to (c) results from muscles with progressively less connective tissue. [CW74]

sponded to the isometric equilibrium point of the new length and only slowly developed tension back up to that point. The apparatus to perform this experiment is shown in Figure 2.11 where the knot at K determines the final amount of shortening. Schematic versions of the curves obtained from their quick-release experiment are shown in Figure 2.12.

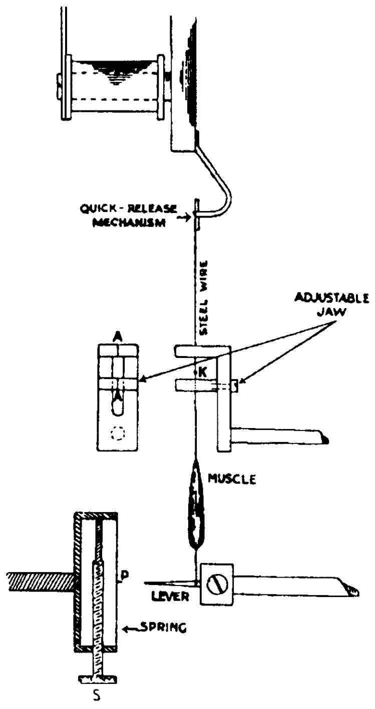


Figure 2.11: Gasser and Hill quick-release apparatus [GH24]

The observation that the measured muscle force did not instantaneously reach the level predicted by the new length indicated a damping effect within the contractile machinery. In the second experiment, a muscle was allowed to shorten at constant velocities, and a series of plots were made of the force required to produce that speed and the amount of shortening that was finally experienced. The area under these tension-shortening curves was then integrated to yield a relationship between work (force * distance) and velocity. The work-velocity curve has a decreasing slope, which is also consistent with the idea of a

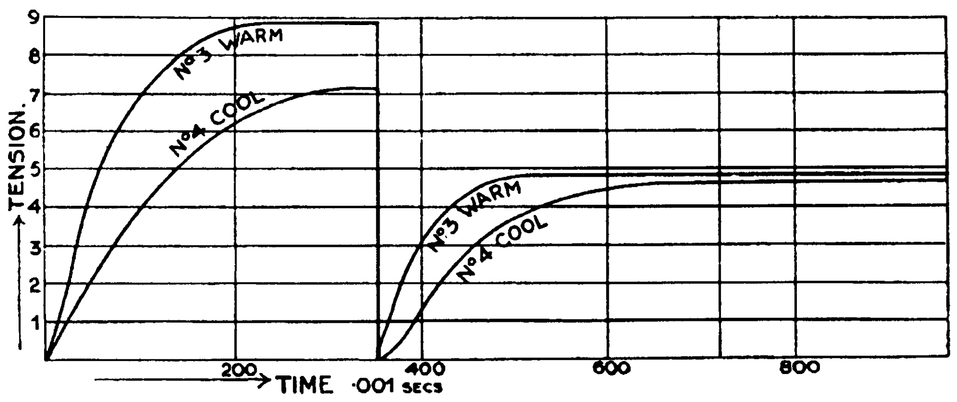


Figure 2.12: Quick-release curves, plotting tension versus time [GH24]

velocity-dependent force within the muscle. These curves are illustrated in Figure 2.13.

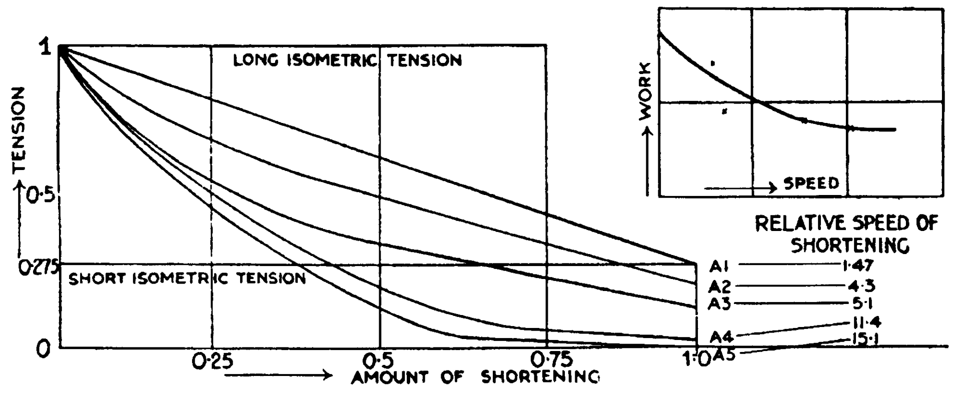


Figure 2.13: Muscles shortening at 5 constant speeds [GH24]

Quick-release experiments also play a role in determining the so-called *series elastic* element of a muscle. The series elastic component was originally proposed by Levin and Wyman in [LW27]. They built a device that allowed them to alternately stretch and release a muscle stimulated at a short rest length such that the passive parallel component does not enter the picture. The tension-length curves they obtained from trials at different speeds on the jaw muscle of a Dog-Fish is shown in Figure 2.14.

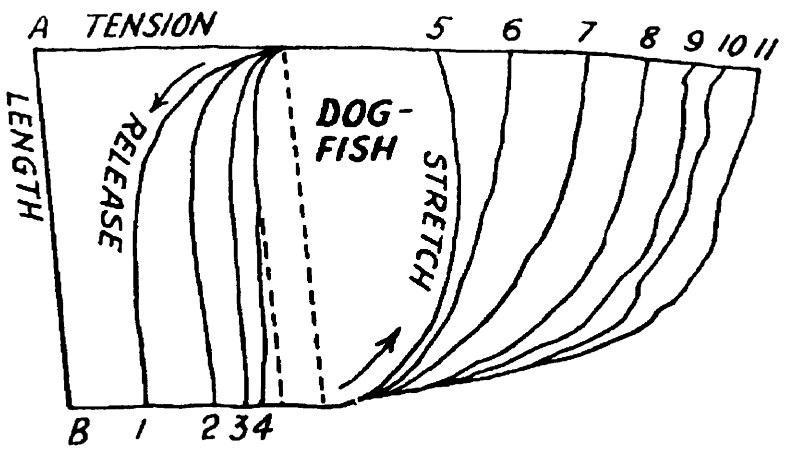


Figure 2.14: Tension-length curves from the jaw muscle of the Dog-Fish [LW27]

The fastest release is the curve on the far left, the quickest stretch is on the far right. If there were *no* series elastic component, the Dog-Fish plots would not have the exponential shape that is observed, rather, the lines of stretch and release would be straight. Levin and Wyman proposed the visco-elastic muscle model II on the right in Figure 2.15 to explain their findings. Their model is an extension of Hill's original visco-elastic model (I in the same figure) that included only the damping effects deduced from his measurements of the relationship between force and velocity.

Wilkie in [Wil56] made very direct measurements of the series elastic component with a quick-release type experiment. A schematic of his device is shown in Figure 2.16. The muscle is stimulated and develops force isometrically, stretching out its series elastic component. On release by the electromagnet at (e), the muscle is subjected to an isotonic load defined by the weight at (c). The stop at (d) insures that the load is applied only after release. The damping system at (f) removes mechanical vibrations in the lever system that would otherwise upset the muscle length measurements.

Wilkie found that immediately after the release, the previously stretched series elastic

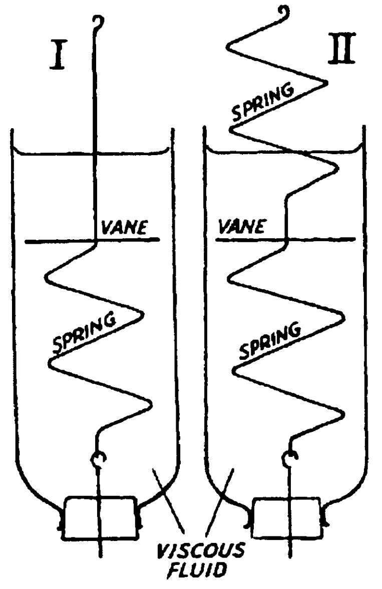


Figure 2.15: Visco-elastic muscle models [LW27]

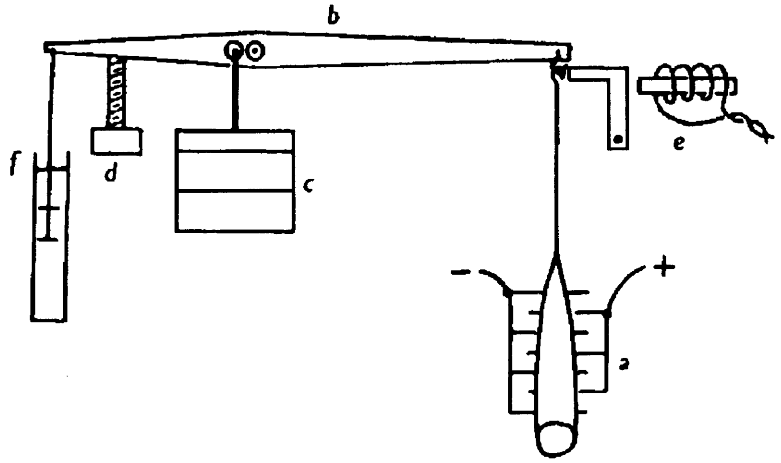


Figure 2.16: Wilkie's quick-release machinery [Wil56]

element shortens very rapidly to a length consistent with the isotonic load that is used. Then the contractile component shortens with a much lower velocity corresponding to the load. These two phases of shortening are shown in Figure 2.17. The initial perpendicular drop of length provides direct evidence for an elastic component in series with the contractile machinery of the muscle.

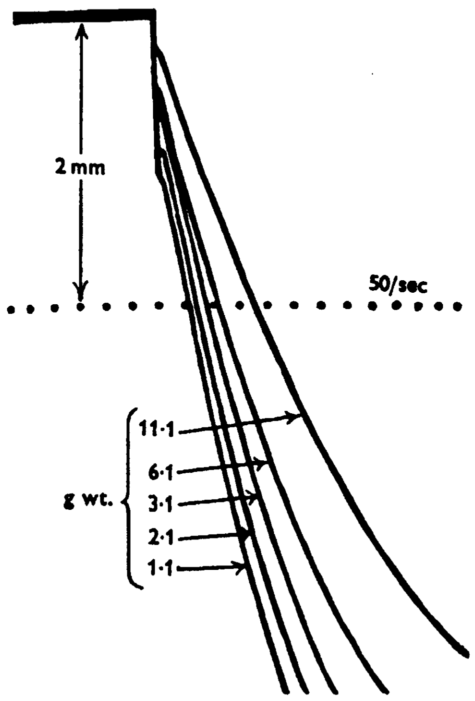


Figure 2.17: Shortening due to quick-release for various isotonic loads [Wil56]

A simple mechanical model of muscle that takes into account the effects described above is shown in Figure 2.18. This model is commonly attributed to A. V. Hill. The active state force T_0 is found, as discussed above, by subtracting the total force measured at different lengths for a stimulated muscle from that found to be due to passive effects alone. The notation $T_0(x_1, t)$ indicates that this force is a function of the muscle length and time-varying activation. The passive parallel stiffness K_{PE} has contributions due to the penetration of connective tissue through the muscle body resulting in the fasciculus divisions and also interfiber elasticity. The parallel damping component B is most likely due to the rate of the biochemical reactions that are responsible for contraction at the myofilament level. The series stiffness K_{SE} is primarily from the effect of tendon at the muscle attachment points, but is also probably partially due to the details of myofilament attachment within a sarcomere. The series element is very important in its ability to buffer the rapid change from inactive to active state and also provides a mechanical energy storage mechanism for the body in motion.

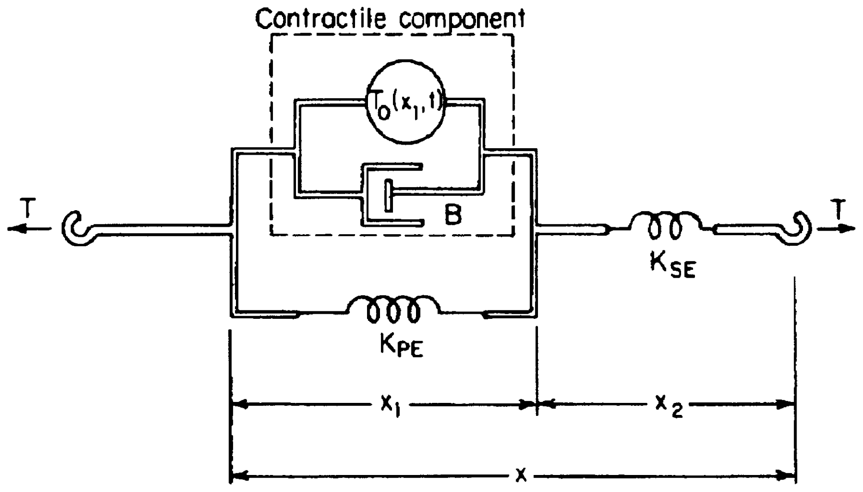


Figure 2.18: Hill's muscle model [McM84]

The Hill model while very simple, has proven enormously useful in practice in making calculations of the force generation of muscles working against different kinds of loads.

2.2.3 Zajac's Force Model

Zajac [ZTS86] [Zaj89] has developed a "dimensionless" lumped model of a *complete* musculotendon actuator that can be easily scaled to model particular whole muscles. Zajac's model is a refinement of the Hill model, and the normalized force curves that are presented directly reflect the non-linearities that result from the action of sliding filaments. The curves for the active and passive muscle force components are taken from measurements of single muscle fibers to ensure that tendon effects are not superimposed. Furthermore, pennation effects are directly included, while the series elastic element *not* associated with tendon is removed.

The isometric force generated in a particular actuator depends on one set of parameters that is considered constant over all actuators and another set that is musculotendon specific. The four specific parameters are,

α	pennation angle
F_0	maximum isometric force of active muscle
l_0^M	optimal muscle length at which F_0 is developed
l_s^T	tendon rest length

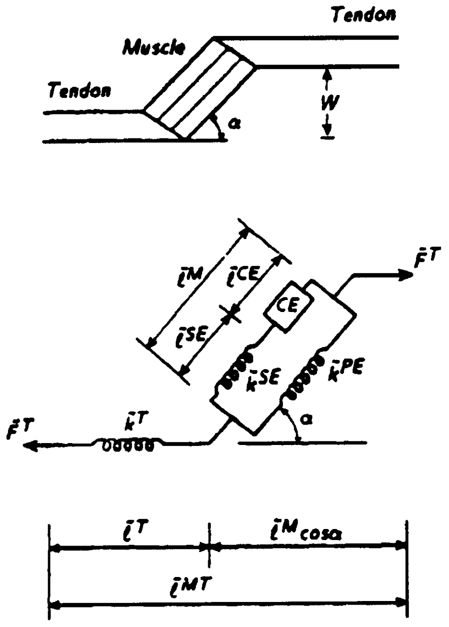


Figure 2.19: Musculotendon architecture [ZTS86]

The active muscle is represented in Figure 2.19 by the contractile element CE. Force developed by passive muscle is from $\tilde{k}^{PE}$ and is summed with the force from CE. The effect of the series elastic element $\tilde{k}^{SE}$ is lumped with the tendon model $\tilde{k}^{T}$. The dimensional units of interest are force and length. F_0 and l_0^M are the normalizing factors for these units. A *tilde* above a symbol denotes that it is a normalized quantity, for example,

$$\tilde{l}^M = \frac{l^M}{l_0^M} \quad \text{normalized muscle fiber length}$$

Other quantities and relationships used are,

l^{MT}	musculotendon length
l^T	tendon length
$l^{MT} = l^T + l^M \cos\alpha$	

Zajac gives the non-specific, dimensionless functions to model a musculotendon actuator in Figure 2.20 and Figure 2.21.

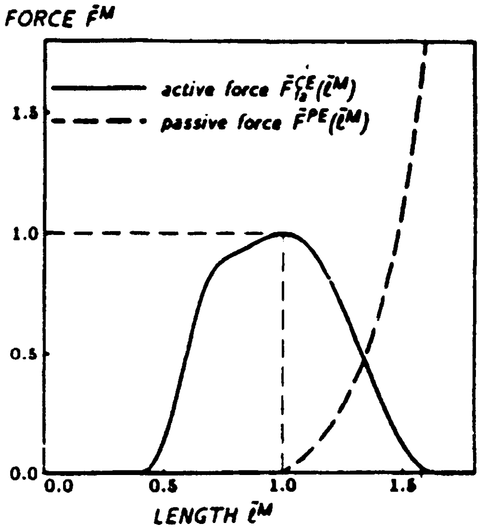


Figure 2.20: Active and passive muscle models. Normalized active muscle force $\tilde{F}_{fa}^{CE}(\tilde{l}^M)$ and passive force $\tilde{F}^{PE}(\tilde{l}^M)$ vs. normalized muscle fiber length $\tilde{l}^M$. [ZTS86]

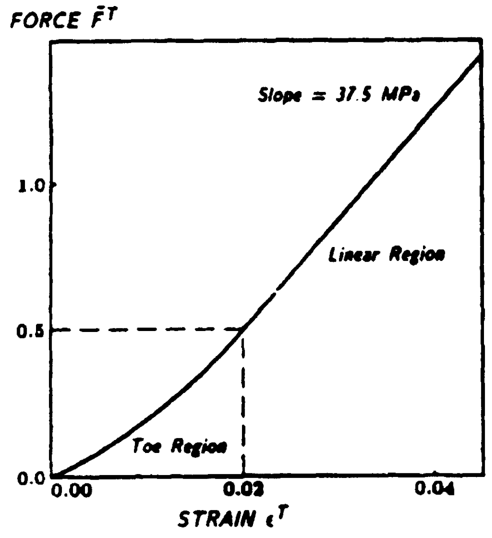


Figure 2.21: Tendon model. Normalized tendon force $\tilde{F}_T(\varepsilon^T)$ vs. tendon strain ε^T. [ZTS86]

The isometric muscle force functions can be written,

$$\tilde{F}_{iso}(\tilde{l}^M, t) = \tilde{F}_{fa}^{CE}(\tilde{l}^M)a(t)$$

$$\tilde{F}_M(\tilde{l}^M, t, \alpha) = (\tilde{F}_{iso}(\tilde{l}^M, t) + \tilde{F}^{PE}(\tilde{l}^M))\cos\alpha$$

where $a(t)$ is the time-varying normalized muscle activation function. For tendon, the normalized force is $\tilde{F}_T(\varepsilon^T)$ where ε^T is the tendon strain defined by,

$$\varepsilon^T = \frac{l^T - l^T_s}{l^T_s} = \frac{(l^{MT} - l^M\cos\alpha) - l^T_s}{l^T_s} = \frac{(l^{MT} - \tilde{l}^M l^M_0 \cos\alpha) - l^T_s}{l^T_s}$$

To use Zajac's muscle model, the functions $\tilde{F}_{fa}^{CE}(\tilde{l}^M)$, $\tilde{F}^{PE}(\tilde{l}^M)$ and $\tilde{F}_T(\varepsilon^T)$ and their derivatives need to be approximated. The active muscle function $\tilde{F}_{fa}^{CE}(\tilde{l}^M)$ is implemented as an interpolating cubic spline through twelve control points from Delp [Del90]. The derivative function $\frac{d\tilde{F}_{fa}^{CE}}{d\tilde{l}^M}$ is simply the derivative of the interpolating spline. The passive parallel force $\tilde{F}^{PE}(\tilde{l}^M)$ is estimated with the quadratic function,

$$\tilde{F}^{PE}(\tilde{l}^M) = 4(\tilde{l}^M - 1)^2 \quad \text{if } \tilde{l}^M \geq 1, \text{ else } 0$$

$$\frac{d\tilde{F}^{PE}}{d\tilde{l}^M} = 8\tilde{l}^M - 8 \quad \text{if } \tilde{l}^M \geq 1, \text{ else } 0$$

Plots of these two approximating functions are presented in Figure 2.22 for comparison with Zajac's curves. The tendon model $\tilde{F}_T(\varepsilon^T)$, is implemented as an order 50 Chebyshev

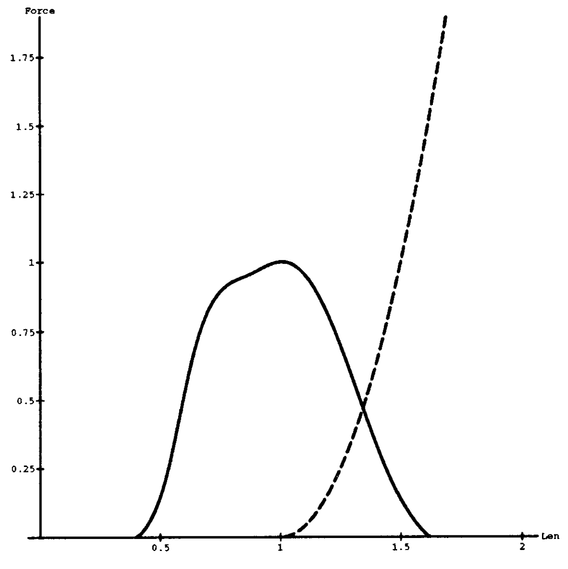


Figure 2.22: Solid line is spline approximated active force function. Dashed line is quadratically approximated passive component.

polynomial. Evaluating and finding the derivatives of such polynomial functions is very straightforward [PFTV88]. The shape of the Chebyshev approximated tendon force is plotted in Figure 2.23.

To characterize the dynamic properties of a musculotendon actuator, it is also necessary to consider the velocity dependent nature of the muscle forces. These damping forces are represented by the dashpot element B in Hill's model of the previous section. Zajac depicts the velocity-force relationship as in Figure 2.24 with the dimensionless form of the velocity $\tilde{v}_r^{CE}$. This muscle velocity is normalized with respect to the fiber rest length l_0^M and the maximum normalized velocity $\tilde{v}_m^{CE}$, which is a fifth parameter specific to modeling the dynamics of a particular muscle.

For our purposes, it is necessary to find force as a function of velocity, rather than

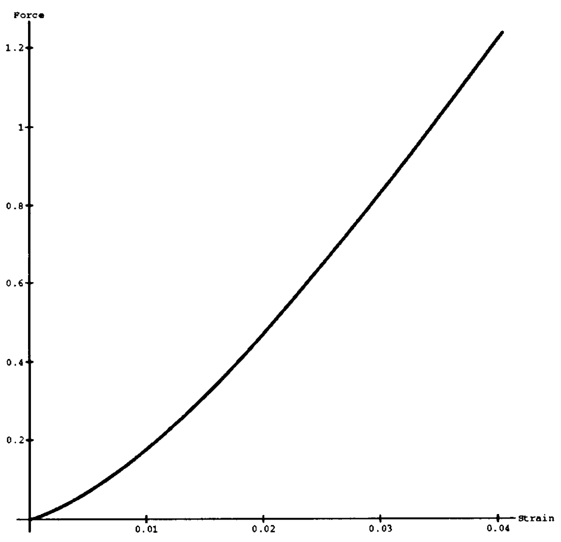


Figure 2.23: Chebyshev approximated tendon model.

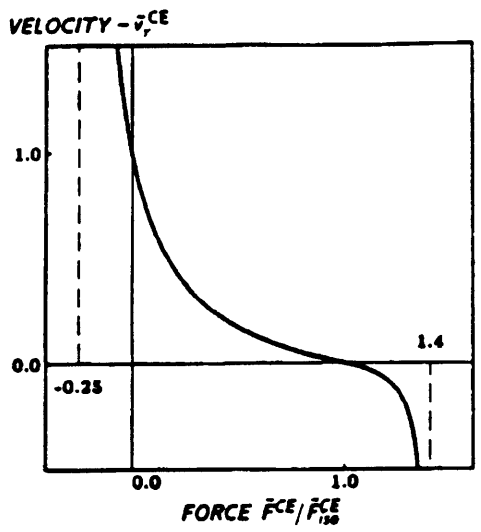


Figure 2.24: Velocity-force function of active muscle. Dimensionless velocity $\tilde{v}_r^{CE}$ vs. amount of normalized force relative to isometric force. [ZTS86]

the velocity as a function of force, as in [AD85]. Thus, Zajac's curve is turned on its side and approximated with,

$$f(\tilde{v}_r^{CE}) = \frac{1.65}{\pi}\left[\arctan(3.1 * (\tilde{v}_r^{CE} + .32774)) - \arctan(3.1 * .32774)\right] + 1$$

which is plotted in Figure 2.25. Negative velocity is for muscle contraction, while lengthening muscle will have positive velocity. The constants are set so that the normalized force will be 1 when the velocity is 0, the force will be 0 when the velocity is -1, and the asymptotes are as in Figure 2.24.

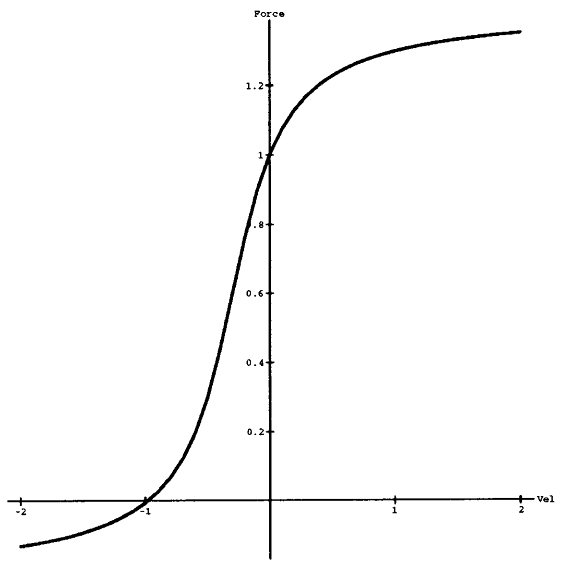


Figure 2.25: Force-velocity function approximated with arctan.

Finally, in the dynamic case, the total active force from the contractile element is seen to be a function of the activation $a(t)$, the force function $\tilde{F}_{fa}^{CE}(\tilde{l}^M)$, and the normalized muscle velocity $\tilde{v}_r^{CE}$,

$$\tilde{F}^{CE} = f(\tilde{v}_r^{CE})\tilde{F}_{iso}(\tilde{l}^M, t) = f(\tilde{v}_r^{CE})a(t)\tilde{F}_{fa}^{CE}(\tilde{l}^M)$$

and the force in the whole muscle is

$$\tilde{F}_M = (\tilde{F}^{CE} + \tilde{F}^{PE}(\tilde{l}^M)) \cos\alpha$$

As an example application of Zajac's model, lets calculate the maximum force that can be generated by an *isometric* musculotendon actuator in steady state. First, observe for this case that the force in the tendon should equal the force in the muscle and that velocity effects will not come into play, so

$$\tilde{F}_T(\varepsilon^T) - \tilde{F}_M(\tilde{l}^M) = 0$$

and writing the tendon strain in terms of $\tilde{l}^M$,

$$\tilde{F}_T\left(\frac{(l^{MT} - \tilde{l}^M l_0^M \cos\alpha) - l_s^T}{l_s^T}\right) - \tilde{F}_M(\tilde{l}^M) = G(\tilde{l}^M) = 0$$

Thus, to find the isometric muscle force, it is necessary only to find the zeros of $G(\tilde{l}^M)$. This is easily done using a one-dimensional root finder like Newton-Raphson [PFTV88] as long as $G(\tilde{l}^M)$ and the derivative $G'(\tilde{l}^M)$ can be evaluated for arbitrary normalized muscle lengths $\tilde{l}^M$. Writing for $G'(\tilde{l}^M)$,

$$G'(\tilde{l}^M) = \frac{d\tilde{F}_T}{d\varepsilon^T}\frac{d\varepsilon^T}{d\tilde{l}^M} - \left(\frac{d\tilde{F}_{iso}}{d\tilde{l}^M} + \frac{d\tilde{F}^{PE}}{d\tilde{l}^M}\right)\cos\alpha$$

And finally in terms of the dimensionless, consistent functions as implemented in Figure 2.22 and Figure 2.23

$$\begin{aligned}
G(\tilde{l}^M) &= \tilde{F}_T\left(\tfrac{-\tilde{l}^M l_0^M \cos\alpha}{l_s^T} + \tfrac{l^{MT}}{l_s^T} - 1\right) - \left(a(t)\tilde{F}_{fa}^{CE}(\tilde{l}^M) + \tilde{F}^{PE}(\tilde{l}^M)\right)\cos\alpha \\
G'(\tilde{l}^M) &= \tfrac{d\tilde{F}_T}{d\varepsilon^T}\left(\tfrac{-l_0^M \cos\alpha}{l_s^T}\right) - \left(a(t)\tfrac{d\tilde{F}_{fa}^{CE}}{d\tilde{l}^M} + \tfrac{d\tilde{F}^{PE}}{d\tilde{l}^M}\right)\cos\alpha
\end{aligned}$$

To illustrate the effect of the tendon slack length on the behavior of Zajac's model, the zeros of $G(\tilde{l}^M)$ were calculated as above for varying values of l_s^T. Plots are presented of normalized muscle force versus length for these different values of tendon rest length in Figure 2.26. The good correspondence between the curves of this figure and the ones presented in [ZTS86] (see Figure 2.27), we use as validation for our implementation.

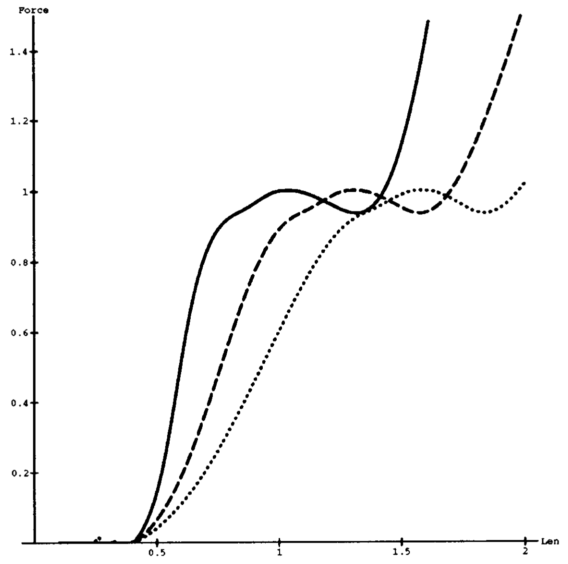


Figure 2.26: Isometric normalized muscle force vs. $l^{MT} - l_s^T$. Solid line is $l_s^T = .1$, dashed line is $l_s^T = 8$, and dotted line is $l_s^T = 16$.

2.3 Muscle Shape

This thesis will test the hypothesis that to make a good simulation of the changes in shape that a contracting muscle experiences, it is sufficient to characterize both the resting material and the changes in force known to be important to the contractile process. The primary benefit of this approach is that if the forces are calculated properly, then not only will it be possible to visualize a muscle in action, but that a valid biomechanical model will also be developed that can be used in further experimentation. For the purposes of visualization, however, it is important to obtain accurate geometric representations of the

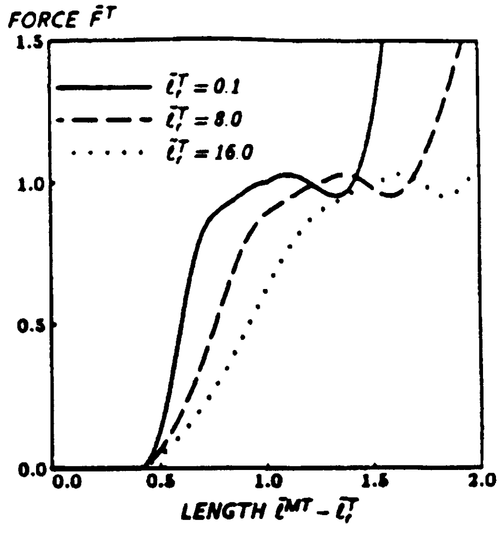


Figure 2.27: Isometric normalized muscle force vs. $l^{MT} - l_s^T$. [ZTS86]

rest shapes of muscle masses upon which the dynamic simulation techniques developed in the next chapter will be run. The emphasis of the thesis is on dynamics and not geometric modeling, so this section will serve only to point to sources from which input data has been obtained.

Because the interest here is primarily in computer graphics based applications, the goal in this section is to construct polygonal models that can be easily treated with standard rendering and animation routines. It will be shown in Section 3.2 how a *free-form deformation* based on twenty-node isoparametric cubes can be developed to govern both the dynamics formulation and the visualization process by defining a space that controls how the points of these polygonal models can be warped as the simulation demands that objects change shape. It will also be shown in Section 5.6 how these polygonal models will be used to define the initial geometries for finite element meshes made from isoparametric cubes.

Geometric models that have served the purposes of the dynamic simulation programs have been prepared from a variety of sources. The first experiments were made with range data of human heads from a Cyberware™ 3D digitizing system. Muscle rest shapes were constructed from both the Swivel 3D™ Professional [MhLH90] modeling program on the Macintosh and from contour stacks derived from magnetic resonance imaging (MRI) systems.

2.3.1 Range Camera Data

Range data such as that from the Cyberware camera is very simple to transform into a polyhedral representation. Two output files are produced by the scanning system. The first is an 8-bit *range* picture that represents a cylindrical arrangement of points from the source object, the second is a 24-bit *image* picture that can be used to texture-map color information back onto the scanned geometry. Two such data files from a human head[1] are shown in Figure 2.28 and Figure 2.29.

The Cartesian x, y location for each pixel in the range picture encodes the cylindrical θ and z coordinates for each sampled point, while the grayscale pixel value at each location is the radius r for that θ and z. The image picture is the RGB color value at each of these digitized points. The cylindrical samples are simply transformed into $\Re^3$ by

$$x = r\cos\theta; \qquad y = r\sin\theta; \qquad z = z$$

To convert range data into polyhedral form, the total number of points and polygons is printed, a point list is made from the range picture, and then connectivity information for each (quadrilateral) polygon is written out. This we call an OSU file after the polyhedral data standard created at Ohio State University. Finally, a subsidiary vertex color file is made from the 24-bit image picture. This is illustrated by a simple C language program,

[1]Thanks to David Sturman for the use of his head

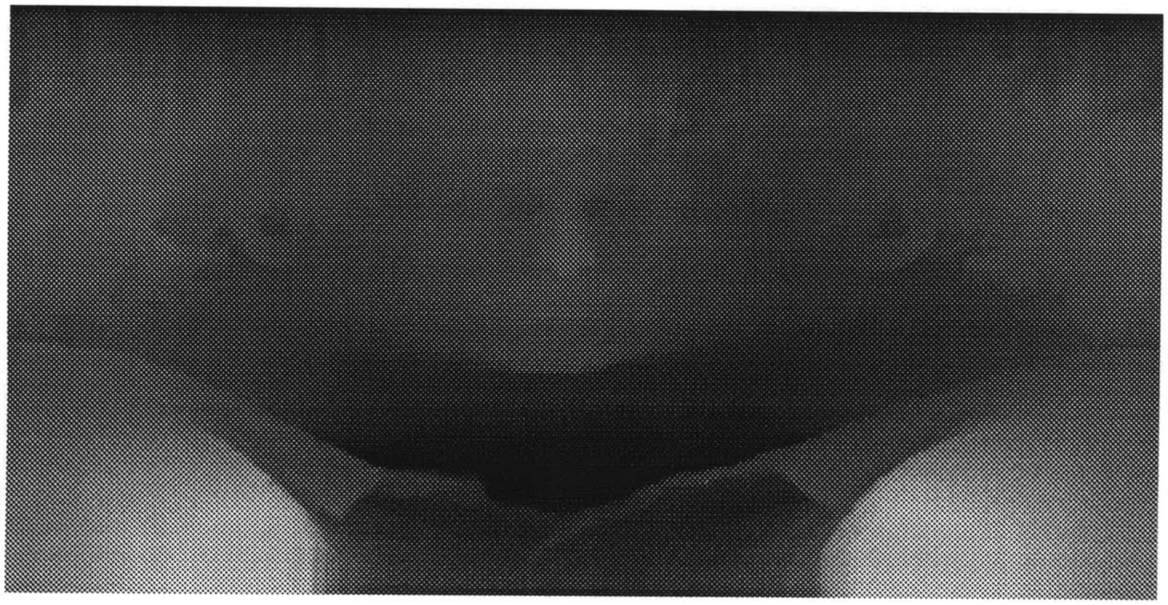


Figure 2.28: Range camera picture from Cyberware™ 3D system

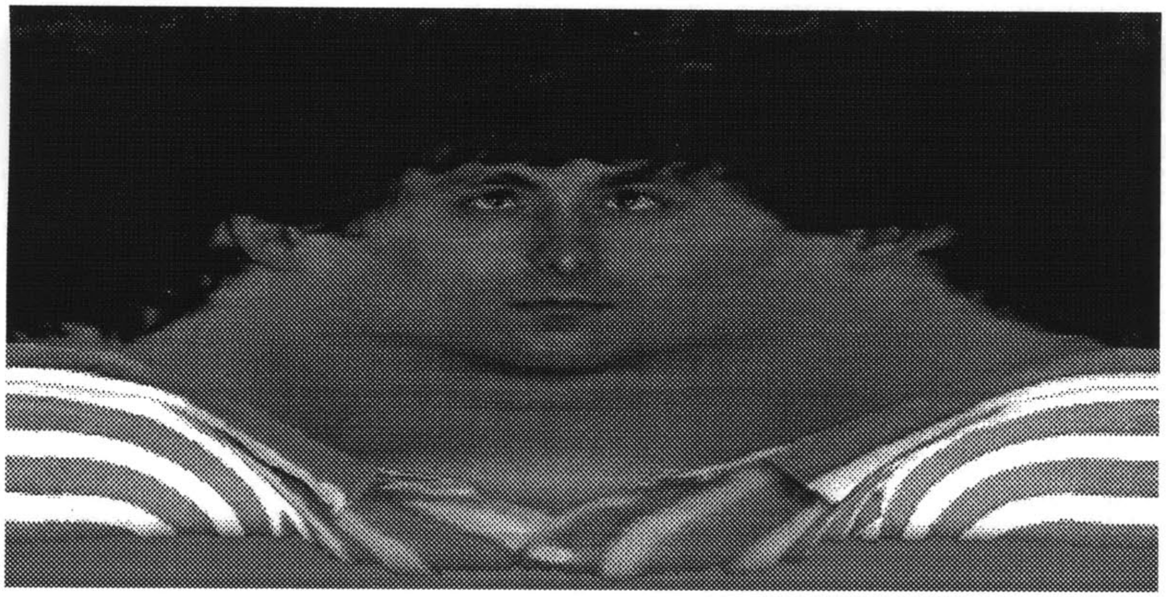


Figure 2.29: Point color picture

range2osu, that makes an OSU file from range data.

```
#include <stdio.h>
#include <math.h>

#define PI  (3.14159265359)

typedef unsigned char byte;
typedef byte PIXEL[3];

range2osu( range, color, width, height, gamma, osufp, vclfp )
byte *range;
PIXEL *color;
int width, height;
double gamma;
FILE *osufp, *vclfp;
{
int no_pts, no_polys;

   no_pts = height * width; no_polys = (height−1) * width;
   fprintf(osufp, "%d %d\n", no_pts, no_polys);

   range2pts( range, width, height, osufp );
   range2quads( range, width, height, osufp );
   range2vcl( color, width, height, gamma, vclfp );
   fflush(osufp); fflush(vclfp);
}
```

To generate the point list, **range2pts** first finds θ, z and r corresponding to pixel i, j. The space is arranged so that θ begins at 90° and increments positively in a full circle, r is in the range [0,1] and z is [-1,1] where $+z$ is up. The result is then transformed into $\Re^3$ and printed.

```
range2pts( range, width, height, osufp )
byte *range;
int width, height;
FILE *osufp;
{
int i, j;
double x, y, z, r, theta, thetainc;

   thetainc = 2 * PI / (double)width;
   for (i=0; i<height; i++)
   {  z = (−2. / (double)(height − 1) * i) + 1.;          /* Find z from current row indx */

      for (j=0, theta = PI/2.; j<width; j++, theta += thetainc)
```

```
            {  r = ((double)(range[i*width + j])) / 255.;        /* Find r from grayscale*/
               x = r * cos(theta);
               y = r * sin(theta);
               fprintf(osufp, "%g %g %g\n", (float)x, (float)y, (float)z);
            }
        }
    }
```

Polygons for the OSU file are tessellated into quadrilaterals by **range2quads**. The polygons are arranged along rows of points where the last polygon in each row is made to wrap around and join with the points from the first polygon in that row. This prevents any shading seams from the renderer.

```
range2quads( width, height, osufp )
int width, height;
FILE *osufp;
{
int i, j, k;

    for (i=0; i<height-1; i++)
    {  k = i*width;
       for (j=0; j<width-1; j++)
       {  k++; fprintf(osufp, "4 %d %d %d %d\n", k, k+1, k+width+1, k+width);
       }
       k++; fprintf(osufp, "4 %d %d %d %d\n", k, i*width+1, (i+1)*width + 1, k+width);
    }
}
```

The vertex color file is made from the 24-bit image picture associated with the range picture. Gamma control is given to set the contrast of the output vertex colors.

```
range2vcl( color, width, height, gamma, vclfp )
PIXEL *color;
int width, height;
double gamma;
FILE *vclfp;
{
int i, j;
double r, g, b;

    for (i=0; i<height; i++)
    {  for (j=0; j<width; j++)
       {  r = ((double)(color[i*width + j][0])) / 255.;
          g = ((double)(color[i*width + j][1])) / 255.;
          b = ((double)(color[i*width + j][2])) / 255.;
```

```
            gamma = 1./gamma;
            r = pow(r, gamma); g = pow(g, gamma); b = pow(b, gamma);
            r = clamp(r, 0., 1.); g = clamp(g, 0., 1.); b = clamp(b, 0., 1.);
            fprintf(vclfp, "%g %g %g\n", (float)r, (float)g, (float)b);
        }
    }
}
```

Typically, the data sets acquired from the Cyberware process are quite large, making full resolution models unwieldy. For example, a 512x256 size range picture will generate an OSU file with over 130,000 polygons. To make lower resolution, more conveniently sized geometric models, a downsampling operation is first performed to the range and color images, then OSU files are made using the **range2osu** program on the resulting smaller pictures. Downsampling in the image-space is much simpler than trying to do the equivalent filtering in world-space in order to make models with fewer polygons. It should be noted that with appropriate filtering, very nice lo-res geometric models can be achieved. In fact, for this kind of data many 2D paint and filtering operations have been found to be very effective [Wil90]. The Adobe Photoshop™ paint program on the Macintosh has proven invaluable in performing the operations mentioned above, as well as allowing interactive repair of noisy range data.

2.3.2 Manual Shape Input

Not surprisingly, often the easiest way to make a computer graphics object is to use an interactive program such as Swivel 3D™ Professional that can create 3D geometries through extrusion, lathing, and skinning operations[2]. The first muscle object that was created in the course of the thesis work was a human biceps from an anatomically accurate plastic model. Points were digitized from the plastic model for both TOP and SIDE views. The biceps was then "lathed" using a circular cross section. The resulting computer graphics

[2]Thanks to Steve Drucker for writing the program to convert Swivel output files into OSU format

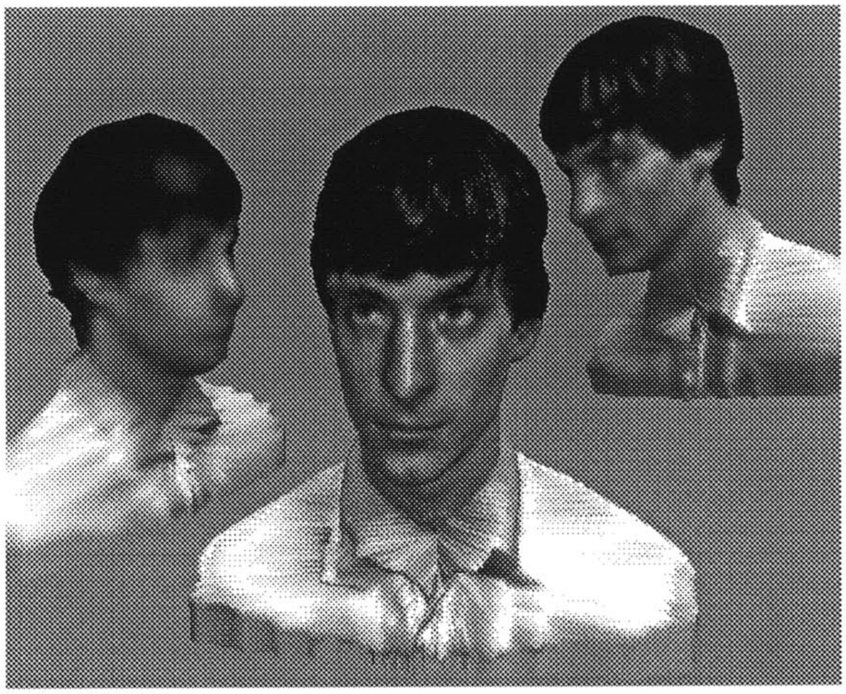


Figure 2.30: Reconstructed "Sturman" heads with 1984, 32512 and 8064 polygons

model is shown in Figure 2.31.

Since much of the experimental work involving muscle biomechanics is done using frog muscles, the *gastrocnemius* muscle from a prepared frog was digitized for use in the computer simulated biomechanical experiments discussed in Section 6.1. Some time was spent observing Dr. Simon Gitzer at MIT in the "frog lab" making force-length measurements from the gastrocnemius of an actual animal. After these were done, the muscle was fully dissected and measured for input into Swivel. The procedure used was much like that for the plastic biceps, and a lathed object was created. This is shown in Figure 2.32.

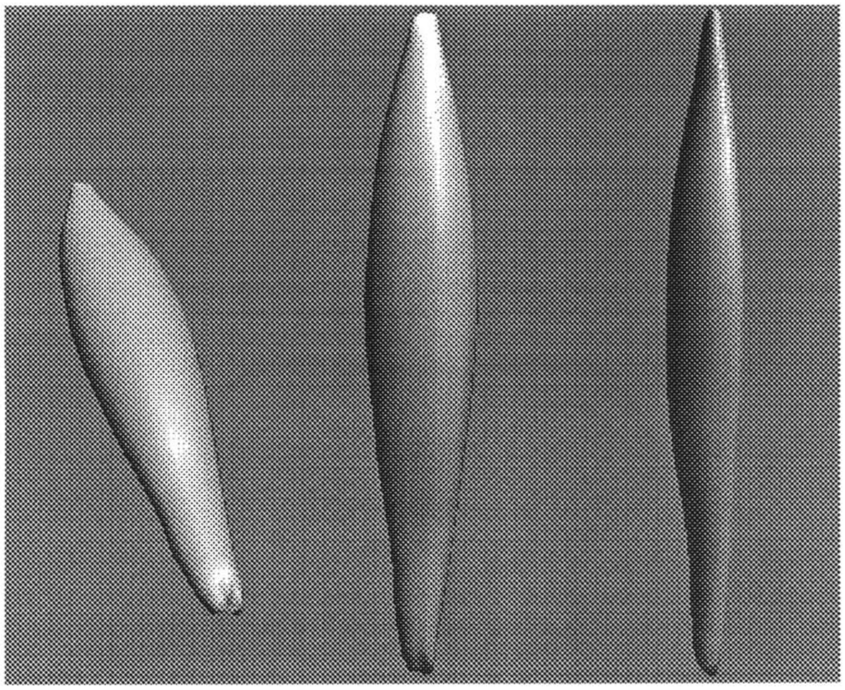


Figure 2.31: Human biceps, 342 polygons

2.3.3 Contour Data

Imaging from CAT and MRI scanning systems is a relatively new but very important source of clinical data. These systems acquire three-dimensional objects as a series of 2D slices arranged along an axis in space. The "skinning" facility in Swivel can be used to operate on data sets of this kind, where anatomical forms are defined by varying the shape of the cross section along the length of the body. More sophisticated techniques such as the *marching cubes* algorithm [LC87] can automatically create polygonal models of constant density surfaces from 3D data arrays of this type.

One of the practical drawbacks of CAT and MRI systems is that they are very expensive and so access to them is limited. For this reason, an effort was made by Dr. Gitzer to reconstruct a set of frog musculature by, in essence, making the stack of contour informa-

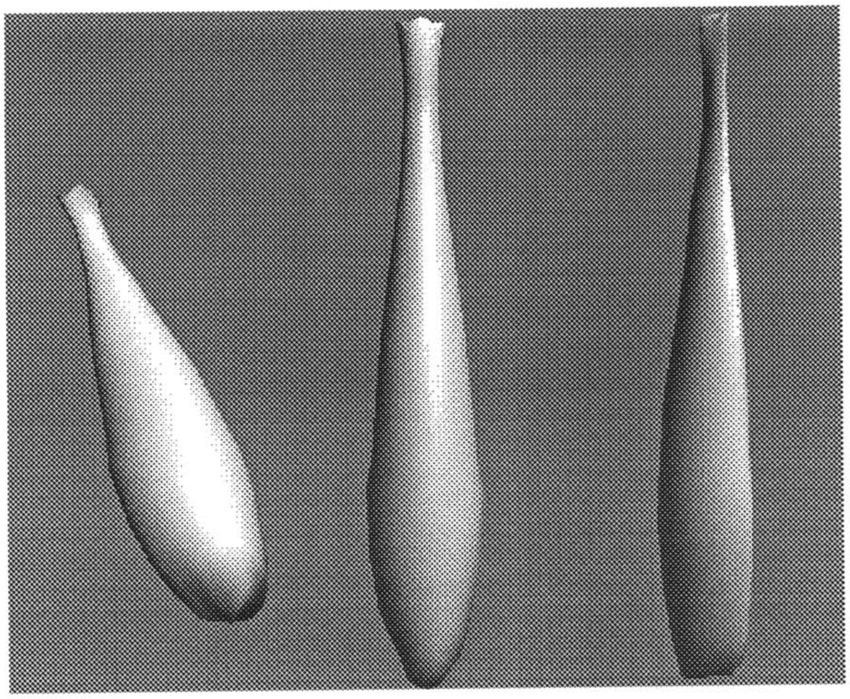


Figure 2.32: Frog gastrocnemius, 576 polygons

tion by hand. A frog leg was embedded in epoxy and sliced into two-dimensional sections. Slides were made of these sections and enlargements made for digitization as a precursor to making a 3D model. The slides were scanned into the computer and the individual muscles differentiated by flood-filling their regions with different grayscale values. Figure 2.33 shows one such slice.

The idea behind trying to digitize the frog musculature as described is to obtain data for a whole musculoskeletal system, with the muscles and bones in their correct relative *in vivo* positions. Many interesting problems are introduced by considering the whole system. Forces must be calculated for modeling contact between bone, muscles and skin. A simulated skeleton can be driven by muscle forces calculated at the tendon attachment points. Higher-level coordination and reflexes can be modeled. Chapter 4 will touch upon some of these topics.

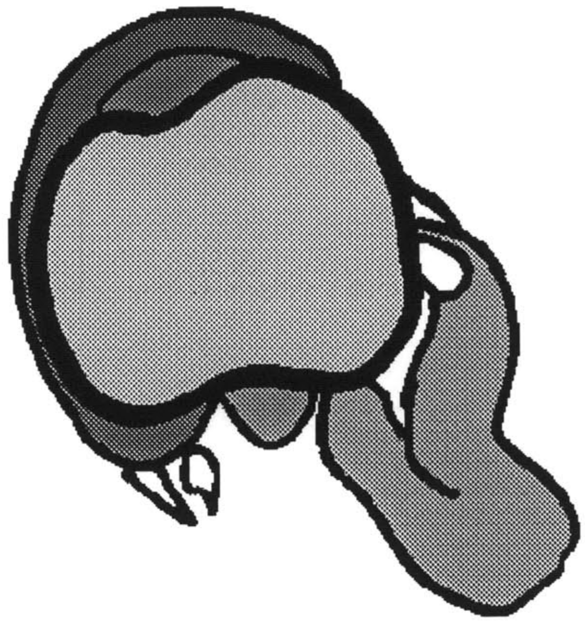


Figure 2.33: Hand-derived slice through frog leg

The effort with the frog leg was subsequently put on hold when a polygonal data set from an entire human left calf became available through Dr. Alan Garfinkel at UCLA. The source for the calf model was a long sequence of MRI scans, that were carefully hand segmented into the individual, anatomical muscle masses and then "skinned" into triangles. Ten muscles, including the medial-gastrocnemius, the lateral-gastrocnemius, and the soleus, and one muscle group make up the data. The tibia and fibula bones are also included. Figure 2.34, from Dr. Garfinkel shows how the MRI scans were anatomically carved up. The image is shown with its colormap inverted for clarity.

Because the leg data was received as files of triangle meshes, no further processing had to be done to ready them for simulation or display. Figure 2.35 shows three different views of the reconstructed leg. The entire data set is shown in the middle view. In the left view, the overlying gastrocnemius muscles and the soleus are removed to show the underlying

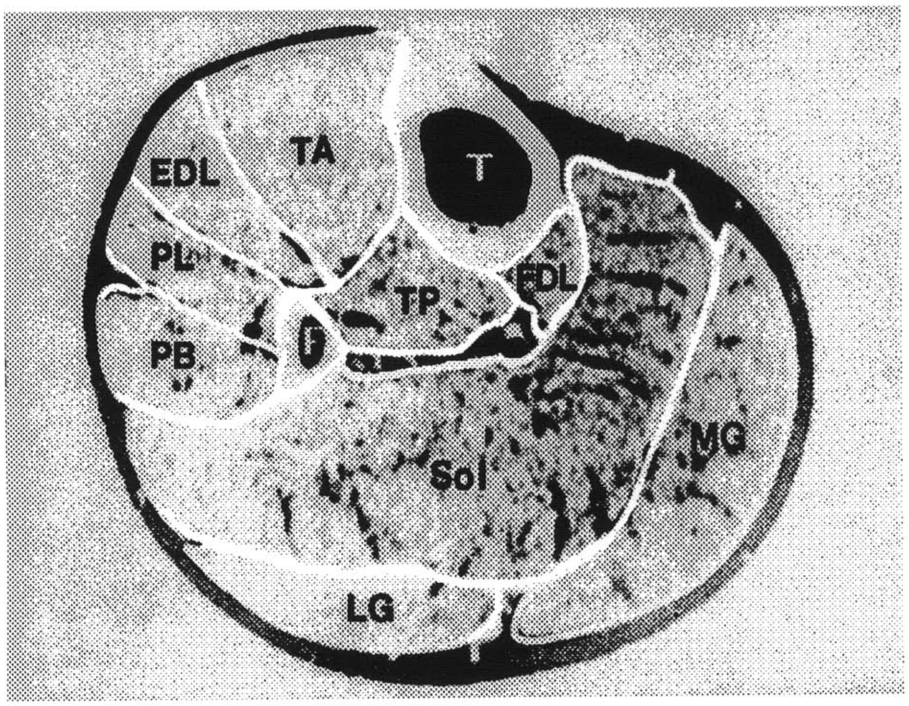


Figure 2.34: MRI slice through human calf

structures, and the tibia and fibula are shown on the right.

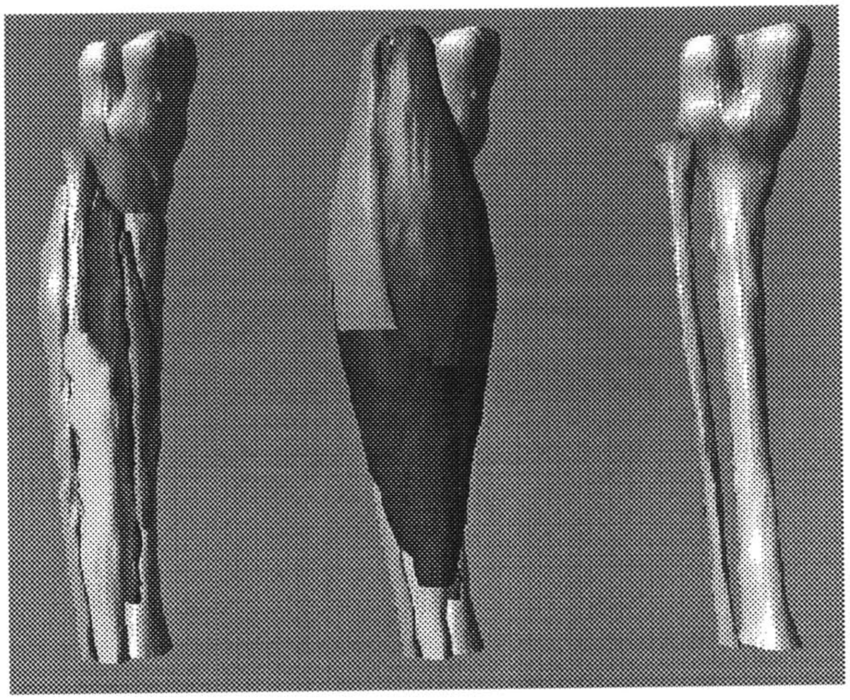


Figure 2.35: Reconstructed legs

The experiments done in Section 6.1.4 using this data will be centered around the medial-gastrocnemius. This muscle was chosen both because of its large size, and because it is on the outside, closest to the skin. Thus it should play a large part in determining the shape of the whole leg. Figure 2.36 shows three views of the medial-gastrocnemius.

2.4 Simulating a Muscle

This section has discussed two different aspects of muscle that are essential to making a simulation —the first topic was muscle *models*, the second concerned the acquisition of muscle *data*. We have shown how the Hill model was developed and discussed how Zajac's model is a refinement. We have also described an implementation of the Zajac model. Our

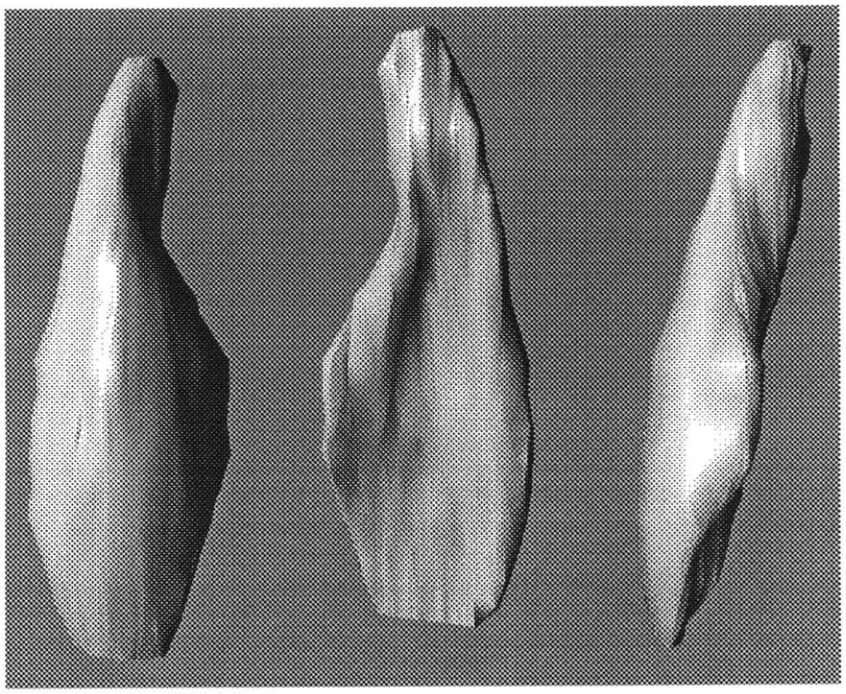


Figure 2.36: Medial-gastrocnemius

data sources are Cyberware scans, hand-made slices, MRI slices and direct geometric output from Swivel. All these are turned into a standard polyhedral representation for display.

To simulate the action of muscle for a computer graphics application, the model and data must be synthesized, and for this the finite element method is used (see Figure 2.37). The FEM will be the vehicle for our muscle model. The polyhedral data is used to define meshes of finite elements. Dynamic equilibrium equations are derived from the mesh. Zajac's model is used to apply forces to the mesh node points. The FEM model will then be dynamically simulated forwards and the mesh will automatically deform in response. A free-form deformation defined by the mesh will help us visualize the resulting changes in shape due to the contraction.

There are many things known about muscle that a computationally based muscle should be able to predict. For the model to be interesting biomechanically, we should be

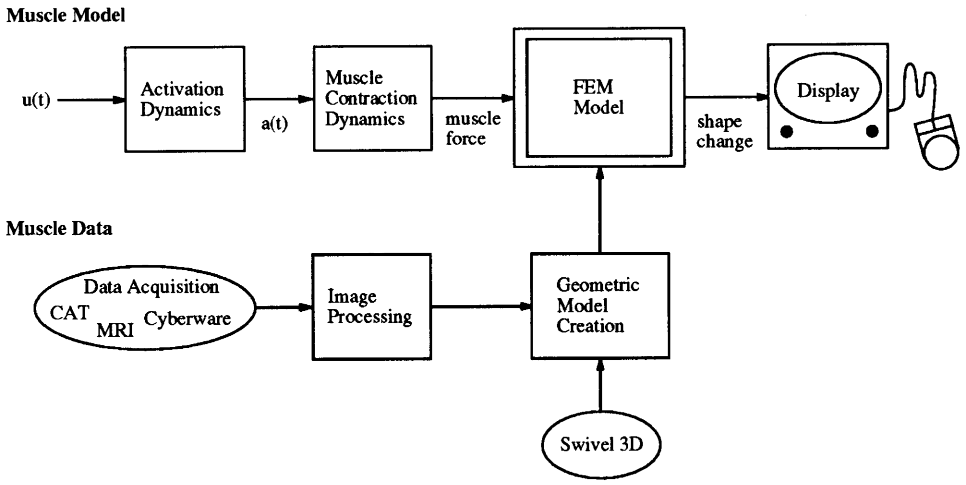


Figure 2.37: Muscle model and muscle data *after* [Zaj89] *and* [LC87]

able to the calculate the timevarying course of the force that a muscle generates at different lengths, and with different tendon rest lengths. The model should also be able to predict the effect of velocity dependent forces. Thus, as a way of validating the final implementation, an effort will be made to reproduce some of the experiments on whole muscle that lead to the development of Hill's biomechanical model.

Besides generating force to move the bones, a 3D model of muscle should conserve volume as its shape changes. Because of the enclosing tendon sheath, muscle preserves its volume as it elongates and contracts. In fact, one of the earliest recorded experiments performed on a single prepared muscle demonstrated this effect [McM84]. Jan Swammerdam performed the experiment illustrated in Figure 2.38 on a muscle removed from a frog in the early 1600's. Later, the constant volume finding was extended to human muscles by Glisson who had subjects place their arms in a water filled tube sealed at the elbow.

The experiment of Jan Swammerdam, circa 1663, showing that a muscle does not increase in volume as it contracts. A frog's muscle (*b*) is placed in an air-filled tube closed at the bottom (*a*). When the fine wire (*c*) is pulled, the nerve is pinched against the support (*d*), causing the muscle to contract. The drop of water in the capillary tube (*e*) does not move up when the muscle contracts. From Needham (1971).

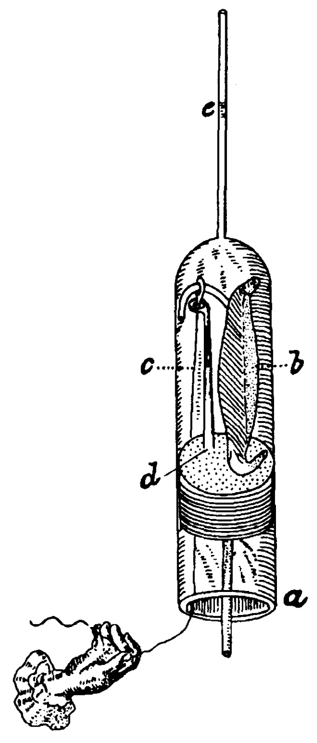


Figure 2.38: Muscles contract with constant volume. [McM84]

Chapter 3

The Finite Element Method

The displacement based finite element method (FEM) has been widely used to model elastic, deformable materials in engineering analysis. It is particularly well suited to a computer implementation and can be used effectively to model real world physical situations with real world boundary conditions and forces. For the purposes of simulating a muscle, the approach will be to formulate the dynamics of a volume preserving elastic object with material properties to be found in the literature. The rest shape for a muscle will be determined from either hand derived or machine scanned histological data. The important input forces will come from Zajac's biomechanical model discussed in the previous section. When considering a musculoskeletal system, other forces need also be considered. These include forces from the fascia sheath drawing the muscle to the bone and muscle-bone and muscle-muscle contact.

To complete the thesis project, a system that performs FEM dynamics on an HP graphics workstation based on the *modal* formulation has been implemented. The rest of this chapter will discuss the finite element foundations for the resulting computer program and point out the tradeoffs that are made to make solving the dynamics of deformation tractable for this hardware platform.

For computer animation, the steps taken to perform dynamic simulations of elastic objects using the finite element are,

1. Begin with polyhedral model
2. Interactively specify FEM mesh
3. Solve for equilibrium equations, do modal transformation, etc.
4. Simulate dynamics by calculating forces and finding mesh displacements
5. Warp points of polyhedral model into deformed mesh space
6. Render new shape

The polyhedral model represents the shape of the elastic body to be simulated and is considered the input *data.* A FEM mesh is constructed that approximates the volume of the data with some number of finite elements. From the FEM mesh and the material properties of the object under investigation, dynamic equilibrium equations are found, the modal transformation is performed, and other *preprocessing* steps are taken. A standard numerical differential equation solver is then used to advance the state of the mesh one time step forwards. This results in displacements for the nodal points of the FEM mesh that are then used to warp the points of the original polyhedral model with a free-form deformation technique. The new shape is then rendered and the next simulation step taken. While steps 1-3 are considered part of a preprocessing procedure, steps 4-6 happen over and over throughout an interactive simulation application and is called the *simulation loop.*

Much of the discussion to follow concerning the finite element method is from Bathe [Bat82] and is presented here as a way of introducing the FEM to computer graphics researchers. Because one of our goals is to use the FEM within interactive graphics applications, it was decided to implement a custom finite element package that could be tightly coupled to graphics output, rather than using a commercially available system. An emphasis carried throughout our implementation is to speed up the steps of the simulation loop whenever possible. One of the key technical contributions of the thesis is to define the free-form deformation of step 5 that "links" the FEM with computer graphics output, in Section 3.2.2.

3.1 Equilibrium Equations

To model the elastic properties of deformable objects, sets of differential equations can be defined that govern the dynamics of their deformation. There are a variety of techniques for deriving these equations—Terzopolis and Fleisher discuss in [TF88] a method based on an analysis of the energy of deformation for an elastic material, while Pentland and Williams in [PW89] discuss the method of finite elements. This thesis models elastic objects as collections of twenty-node isoparametric brick elements. For a body having n nodal points, the finite element equations have the form

$$M\ddot{u} + C\dot{u} + Ku = R$$

Where u is an $3n$ vector of nodal displacements, M, C and K are $3n$ x $3n$ matrices describing the mass, damping and stiffness between points within the body, and R is a $3n$ vector of forces applied to each node.

3.1.1 Stiffness Matrix K

The stiffness matrix K can be found by choosing a specific interpolating function for displacement throughout the body and using a linear approximation for elasticity. Though twenty-node isoparametric elements will be used in the subsequent muscle analysis, the discussion that follows is based on a four-node tetrahedron element for simplicity. It will be shown how the stiffness matrix for a twenty-node element can be found in a similar way. Following a development for linear planar elements from [Roc83], K is derived for the special case of the tetrahedral finite element in Figure 3.1 as follows.

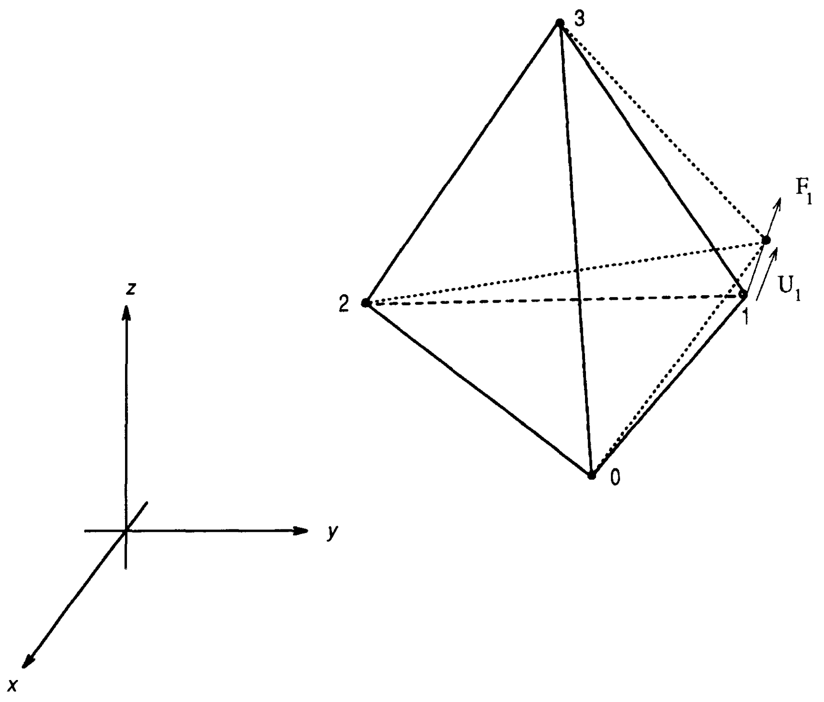


Figure 3.1: Four node tetrahedron element

At each of the four nodes of the tetrahedron, define a three-vector for the displacement,

$$\{U\} = \begin{Bmatrix} u \\ v \\ w \end{Bmatrix}$$

and for the force.

$$\{F\} = \begin{Bmatrix} F_x \\ F_y \\ F_z \end{Bmatrix}$$

The tetrahedron has three displacement degrees of freedom for each of the four nodes, for a total of twelve DOFs. For the tetrahedral element these are,

$$\{U^e\} = \left\{ \begin{array}{cccc} U_0 & U_1 & U_2 & U_3 \end{array} \right\}^T = \left\{ \begin{array}{ccccccc} u_0 & v_0 & w_0 & u_1 & v_1 & w_1 & \cdots \end{array} \right\}^T$$

and the nodal forces acting on the element are written,

$$\{F^e\} = \left\{ \begin{array}{cccc} F_0 & F_1 & F_2 & F_3 \end{array} \right\}^T = \left\{ \begin{array}{ccccccc} F_{x_0} & F_{y_0} & F_{z_0} & F_{x_1} & F_{y_1} & F_{z_1} & \cdots \end{array} \right\}^T$$

Now, choose a displacement function $f(x, y, z)$ that defines the displacement $U(x, y, z)$ at any point in the element. For the tetrahedron it is natural to choose the linear functions,

$$\begin{aligned} u(x,y,z) &= \alpha_1 + \alpha_2 x + \alpha_3 y + \alpha_4 z \\ v(x,y,z) &= \alpha_5 + \alpha_6 x + \alpha_7 y + \alpha_8 z \\ w(x,y,z) &= \alpha_9 + \alpha_{10} x + \alpha_{11} y + \alpha_{12} z \end{aligned}$$

So the internal displacement is,

$$\begin{aligned} \{U(x,y,z)\} &= \left\{ \begin{array}{c} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{array} \right\} \\ &= \left[\begin{array}{cccccccccccc} 1 & x & y & z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & x & y & z & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x & y & z \end{array} \right] \left[\begin{array}{c} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_{12} \end{array} \right] \end{aligned}$$

or more compactly,

$$\{U(x,y,z)\} = [f(x,y,z)]\{\alpha\} \tag{3.1}$$

The next step is to express the state of displacement $U(x, y, z)$ within the element in terms of the nodal displacements $\{U^e\}$. Looking at node 1,

$$\{U_1\} = \{U(x_1, y_1, z_1)\} = [f(x_1, y_1, z_1)]\{\alpha\}$$

or,

$$\{U_1\} = \begin{bmatrix} 1 & x_1 & y_1 & z_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & x_1 & y_1 & z_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x_1 & y_1 & z_1 \end{bmatrix} \{\alpha\}$$

For the whole element,

$$\{U^e\} = \left\{ \begin{array}{c} \{U_0\} \\ \{U_1\} \\ \{U_2\} \\ \{U_3\} \end{array} \right\} = \begin{bmatrix} [f(x_0, y_0, z_0)] \\ [f(x_1, y_1, z_1)] \\ [f(x_2, y_2, z_2)] \\ [f(x_3, y_3, z_3)] \end{bmatrix} \{\alpha\} = [A]\{\alpha\}$$

Finally, solving for the vector α,

$$\{\alpha\} = [A]^{-1}\{U^e\} \tag{3.2}$$

And substituting for $\{\alpha\}$ in Equation 3.1

$$\{U(x, y, z)\} = [f(x, y, z)][A]^{-1}\{U^e\} = [H]\{U^e\}$$

Now, lets relate the strains $\{\varepsilon(x, y, z)\}$ at any point within the tetrahedral element to the displacements $\{U(x, y, z)\}$, and hence to the nodal displacements $\{U^e\}$. The standard first-order approximation to the strain-displacement relationship is, from elasticity theory,

$$\{\varepsilon(x,y,z)\} = \left\{\begin{array}{c} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{array}\right\} = \left\{\begin{array}{c} \partial u/\partial x \\ \partial v/\partial y \\ \partial w/\partial z \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \end{array}\right\}$$

Refering back to the linear interpolating functions, the strains can be written,

$$\begin{aligned} \varepsilon_x &= \alpha_2 \\ \varepsilon_y &= \alpha_7 \\ \varepsilon_z &= \alpha_{12} \\ \gamma_{xy} &= \alpha_3 + \alpha_6 \\ \gamma_{yz} &= \alpha_8 + \alpha_{11} \\ \gamma_{zx} &= \alpha_{10} + \alpha_4 \end{aligned}$$

Or in matrix form,

$$\{\varepsilon(x,y,z)\} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \left\{\begin{array}{c} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \vdots \\ \alpha_{12} \end{array}\right\} = [D]\{\alpha\}$$

And making the substitution for $\{\alpha\}$ from Equation 3.2,

$$\{\varepsilon(x,y,z)\} = [D][A]^{-1}\{U^e\} = [B]\{U^e\}$$

The next step in solving for the tetrahedron's stiffness matrix is relating the internal stresses $\{\sigma(x,y,z)\}$ to strains $\{\varepsilon(x,y,z)\}$ and to the nodal displacements $\{U^e\}$. For the special case of a linear, isotropic, elastic material,

$$\{\sigma(x,y,z)\} = [\mathcal{C}]\{\varepsilon(x,y,z)\} \tag{3.3}$$

$$[\mathcal{C}] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ & & 1 & 0 & 0 & 0 \\ & & & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ & & & & \frac{1-2\nu}{2(1-\nu)} & 0 \\ & & & & & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix}$$

where

$\mathcal{C}$	is a symmetric matrix
E	is Young's modulus of elasticity
ν	is Poisson's ratio

By replacing the internal stresses $\{\sigma(x,y,z)\}$ with statically equivalent nodal forces $\{F^e\}$, nodal forces can be related to nodal displacements $\{U^e\}$ to obtain the element stiffness matrix K. This is a well known application of the *principle of virtual work.*

Lets choose an arbitrary set of nodal displacements

$$\{U^{*e}\} = \begin{Bmatrix} \{U_0^{*e}\} \\ \{U_1^{*e}\} \\ \{U_2^{*e}\} \\ \{U_3^{*e}\} \end{Bmatrix}$$

The total external work represented by this displacement is given by

$$W_{ext} = \{U_0^{*e}\}\{F_0^e\} + \cdots + \{U_3^{*e}\}\{F_3^e\} = \{U^{*e}\}^T\{F^e\}$$

If the chosen arbitrary displacements cause a strain $\{\varepsilon(x,y,z)^*\}$, the internal work is

$$W_{int} = \{\varepsilon(x,y,z)^*\}^T\{\sigma(x,y,z)\}$$

Integrating through the element volume to get the total internal work,

$$\int_V W_{int} dV = \int_V \{\varepsilon(x,y,z)^*\}^T\{\sigma(x,y,z)\} dV$$

From the relationships derived above it is known

$$\begin{aligned}\{\varepsilon(x,y,z)^*\} &= [B]\{U^{*e}\}\\ \{\sigma(x,y,z)\} &= [C][B]\{U^e\}\end{aligned}$$

And substituting into the internal work integral

$$\int_V W_{int} dV = \int_V \{U^{*e}\}^T[B]^T[C][B]\{U\} dV$$

Finally, equating the total internal and external work, and dividing through by $\{U^{*e}\}^T$,

$$\begin{aligned}\{F^e\} &= \left[\int_V [B]^T[C][B] dV\right]\{U\}\\ &= [K]\{U^e\}\end{aligned}$$

Thus, the stiffness matrix for a single tetrahedral element, is the integral

$$[K] = \int_V [B]^T[C][B] dV$$

Where B is the nodal-displacement to internal-strain matrix and C is the strain to stress (constitutive) matrix. If the material comprising the tetrahedron is homogeneous, the matrices under the integral will be constant and so,

$$[K] = (\text{volume of tetrahedron}) * [B]^T[C][B]$$

3.1.2 Mass Matrix M

While the derivation for the stiffness matrix is relatively involved, the mass matrix M is often represented by a simple diagonal matrix. Physically this corresponds to the total mass of the element being evenly concentrated at each of the nodes. The approximation inherent in a lumped mass model tends to lessen as more and more elements are used to represent a given physical system. However, because our FEM implementation is geared towards workstation size computers, the tendency will be towards using as few elements as possible. Thus, the *consistent* mass matrix is used rather than the diagonal lumped mass matrix.

The consistent mass matrix is found in a way much like the stiffness matrix. First, note that the internal force per unit volume due to the acceleration of gravity is

$$F_{gravity}(x, y, z) = \rho[H]\{\ddot{U}^e\}$$

where ρ is the (usually constant) mass density, $\ddot{U}^e$ is the acceleration at each node, and $[H]\{\ddot{U}^e\}$ is the internal acceleration within the element at point x, y, z. Applying the principle of virtual work as in the derivation of K, the total external work can be written

$$W_{ext} = \{U^{*e}\}^T\{F^e_{gravity}\}$$

And integrating for the total internal work

$$\int_V W_{int} dV = \int_V \{U^{*e}\}^T[H]^T\rho[H]\{\ddot{U}^e\}dV$$

Equating the total internal and external work functions and dividing through by $\{U^{*e}\}^T$

$$\begin{aligned}\{F^e_{gravity}\} &= \left[\int_V [H]^T\rho[H]dV\right]\{\ddot{U}^e\} \\ &= [M]\{\ddot{U}^e\}\end{aligned}$$

And so the mass matrix for a single element is the integral

$$[M] = \int_V \rho[H]^T[H]dV$$

A discussion concerning the damping matrix C will be put off for Section 3.4 covering *modal analysis*.

3.2 Isoparametric Interpolation

3.2.1 Stiffness Matrix

In deriving K for the four-node tetrahedron, the element displacements $u(x,y,z)$, $v(x,y,z)$, and $w(x,y,z)$ were written as linear functions of x, y, and z with undetermined coefficients $\{\alpha\}$. The matrix A was then found from the nodal positions and the matrix H calculated from A^{-1} to express the state of *internal* displacement within the tetrahedral element as a function of *nodal* point displacement. The key to the isoparametric finite element formulation is to establish a direct relationship between the element nodal point displacements and the internal displacements through the use of *interpolating functions.* Finding K and M for an isoparametric finite element then does not require building the A matrix for a particular set of nodal positions, rather H and B are obtained directly.

In general, for an isoparametric element defined by q nodes, the expression for a local point defined by the element is written as in [Bat82],

$$\begin{aligned} x(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)x_i \\ y(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)y_i \\ z(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)z_i \end{aligned} \tag{3.4}$$

where h_i are the q interpolating functions defined in the natural coordinates r, s and t of the isoparametric element, each of which span -1 to $+1$. The q nodal positions are given by x_i, y_i and z_i.

In matrix notation

$$\{X(r,s,t)\} = [h_i(r,s,t)]\{X^e\}$$

Similarly, the function relating nodal displacements to internal displacement within the element is

$$\begin{aligned} u(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)u_i \\ v(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)v_i \\ w(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)w_i \end{aligned} \tag{3.5}$$

where the q nodal displacements are given by u_i, v_i and w_i.

This is also more easily written in the matrix form,

$$\{U(r,s,t)\} = [h_i(r,s,t)]\{U^e\}$$

The interpolation functions h_i are defined in such a way that the value of h_i at r, s and t corresponding to node i is 1 and is 0 at all the other nodes. This property allows the interpolating functions to be determined in a systematic way for any predefined nodal point layout. The specific isoparametric element used in the thesis to model muscle is the twenty-node brick which defines a parabolic interpolation along each of its twelve edges. Refering to Figure 3.2, the twenty nodes are arranged as in Table 3.1.

The interpolation functions h_i for a brick element with from eight to twenty nodes are given in [Bat82],

$$h_0 = g_0 - (g_8 + g_{11} + g_{16})/2$$

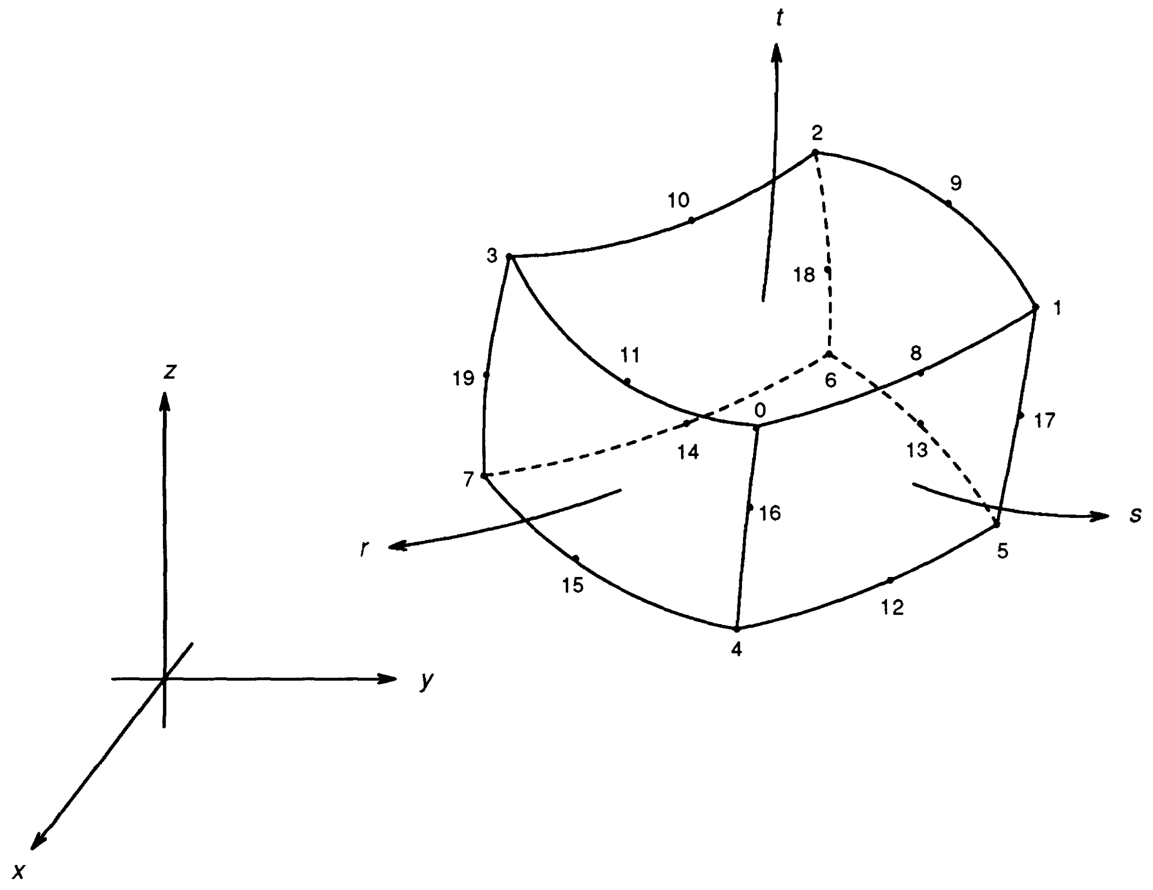


Figure 3.2: Twenty-node isoparametric brick, C numbering. *adapted from* [Bathe]

node	rst	node	rst
0	1 1 1	10	0 -1 1
1	-1 1 1	11	1 0 1
2	-1 -1 1	12	0 1 -1
3	1 -1 1	13	-1 0 -1
4	1 1 -1	14	0 -1 -1
5	-1 1 -1	15	1 0 -1
6	-1 -1 -1	16	1 1 0
7	1 -1 -1	17	-1 1 0
8	0 1 1	18	-1 -1 0
9	-1 0 1	19	1 -1 0

Table 3.1: Local space node coordinates for isoparametric brick

$$\begin{aligned}
h_1 &= g_1 - (g_8 + g_9 + g_{17})/2 \\
h_2 &= g_2 - (g_9 + g_{10} + g_{18})/2 \\
h_3 &= g_3 - (g_{10} + g_{11} + g_{19})/2 \\
h_4 &= g_4 - (g_{12} + g_{15} + g_{16})/2 \\
h_5 &= g_5 - (g_{12} + g_{13} + g_{17})/2 \\
h_6 &= g_6 - (g_{13} + g_{14} + g_{18})/2 \\
h_7 &= g_7 - (g_{14} + g_{15} + g_{19})/2 \\
h_j &= g_j \qquad \text{for } j = 8, \ldots, 19
\end{aligned}$$

where,

$$\begin{aligned}
g_i &= 0 \quad \text{if node } i \text{ is not included, otherwise} \\
g_i &= G(r, r_i)G(s, s_i)G(t, t_i)
\end{aligned}$$

and $\beta = r, s, t$

$$\begin{aligned}
G(\beta, \beta_i) &= \tfrac{1}{2}(1 + \beta_i \beta) \quad \text{for } \beta_i = \pm 1 \\
G(\beta, \beta_i) &= (1 - \beta^2) \quad \text{for } \beta_i = 0
\end{aligned}$$

The twenty interpolation functions were implemented in the C language as formulated above. In addition, C routines were written for the partial derivatives of h_i with respect to r, s and t. These derivative functions will be used presently. A source listing for this code is given in Appendix A. An example program using this subroutine library is presented in **c20_Natural2Local** that does the coordinate transformation of Equation 3.4 to an array of input points given twenty nodal positions $\{X^e\}$.

```
typedef float Vector[3];

c20_Natural2Local( node_points, natural, local, num )
Vector *node_points;    /* twenty nodal positions */
Vector *natural;        /* input natural points */
Vector *local;          /* output local points */
int num;                /* number of points */
{
int i, j;
double r, s, t, x, y, z, h, c20_h();

   for (i=0; i<num; i++)
   {
      r = natural[i][0]; s = natural[i][1]; t = natural[i][2];
      x = y = z = 0.;

      for (j=0; j<20; j++)
      {  h = c20_h( r, s, t, j );
         x += h * node_points[j][0];
         y += h * node_points[j][1];
         z += h * node_points[j][2];
      }
      local[i][0] = x; local[i][1] = y; local[i][2] = z;
   }
}
```

Now, to find the stiffness matrix K for a twenty-node isoparametric element, the principle of virtual work is applied as for the four-node tetrahedron [Bat82]. From Section 3.1.1,

$$[K] = \int_V [B]^T[C][B]dV$$

where B is the matrix that relates nodal displacements to internal strains, that is

$$\{\varepsilon(x,y,z)\} = \left\{ \begin{array}{c} \partial u/\partial x \\ \partial v/\partial y \\ \partial w/\partial z \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \end{array} \right\} = [B]\{U^e\}$$

From the general expression for the displacement interpolating functions given in Equation 3.5, it is easy to see that the matrix form for H is, as a function of r, s and t,

$$\{U(r,s,t)\} = \begin{bmatrix} h_0(r,s,t) & 0 & 0 & \dots & h_{19}(r,s,t) & 0 & 0 \\ 0 & h_0(r,s,t) & 0 & \dots & 0 & h_{19}(r,s,t) & 0 \\ 0 & 0 & h_0(r,s,t) & \dots & 0 & 0 & h_{19}(r,s,t) \end{bmatrix} \{U^e\}$$

The functions h_i are formulated in terms of the natural coordinates r, s, and t. Hence, the partial derivatives of the interpolation functions in terms of the natural coordinates are straightforward to find. The B matrix, however, relates quantities in terms of the local coordinate system x, y and z. Thus, the 3 x 3 *Jacobian* matrix that expresses the chain rule is constructed from Equation 3.4,

$$\left\{ \begin{array}{c} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \\ \frac{\partial}{\partial t} \end{array} \right\} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} & \frac{\partial z}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{bmatrix} \left\{ \begin{array}{c} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{array} \right\}$$

or, in matrix form,

$$\left\{ \frac{\partial}{\partial r} \right\} = [J] \left\{ \frac{\partial}{\partial x} \right\}$$

and, for the derivatives with respect to the local coordinate system,

$$\left\{ \frac{\partial}{\partial x} \right\} = [J^{-1}] \left\{ \frac{\partial}{\partial r} \right\}$$

In practice, the approach taken to calculate B is to first fill a 3 x 20 matrix with the partials of the interpolating functions

$$[D_r] = \begin{bmatrix} \frac{\partial h_0}{\partial r} & \frac{\partial h_1}{\partial r} & \cdots & \frac{\partial h_{19}}{\partial r} \\ \frac{\partial h_0}{\partial s} & \frac{\partial h_1}{\partial s} & \cdots & \frac{\partial h_{19}}{\partial s} \\ \frac{\partial h_0}{\partial t} & \frac{\partial h_1}{\partial t} & \cdots & \frac{\partial h_{19}}{\partial t} \end{bmatrix}$$

then to multiply by the inverse Jacobian to get the partials with respect to x, y, and z

$$[D_x] = \begin{bmatrix} \frac{\partial h_0}{\partial x} & \frac{\partial h_1}{\partial x} & \cdots & \frac{\partial h_{19}}{\partial x} \\ \frac{\partial h_0}{\partial y} & \frac{\partial h_1}{\partial y} & \cdots & \frac{\partial h_{19}}{\partial y} \\ \frac{\partial h_0}{\partial a} & \frac{\partial h_1}{\partial z} & \cdots & \frac{\partial h_{19}}{\partial z} \end{bmatrix} = [J^{-1}][D_r]$$

and the 6 x 20 B matrix is found, according to the strain-displacement relationship, by filling in from D_x

$$[B] = \begin{bmatrix} \frac{\partial h_1}{\partial x} & 0 & 0 & \cdots \\ 0 & \frac{\partial h_1}{\partial y} & 0 & \cdots \\ 0 & 0 & \frac{\partial h_1}{\partial z} & \cdots \\ \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial x} & 0 & \cdots \\ 0 & \frac{\partial h_1}{\partial z} & \frac{\partial h_1}{\partial y} & \cdots \\ \frac{\partial h_1}{\partial z} & 0 & \frac{\partial h_1}{\partial x} & \cdots \end{bmatrix}$$

Now, to obtain the stiffness matrix K requires evaluating the volume integral

$$[K] = \int_V [B]^T [\mathcal{C}][B] dxdydz$$

Note that a change of variables in B has been performed from the local coordinate system to the natural coordinate system and so

$$[K] = \int_V [B]^T [\mathcal{C}][B] \det J drdsdt$$

where $\det J$ is the determinant of the Jacobian matrix.

The last problem to be faced before K can be found for the twenty node isoparametric element is that both the matrix B and $\det J$ are complex functions of r, s, t, and so the volume integral can not in general be explicitly calculated. Instead, a numerical Gauss integration is done to find K [Bat82]. Writing the integral as a sum,

$$[K] = \sum_{i=0}^{3} \sum_{j=0}^{3} \sum_{k=0}^{3} F(r_i, s_j, t_k)\alpha_{ijk}$$

where $F = B^T C B \det J$, and α_{ijk} is a weighting factor that depends on the sample r_i, s_j, t_k under consideration. This summation process is illustrated in the C routine **c20_find_K** that finds K for a material defined by Young's modulus, Poisson's ratio, and an input set of twenty node points.

```
#include <local/matrix2.h>

typedef float Vector[3];

c20_find_K( node_points, youngs_mod, poisson_ratio, K )
Vector *node_points;                /* twenty nodal positions */
double youngs_mod, poisson_ratio;     /* material properties */
Matrix2_D *K;                       /* output stiffness matrix */
{
int i, j, k;
Matrix2_D *B, *C, *mtmp;
double r, s, t, detJ;

   B = m2d_create(6,60);
   C = m2d_create(6,6);
   mtmp = m2d_create(0,0);
   m2d_set_rc( K, 60 , 60 );

   /* C matrix contains material properties */
   fe_strain2stress_LinearIsotropicElastic( C, youngs_mod, poisson_ratio );
   m2d_zero( K );

   for (i=0; i<3; i++)
   {  for (j=0; j<3; j++)
      {  for (k=0; k<3; k++)
         {  Gauss_rst( i, j, k, &r, &s, &t );        /* determine sample points */

            /* Find B(r,s,t) and det[J(r,s,t)] */
            c20_find_B( node_points,  r, s, t, &detJ, B );
```

```
                m2d_transpose( B, Bt );
                m2d_multiply( Bt, C, mtmp );
                m2d_multiply( mtmp, B, mtmp );
                m2d_scale( mtmp, Gauss_weight(i,j,k) * detJ );
                m2d_plus( K, mtmp, K );
            }
        }
    }
    m2d_free( B ); m2d_free( C ); m2d_free( m2d_tmp );
}
```

The routines to obtain the Gauss sample points and weighting factors are,

```
static double gauss_weights[3] =
{ .555555555555556, .888888888888889, .555555555555556
};
static double gauss_sample_points[3] =
{ -.774596669241483, 0., .774596669241483
};

double Gauss_weight( i, j, k )
int i, j, k;
{ return( gauss_weights[i] * gauss_weights[j] * gauss_weights[k] );
}

Gauss_rst( i, j, k, r, s, t )
int i, j, k;
double *r, *s, *t;
{ *r = gauss_sample_points[i];
  *s = gauss_sample_points[j];
  *t = gauss_sample_points[k];
}
```

Futhermore, the consistent mass matrix M for the twenty-node element can also be found through Gauss integration

$$[M] = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} F(r_i, s_j, t_k)\alpha_{ijk}$$

In this case, $F = \rho H^T H$ and the Gauss sample points and weighting factors are as before. This whole process is explained in much greater detail in Bathe's excellent text [Bat82].

3.2.2 Computer Graphics

We have seen how the interpolation functions h_i for the twenty-node brick element can be used to build the K and M matrices that define the finite element dynamics equilibrium equations. As forces are applied to the finite element model, the nodal points of the isoparametric cube deform dynamically in response. For the purposes of computer graphics, one goal of the thesis was to find a way to use familiar geometric representations, such as polygon and spline based patch descriptions, to define objects but to also have them warp and deform as can the twenty-node brick. This is conceptually very easy to accomplish if the points of a computer graphics object—be they points from a polyhedral or patch representation—are treated as natural coordinates r, s and t that sample an isoparametric space. In this way the twenty-node brick can define a *free-form deformation* as Sederburg describes for trivariate Bernstein polynomials in [SP86].

To describe the effect of a free-form deformation, Sederburg makes the analogy of a parallelepiped of clear, flexible plastic into which is embedded objects that are to be deformed. As the plastic is stretched and twisted, so too are the objects inside. This effect is illustrated in Figures 3.3 and 3.4.

Returning to Equation 3.4, the interpolation functions h_i can be seen to define a mapping from the natural coordinate system to the local coordinate system where the twenty nodal positions x_i, y_i, z_i are considered control points that determine the particular shape of a deformation.

$$\begin{aligned} x(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)x_i \\ y(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)y_i \\ z(r,s,t) &= \sum_{i=0}^{q} h_i(r,s,t)z_i \end{aligned}$$

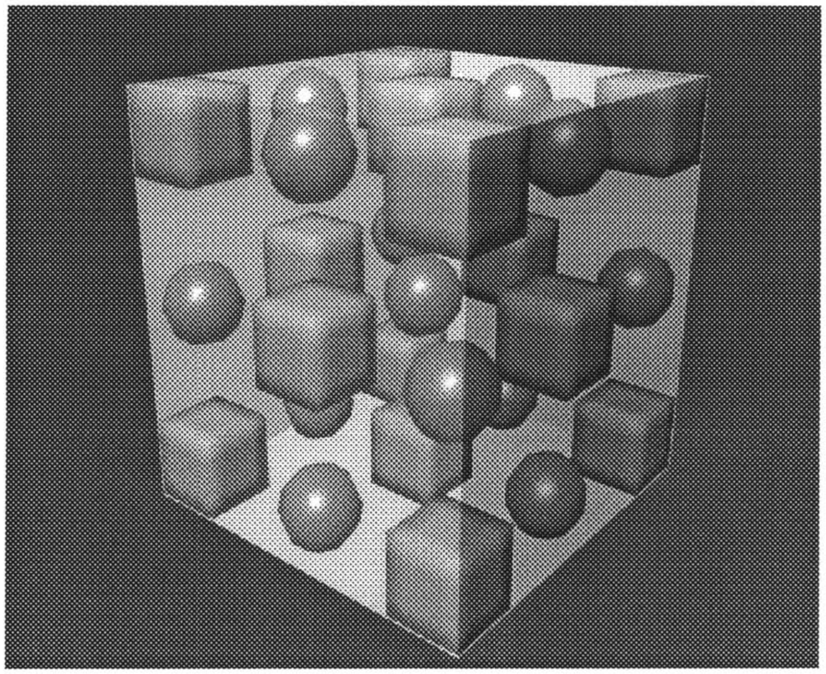


Figure 3.3: Undeformed plastic

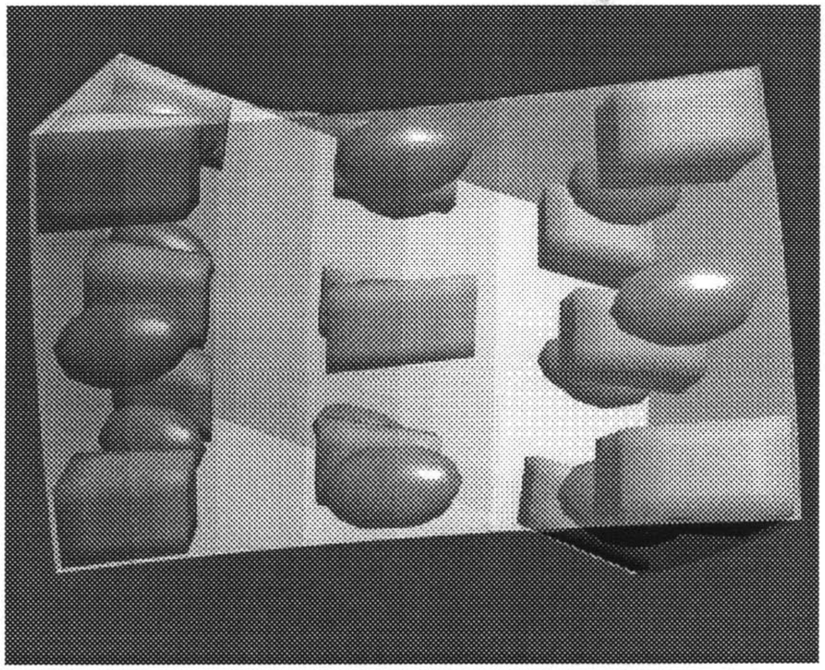


Figure 3.4: Stretched and twisted plastic

So, to dynamically simulate an object, whose geometry is described by point samples connected by polygons or spline patches, as a deformable object that can respond to external forces, three steps must be taken. First, its space is subdivided into some (preferably small) number of adjacent isoparametric finite elements. Then the stiffness and mass matrices are found as has been discussed, and the equilibrium equations solved for a given set of input forces to find the resulting nodal displacements of the isoparametric elements. Finally, using the new nodal positions as control points defining a free-form deformation, the now warped geometric objects embedded in the isoparametric space are rendered to the framebuffer.

The underlying assumption made above is that the points of the object to be drawn are in the natural coordinate system r, s, t of the isoparametric element defining the deformation. In general, of course, this is not the case. The nodes of the isoparametric element and the points describing the object geometry are both, in fact, usually defined in the local coordinate system x, y, z. So, given a computer graphics object and a bounding set of isoparametric elements, it must be determined for each point which containing element to use and then a mapping from local to natural coordinates for that element should be performed. That is, the inverse mapping to Equation 3.4 needs to be found for a given set of nodal points and an input point $L_0 = \{x_0 \;\; y_0 \;\; z_0\}^T$.

Finding this inverse function is relatively straight forward and comes down to a three-dimensional root finding problem which was solved using the Newton-Raphson method as described in [PFTV88]. That is, the problem is to find r, s and t such that

$$[h_i(r,s,t)]\{X^e\} - \{L_0\} = 0$$

The key to this is again the Jacobian matrix J that relates partial derivatives of the coordinate transformation function with respect to x, y and z, to the derivatives with respect to r, s and t.

The basic procedure is as follows. Given a local space point L_0, and twenty nodal positions that define an isoparametric space $\{X^e\}$, make an initial guess for the natural coordinates of point L_0, call this point $N' = \{r' \quad s' \quad t'\}^T$. For this first guess, take the natural coordinates of the Euclidean nearest nodal point of the twenty-node brick from Table 3.1. To find out how close this guess is, calculate

$$\{\beta\} = \{L_0\} - [h_i(r', s', t')]\{X^e\}$$

Now, construct the Jacobian $J(r', s', t')$. To find a correction for N', multiply β by the inverse Jacobian to obtain the step to take in the natural coordinate system

$$\{\delta N\} = [J]^{-1}\{\beta\}$$

and δN is added to the previous solution vector

$$\{N^{new}\} = \{N'\} + \{\delta N\}$$

The process is then repeated with N^{new} until convergence is obtained. A final check must be made to verify that the r, s and t ultimately found is in the interval $[0 - 1]$. If not, then the input local point L_0 is not actually a member of the twenty-node isoparametric finite element defined by $\{X^e\}$.

This iterative technique for performing the local to natural coordinate mapping is illustrated in the routine **c20_Local2Natural** that, for a given twenty node element, will do the inverse transformation for an array of input points. The maximum number of trials allowed is 50, but typically the iteration stops after 5 or fewer repetitions. The convergence check is made by summing the absolute values of the elements of β both before and after the multiplication by J and comparing the result against a predefined tolerance. The tolerance used is 10^{-8} which is approximately the machine accuracy of a single precision float.

```
#include <math.h>
#include <local/matrix2.h>
```

```
typedef float Vector[3];

#define NTRIAL (50)
#define TOLF (1e-8)
#define TOLX (1e-8)

c20_Local2Natural( node_points, local, natural, num )
Vector *node_points;    /* twenty nodal positions */
Vector *local;          /* input local points */
Vector *natural;        /* output natural points */
int num;                /* number of points */
{
int i, j, k;
Vector N, L, L0;
VectorN_I *index = vni_create(3);
VectorN_D *beta = vnd_create(3);
Matrix2_D *Jac = m2d_create(3,3);
double d, errf, errx;

  for (i=0; i<num; i++)
  {
    L0[0] = local[i][0]; L0[1] = local[i][1]; L0[2] = local[i][2];
    make_1st_guess_for_natural( node_points, L0, N );

    for (j=0; j<NTRIAL; j++)
    {  c20_Natural2Local(node_points, &N, &L, 1);        /* Do coordinate mapping to guess */
       vnd_set( beta, 0, L0[0] - L[0] );                 /* Calculate beta */
       vnd_set( beta, 1, L0[1] - L[1] );
       vnd_set( beta, 2, L0[2] - L[2] );

       for (k=0, errf = 0.; k<3; k++)        /* Check for convergence */
         errf += fabs( vnd_get(beta,k) );

       if (errf <= TOLF) break;

       c20_make_jacobian( node_points, r, s, t, Jac );
       m2d_decompose( Jac, index, &d );      /* Solve linear eqns using LU decomp */
       m2d_solve( beta, Jac, index, beta );

       for (k=0, errx = 0.; k<3; k++)        /* Check for convergence again */
         errx += fabs( vnd_get(beta,k) );

       N[0] += vnd_get( beta, 0 );           /* Update solution */
       N[1] += vnd_get( beta, 1 );
       N[2] += vnd_get( beta, 2 );

       if (errx <= TOLX) break;
    }
    r = N[0]; s = N[1]; t = N[2];
```

```
        if (!(r > 1. || r < -1.  || s > 1. || s < -1.  || t > 1. || t < -1.))
        {  natural[i][0] = r; natural[i][1] = s; natural[i][2] = t;
        }
     }
     m2d_free( Jac ); vnd_free( beta ); vni_free( index );
  }
```

The first guess is made by finding the node point closest to L_0, as in the following C code

```
static Vector UNIT_NODES[20] =
{  {1., 1., 1.},     {-1., 1., 1.},     {-1., -1., 1.},     {1., -1., 1.},
   {1., 1., -1.},    {-1., 1., -1.},    {-1., -1., -1.},    {1., -1., -1.},
   {0., 1., 1.},     {-1., 0., 1.},     {0., -1., 1.},      {1., 0., 1.},
   {0., 1., -1.},    {-1., 0., -1.},    {0., -1., -1.},     {1., 0., -1.},
   {1., 1., 0.},     {-1., 1., 0.},     {-1., -1., 0.},     {1., -1., 0.}
};

static make_1st_guess_for_natural( node_points, L, N )
Vector *node_points;
Vector L, N;
{
int j, min_node;
float cur_min, tmp;
float VVdist();            /* find Euclidean distance between two Vectors */

   min_node = 0;
   cur_min = VVdist( L, node_points[0] );

   for (j=1; j<20; j++)
   {  if ((tmp = VVdist( L, node_points[j] )) < cur_min)
      {  cur_min = tmp;
         min_node = j;
      }
   }
   N[0] = UNIT_NODES[min_node][0] );
   N[1] = UNIT_NODES[min_node][1] );
   N[2] = UNIT_NODES[min_node][2] );
}
```

This section has introduced the isoparametric finite element formulation and shown how it can be used to construct the dynamic equilibrium equations that are used for the purposes of modeling deformable objects. It has also been demonstrated how the twenty-node brick element can be used to define a free-form deformation which is used to produce graphical output that visualizes simulation results. For both of these purposes, the interpolating functions h_i were used extensively. In the first case to calculate the Jacobian matrix which was used in the Gauss integration of the stiffness matrix K, and in the second to do the coordinate mapping from r, s, t space to x, y, z, as coded in **c20_Natural2Local**. Because this transformation is used so much, and especially since it is a part of the *simulation loop*, it is worthwhile to think of a way to speed up the process.

The way this was accomplished was to use the symbolic math capability of *Mathematica* to expand the summation in Equation 3.4 by premultipling the effect of the node points, collecting like terms of the resulting polynomial, and simplifying. For the twenty-node brick element this results in a polynomial expression for each of the three primary directions. These polynomial functions are each characterized by twenty coefficients that are in terms of the element nodal positions $\{X^e\}$. Using this formulation, it is easy to rewrite the natural to local coordinate mapping routine to be much faster than before.

```
typedef float Vector[3];

c20_Natural2Local_fast( node_points, natural, local, num )
Vector *node_points;    /* twenty nodal positions */
Vector *natural;        /* input natural points */
Vector *local;          /* output local points */
int num;                /* number of points */
{
int i;
double r, s, t, CX[20], CY[20], CZ[20], c20_X();

   c20_make_polynomial( node_points, 0, CX );
   c20_make_polynomial( node_points, 1, CY );
   c20_make_polynomial( node_points, 2, CZ );

   for (i=0; i<num; i++)
   {  r = natural[i][0]; s = natural[i][1]; t = natural[i][2];
      local[i][0] = c20_X( CX, r, s, t );
      local[i][1] = c20_X( CY, r, s, t );
      local[i][2] = c20_X( CZ, r, s, t );
   }
}
```

Where **c20_X** evaluates a given input polynomial at a particular r, s, t.

```
double c20_X( C, r, s, t )
double *C, r, s, t;
{
double r2, s2, t2, ans;

   r2 = r*r; s2 = s*s; t2 = t*t;

   ans = C[0]                + C[1]   * r * s
      + C[2]   * s * t2     + C[3]   * r * s * t2
      + C[4]   * r * t2     + C[5]   * r * s * t
      + C[6]   * t2         + C[7]   * r2 * t
      + C[8]   * r2 * s     + C[9]   * r2 * s * t
      + C[10]  * r2         + C[11]  * s * t
      + C[12]  * s          + C[13]  * r * t
      + C[14]  * s2 * t     + C[15]  * r
      + C[16]  * s2         + C[17]  * r * s2 * t
      + C[18]  * r * s2     + C[19]  * t;

   return( ans );
}
```

The routine to construct the polynomial coefficients, while straightforward is quite long and so is left for Appendix B. Thanks to Steve Pieper for implementing the blending functions in *Mathematica* and simplifying them.

For an interactive graphics application, an objective is to stay *rendering bound* whenever possible. This means avoiding calculations in the simulation loop that are slower than the time needed to render the final pixels to the display. Comparisons were made using the fast interpolation scheme for objects with different numbers of points both against the straightforward interpolation method and against the rendering time required on an HP graphics workstation using HP's built-in Starbase graphics library in Table 3.2. All times are in seconds.

It is easy to see from Table 3.2 that the fast isoparametric interpolation technique developed in this section is over 13 times faster than the standard interpolation method

no. of points	rendering time	standard interpolation	fast interpolation
386	.17	.56	.042
1352	.46	2.0	.14
8192	3.5	12.0	.9

Table 3.2: Run-time in seconds for two isoparametric interpolation implementations

and about 3 times faster than the rendering times recorded.

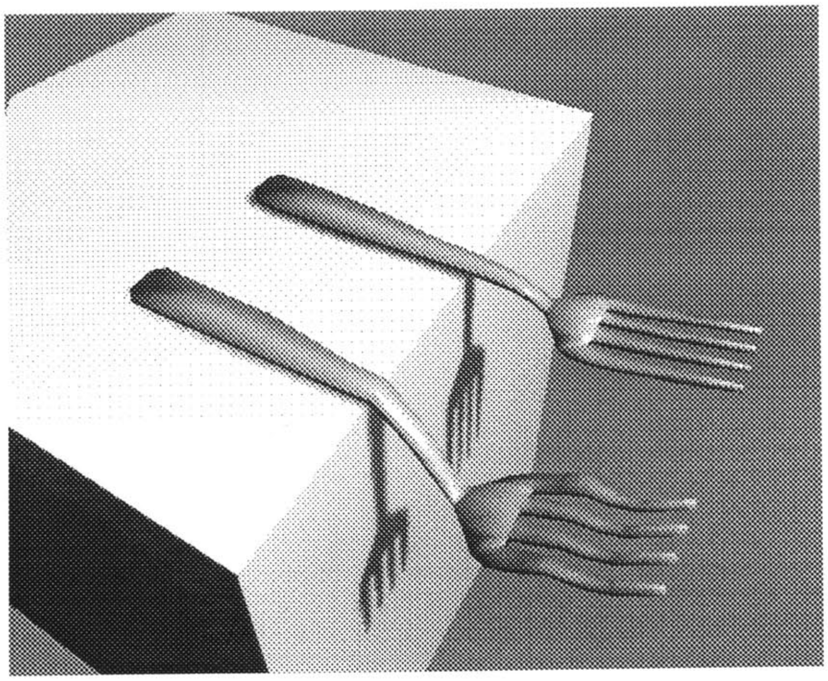


Figure 3.5: A bent fork

3.3 Global Matrix Assembling

In performing a finite element analysis of a muscle, or any kind of real physical structure, it should always be kept in mind the inherent mechanical idealizations that are being made.

Essentially, the displacement based FEM formulation that is used here approximates a given elastic deformable material by discretizing its volume into a collection of elements that can take on certain predefined shapes known as *constant strain states.* Ideally, a good finite element model will converge to the analytic solution—when one is available—as the number of elements increases. For convergence to this "true" solution to happen, there are two criteria that must be met, that is, the elements must be *complete* and *compatible* [Bat82]. Completeness means that the displacement functions of the elements used in the model must be able to represent rigid body motions as well as the constant strain states. It will be seen that these fundamental shapes are determined by the eigenvectors of the stiffness matrix in the next section covering modal methods. Compatibility means that displacement within an element as well as displacement between element boundaries must be continuous. Because the twenty-node brick that is used here describes only the three translational degrees of freedom at each node, compatibility is easily ensured by explicitly joining the nodes of adjacent elements to guarantee that the elements can never separate.

This requirement of sharing nodes between individual finite elements naturally leads to the topic of global matrix assembling, which is perhaps most easily explained in terms of a simple example. Given the structure in Figure 3.6 made from two four-node tetrahedron elements, the nodes comprising element one are $\{0 \;\; 1 \;\; 2 \;\; 3\}$ and those for element two are $\{0 \;\; 1 \;\; 2 \;\; 4\}$. The relationship between nodal displacements and nodal forces for the whole assemblage is as before

$$\{F\} = [K]\{U\}$$

where K is now the *global* stiffness matrix that has, in this case, dimension 15 x 15 and the nodal displacements and forces are

$$\{U\} \;=\; \left\{ \; U_0 \quad U_1 \quad U_2 \quad U_3 \quad U_4 \; \right\}^T$$

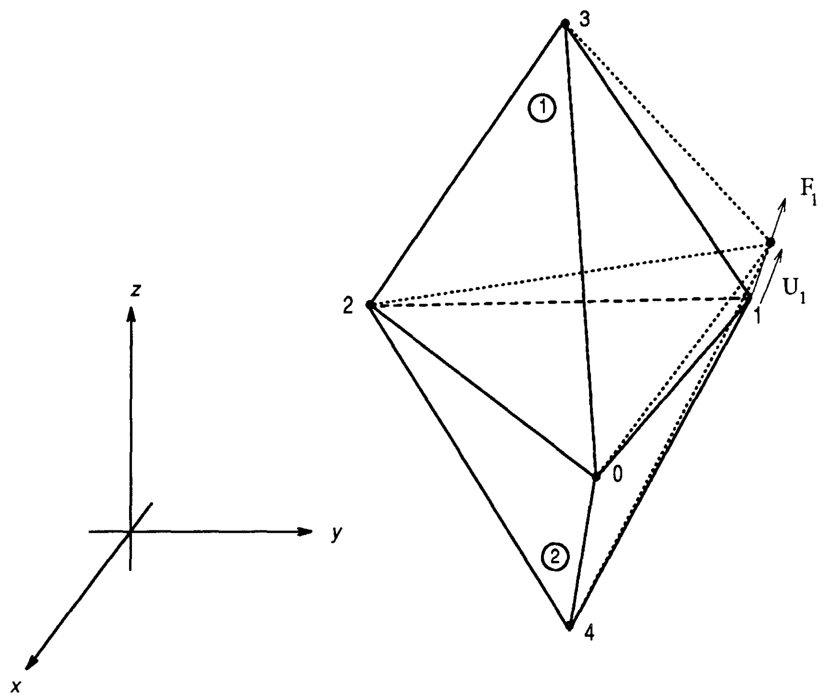


Figure 3.6: Structure made from two tetrahedron elements

$$\{F\} = \left\{ F_0 \quad F_1 \quad F_2 \quad F_3 \quad F_4 \right\}^T$$

With this element and node numbering in place, the *local* stiffness matrices for each element are first constructed individually. For element one

$$\{F^{L1}\} = [K^{L1}]\{U^{L1}\}$$

and for element two

$$\{F^{L2}\} = [K^{L2}]\{U^{L2}\}$$

These local stiffness matrices are made exactly as described earlier in the chapter and each have dimension 12 x 12 for the four-node tetrahedron element. Care must be taken, however, to keep track of the node numbering for the local K matrices

$$\{U^{L1}\} = \left\{ U_0 \quad U_1 \quad U_2 \quad U_3 \right\}^T$$

$$\{U^{L2}\} = \left\{ U_0 \quad U_1 \quad U_2 \quad U_4 \right\}^T$$

Now to construct the global stiffness matrix that determines the equilibrium equations for the whole structure, the principle of superposition is applied to the forces acting at each of the nodes. Looking at node 0, it can be seen that there are relevant forces acting from both finite elements, encoded in the top row of each of the local stiffness matrices. Writing the matrix multiplications out, and remembering that each node actually has three degrees of freedom,

$$\begin{aligned}
\{F_0^{L1}\} &= \{k_{00}^{L1}\}\{U_0\} + \{k_{01}^{L1}\}\{U_1\} + \{k_{02}^{L1}\}\{U_2\} + \{k_{03}^{L1}\}\{U_3\} \\
\{F_0^{L2}\} &= \{k_{00}^{L2}\}\{U_0\} + \{k_{01}^{L2}\}\{U_1\} + \{k_{02}^{L2}\}\{U_2\} + \{k_{03}^{L2}\}\{U_4\}
\end{aligned}$$

The total force acting at node 0 is then the vector sum

$$\begin{aligned}
\{F_0\} &= \left(\{k_{00}^{L1}\} + \{k_{00}^{L2}\}\right)\{U_0\} + \left(\{k_{01}^{L1}\} + \{k_{01}^{L2}\}\right)\{U_1\} \\
&\quad + \left(\{k_{02}^{L1}\} + \{k_{02}^{L2}\}\right)\{U_2\} + \{k_{03}^{L1}\}\{U_3\} + \{k_{03}^{L2}\}\{U_4\}
\end{aligned}$$

Which defines the first three rows of the global stiffness matrix. The other twelve rows are obtained with an identical procedure by summing the force components acting on the remaining nodes. The *global mass matrix* needed for dynamics is derived in a wholly analogous fashion by correctly summing the local element mass matrices.

3.4 Modal Analysis

3.4.1 Solving the Equilibrium Equations

We have discussed the construction of a set of equilibrium equations that determines the dynamic behavior of a finite element idealization of a given physical structure, by first applying the principle of virtual work to obtain local element matrices and then assembling these into a single set of global matrices that governs the structure as a whole. For the dynamic case, these are written as before

$$M\ddot{u} + C\dot{u} + Ku = R \tag{3.6}$$

Where M, C and K are the mass, damping and stiffness matrices; R is the external load vector; and $u, \dot{u}$ and $\ddot{u}$ are the nodal displacement, velocity and acceleration vectors that all have the same rank n. This system can be characterized as a coupled set of second-order differential equations for the displacements u.

It is well known that any problem involving ordinary differential equations can always be reduced to an equivalent system of first order differential equations. Equation 3.6 is rewritten,

$$\dot{u} = v(t) \tag{3.7}$$

$$\dot{v} = M^{-1}(R - Ku - Cv) \tag{3.8}$$

In this form, the time-varying nodal displacements can be solved by specifying a set of initial conditions and using a suitable numerical integrator such as Runge-Kutta or a predictor-corrector [PFTV88]. The way these equations are typically expressed to a standard differential equation solver is through a user supplied **derivs** function that calculates *dydt* at *t*.

```
derivs( t, y, dydt )
double t;
double y[m], dydt[m];
```

Note that m in this case is two times the number of nodal degrees of freedom, or $2n$. The numerical complexity of **derivs** for this kind of solution is order n^2 for doing the three matrix multiplies necessary to calculate $\dot{v}$.

The time spent performing a dynamic analysis is typically dominated by the number of calls to **derivs**. It is possible to take advantage of the block structure of the globally assembled FEM matrices by using a sparse matrix package to speed up the matrix multiplies, but it is possible to do better by first transforming the equilibrium equations into the *modal* displacement basis. The modal transformation is easy to develop and the discussion here follows [Bat82] and [PW89].

First re-write the nodal displacement vector in the following way

$$u(t) = Px(t) \tag{3.9}$$

where P is a square matrix that represents an as yet undetermined transformation and x is a vector with the same rank as u. The elements of the vector x are called *generalized displacements* and are analogous to the generalized degrees of freedom that Schröder writes about in [SZ90].

Substituting this into Equation 3.6 and premultipling by P^T yields,

$$\tilde{M}\ddot{x} + \tilde{C}\dot{x} + \tilde{K}x = \tilde{R}$$

where

$$\tilde{M} = P^T M P; \qquad \tilde{C} = P^T C P; \qquad \tilde{K} = P^T K P; \qquad \tilde{R} = P^T R$$

3.4.2 The Modal Transformation

Now, to obtain the modal transformation, consider the homogeneous equilibrium equations without damping

$$M\ddot{u} + Ku = 0 \tag{3.10}$$

To solve this equation, assume a solution of the form

$$u = \phi e^{j\omega t}$$

where ϕ is a vector of rank n. This yields upon substitution

$$-\omega^2 M\phi e^{j\omega t} + K\phi e^{j\omega t} = 0$$

or,

$$K\phi = \lambda M\phi \tag{3.11}$$

where $\lambda = \omega^2$.

Equation 3.11 has the form of a *generalized eigenvalue* problem. To be able to use the available numerical solvers to find the n eigenvalues λ_i and eigenvectors ϕ_i, this equation needs to be reduced to the standard form $K\phi = \lambda\phi$. This is done by first finding the Cholesky factorization (matrix square root) of the mass matrix

$$M = SS^T$$

Substituting for M in Equation 3.11 gives

$$K\phi = \lambda SS^T\phi$$

and premultipling by S^{-1}

$$S^{-1}K\phi = \lambda S^T\phi$$

Finally, define the vector

$$\phi' = S^T\phi$$

which yields the standard eigenvalue problem

$$K'\phi' = \lambda\phi' \tag{3.12}$$

where $K' = S^{-1}KS^{-T}$.

The eigenvalues and eigenvectors of Equation 3.12 can then be determined and ϕ is found to be $\phi = S^{-T}\phi'$. Lets write the n eigensolutions in the following matrix form

$$\Phi = [\phi_1, \phi_2, \ldots, \phi_n]; \qquad \Omega^2 = \begin{bmatrix} \omega_1^2 & & & \\ & \omega_2^2 & & \\ & & \ddots & \\ & & & \omega_n^2 \end{bmatrix}$$

where the columns of Φ are the eigenvectors ϕ_i and Ω^2 is a diagonal matrix filled with the corresponding eigenvalues $\lambda_i = \omega_i^2$.

The eigenvectors obtained in this way have a special property known as M-orthogonality. This means that

$$\Phi^T M \Phi = I$$

Where I is the identity matrix. To show this, expand the multiplication for two eigenvectors ϕ_i, ϕ_j, using the expressions for M and ϕ from before

$$\begin{aligned} \phi_i^T M \phi_j &= \phi_i'^T S^{-1} S S^T S^{-T} \phi_j' \\ &= \phi_i'^T \phi_j' \end{aligned}$$

Because the eigenvectors ϕ' span an orthonormal basis,

$$\phi_i'^T \phi_j' = \delta_{ij}$$

Where δ_{ij} is the Kronecker delta which is 1, if $i = j$, and 0 otherwise. Now write the n solutions to the generalized eigenproblem in Equation 3.11 as

$$K\Phi = M\Phi\Omega^2$$

Premultipling by Φ^T and using the M-orthogonality of the eigenvectors gives

$$\Phi^T K \Phi = \Omega^2$$

Now it becomes clear that the matrix Φ is a good choice for P in Equation 3.9. That is, using

$$u(t) = \Phi x(t)$$

which relates the mesh nodal point displacements u to generalized displacements x through the eigenvector matrix Φ, the modally transformed equilibrium equations take the form

$$\ddot{x} + \Phi^T C \Phi \dot{x} + \Omega^2 x = \Phi^T R \tag{3.13}$$

where the eigenvector ϕ_i is called the *mode-shape vector* for the generalized displacement x_i and ω_i is the corresponding natural frequency of vibration. That is, a single displacement x_i effects the shape of the entire mesh through the action of the vector ϕ_i at all the node points.

These equations totally decouple as long as the modally transformed damping matrix $\Phi^T C \Phi$ is also diagonal. What is often done in practice is to assume *Rayleigh* damping for a structure, so that

$$C = \alpha M + \beta K$$

where α and β are constants determined either through experiment or to satisfy predefined time decay characteristics. Equation 3.13 can then be written as $3n$ independent second-order differential equations

$$\ddot{x_i} + \gamma_i \dot{x_i} + \omega_i^2 x_i = r_i \tag{3.14}$$

where γ_i is the damping factor for mode shape i, which in the Rayleigh case has the form

$$\gamma_i = \alpha + \beta \omega_i^2$$

So the modal transformation completely diagonalizes the coupled set of differential equations in Equation 3.6. These equations can now actually be solved completely separately from each other, but because of the way forces are calculated it is more convenient to formulate them in a single **derivs** function. Furthermore, the time complexity for **derivs** has been reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ for finding $\dot{v}$ from a given set of input forces. Unfortunately, for the kinds of forces we are interested in simulating, the overall complexity will return to $\mathcal{O}(n^2)$, but more about this in Section 3.4.4.

3.4.3 Computer Graphics

As generalized displacements x_i are computed, the mesh undergoes changes in shape defined by the eigenvectors ϕ_i. These vibration mode shape vectors define the constant strain states for the finite element idealization of a structure to be modeled. That is, any new shape that the structure can assume will be a superposition of some number of the eigenvectors ϕ_i. Thus, the modal transformation procedure also *analyzes* a structure by revealing the natural shape changes that it can undergo. For an assemblage of *complete* elements this will also include the rigid body motion of the structure [Bat82]. For a three-dimensional element, like the twenty-node isoparametric brick that has been discussed, there are six rigid body modes—three translational and three rotational. Since there are $3n = 60$ total degrees of freedom for the twenty-node element, the 54 remaining modes each represent a different deformation of some kind.

The rigid body modes for a three-dimensional structure are encoded in the first six eigenvectors $\phi_0, \ldots, \phi_5$ where the corresponding eigenvalues $\lambda_0, \ldots, \lambda_5$ all have value zero. For the purposes of computer graphics and animation, an additional change of basis is performed for the rigid body modes by replacing these six eigenvectors with the standard translations in the x, y, and z directions and rigid body rotations around the three coordinate axes. This is tantamount to choosing *the right control knobs* as described in [PW89] so that the dynamic, deformable models can be more easily positioned and controlled. In our case, however, the remaining constant strain modes will be left intact to govern the deformation response due to input forces.

More importantly perhaps, by performing this replacement of the first six eigenvectors by simple rigid body translations and rotations, it is straightforward to achieve the effect of a *Total Lagrangian* analysis as discussed in [Bat82]. The Total Lagrangian is important when the element bodies undergo large displacements or large rotations, as is typical in computer graphic animations. This is because the global K matrix, as developed above, only properly expresses the nodal relationships in the configuration in which the Gauss integration is performed. Intuitively, we can realize that the Jacobian matrix that relates the natural coordinate system of an isoparametric finite element to the local reference frame, and which plays a primary role in the construction of the stiffness matrix, changes dramatically as the element assemblage undergoes large rotations. For an analysis that takes care of such rotations, the equations of motion are arranged so that external forces always act in the original space of the body, thus obviating the need to recalculate K. By separating the effects of rigid body motion from elastic deformation the required force transformation is simple to compute. This subject will be dealt with in greater detail in Section 3.4.4.

Section 3.2.2 discussed how the points of a polygonal or spline based object embedded in the isoparametric space of a finite element mesh can be warped. The modal transformations change this operation little. The only differences are that since *generalized* displacements are now being solved for, the *nodal* displacements must be calculated before the rendering step as

$$u = \sum_{i=6}^{3n} \phi_i x_i$$

Furthermore, the rigid body position and orientation can be simply determined from the first six generalized displacements as a series of standard 4 x 4 affine matrix operations where the three rotations accept angular radians

$$Rz(x_5) \cdot Ry(x_4) \cdot Rx(x_3) \cdot T(x_0, x_1, x_2) = M_{rot} \cdot M_{trans} = M_{rigid}$$

The first five eigenvector mode shapes $\phi_6, \ldots, \phi_{10}$ are illustrated for a single twenty-node brick element and for the Cyberware head model made from three twenty-node bricks in Figure 3.7 and Figure 3.8.

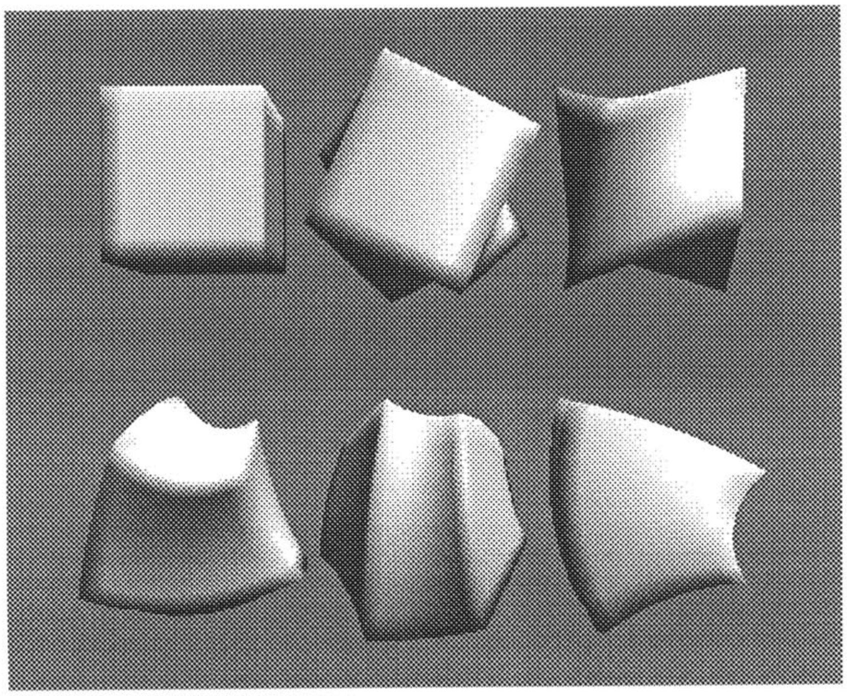


Figure 3.7: Mode shapes for twenty-node brick

3.4.4 Deriving Modal Forces

To find the forces necessary to drive the modal form of the equilibrium equations, two steps are taken. First the external world space forces, R, acting on the node points of the finite element mesh are found. Then the modal equivalent forces, R_{modal} are resolved by performing the transformation discussed above,

$$R_{modal} = \Phi^T R$$

However, because the rigid body translations and rotations are not treated in the

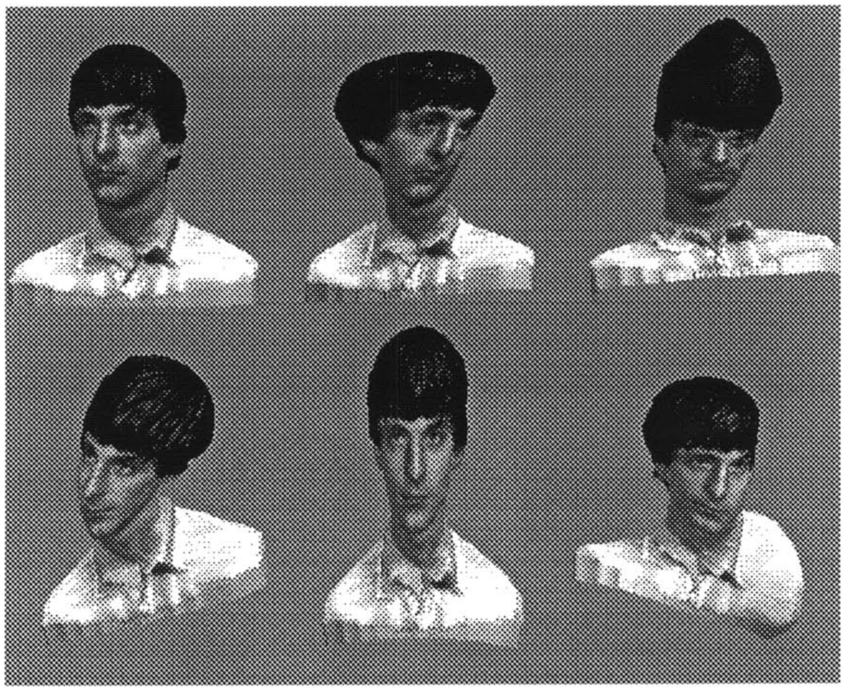


Figure 3.8: Mode shapes for Cyberware head

matrix Φ, two further complications are introduced. The world space force R must first be rotated into the original coordinate system (local space) of the dynamic body before the eigenvector multiplication is carried out, and the explicit world space rigid body forces must be computed.

The components of the world space force vector R are comprised from a variety of sources—force fields such as gravity, spring/damper systems acting at the mesh node points, constraint goal positions, muscle fiber or tendon forces, and collision/contact forces. The translational part of the rigid body force is found simply by summing the forces acting at the n node points,

$$R_{trans} = \sum_{i=0}^{n} R_i$$

Assuming the center of mass of the finite element mesh is at the origin, the rigid body torques can be calculated with an element-wise cross product operation to R and the current world space location of each of the nodal points P,

$$R_{rot} = \sum_{i=0}^{n} P_i \otimes R_i$$

To calculate the modal elastic forces that will govern the deformation of the finite element mesh, the elements of R must first be transformed into the body's local space using the inverse of the rotation matrix M_{rot} defined in Section 3.4.3,

$$R_{lsp}[i] = R_i \cdot M_{rot}^T$$

It is these local space forces that are finally multiplied by the transposed eigenvector matrix to yield the modal equivalent forces,

$$R_{modal} = \Phi^T R_{lsp}$$

Note that because the first six modes have been replaced by explicit rigid body translations and rotations, only elements 6 through n of R_{modal} are used.

The arrows in Figure 3.9 show the magnitude and direction of world space forces, R_i, acting at the twenty mesh nodal points of a simple cube element. The main force component is from a constraint that drags node 0 of the mesh upwards and to the right to meet the soccer ball (R_0). Forces at the other mesh points are from collisions with the floor and gravity. Note that even though the gravity acceleration is always directed downwards, the force to produce this acceleration points upwards at the corner nodes. This is due to a non-uniform distribution of the cube's mass and is a direct consequence of using the *consistent mass matrix* discussed in Section 3.1.2.

3.4.5 Mode Truncation

The development of the modal transformation, as presented here, was motivated by a desire to speed up calculations within the **derivs** function called by the differential equation solver

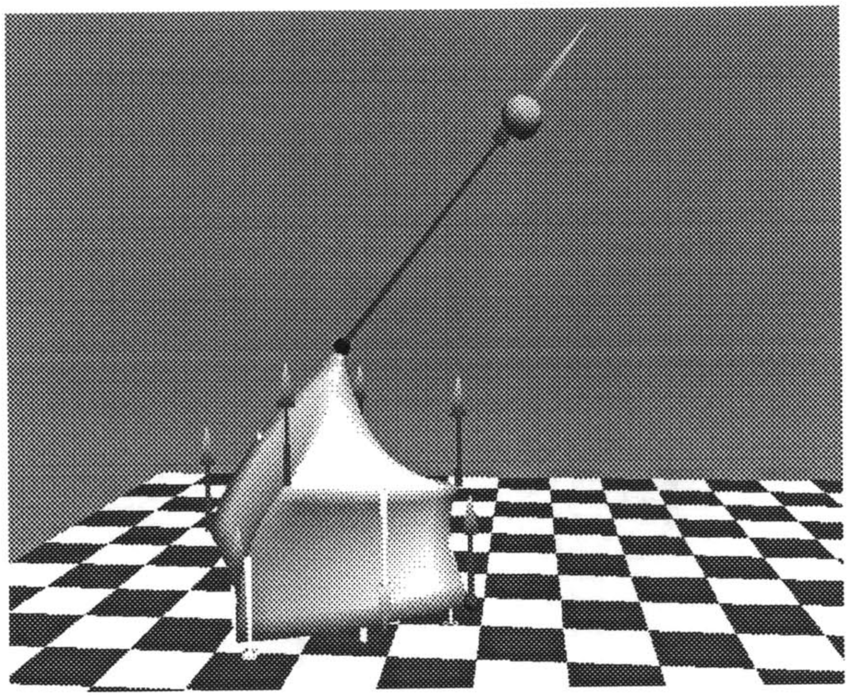


Figure 3.9: World space mesh forces for twenty-node brick

that is used to integrate the finite element equilibrium equations. The form of Equation 3.8, indicates that solving the standard form of these equations is at least $\mathcal{O}(3n^2)$ for the three matrix multiplies. By determining the answer to the generalized eigenvalue problem and constructing the matrix Φ as a preprocessing step to solving the ODE's, the matrices M, C and K are diagonalized and so it would seem that **derivs** now has complexity $\mathcal{O}(3n)$. However, because of the way the force vector R is calculated, the solution complexity returns to $\mathcal{O}(n^2)$. That is, finding the modal equivalent force, R_{modal}, to a time-varying external world space forcing function requires the matrix multiply from above,

$$R_{modal} = \Phi^T R_{lsp}$$

Furthermore, to enforce the kinds of constraints desirable for computer animation, the current world space mesh nodal point locations must be updated within **derivs** as well.

One such constraint would be the attachment of a mesh point to a world space location, as in Figure 3.9. The mesh displacements, $u(t)$, are then calculated within **derivs** from the generalized displacements, $x(t)$,

$$u(t) = \Phi x(t)$$

At first glance then, it may appear that the modal transformation has done little except change the complexity of **derivs** from $\mathcal{O}(3n^2)$ to $\mathcal{O}(2n^2)$. In practice, however, there can be considerable savings with this method. In [Bat82], Bathe states that frequently a good approximate solution to the actual response of a real physical system can be gotten by considering only a small fraction of the total number of decoupled equations. That is, by considering only equations $i = 1, \ldots, p$ of Equation 3.14 where $p << n$, a good approximation to the physical response of a system can still often be obtained.

By truncating the decoupled set of equations to include only the first p modes yields three advantages. First, only the first p eigenvalues and eigenvectors need to be found. Second, the two matrix multiplies with Φ that are now calculated by **derivs** can instead be written as the sparser summations,

$$u(t) = \sum_{i=6}^{p} \phi_i x_i(t)$$

Most importantly in terms of computational efficiency, however, mode truncation can lead to many, many fewer calls to the **derivs** function by the differential equation solver. The eigenvalues $\lambda_i = \omega_i^2$ found from the stiffness matrix K determine the frequency of vibration for the associated mode shape vector ϕ_i. Depending on the frequency content of the input forcing function, different combinations of modes will be excited. A standard ODE solver with adaptive stepsize control will automatically subdivide its interval to capture the highest frequency changes. Thus, by bandlimiting the impulse response of the system through mode truncation, fewer calls to **derivs** will result.

The tradeoff inherent in mode truncation is computational speed versus the accuracy of the final solution. One way to estimate the effect of mode truncation on the quality of a simulation is to consider that for modes larger than p, the effective force will be zero, so that the mode shapes ϕ_i for $i > p$ will not have any effect. From the previous section, it was seen that for our modal formulation, the input vector R contributes to both rigid body and deformation force quantities,

$$\begin{tikzcd}
 & & R \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\
R_{\mathrm{trans}} & R_{\mathrm{rot}} & R_{\mathrm{lsp}} & R_{\mathrm{modal}} = \Phi^{T} R_{\mathrm{lsp}}
\end{tikzcd}$$

So the effect of truncating modes will not produce any error in the important rigid body response of a system, but could lead to underestimating the maximum deflections generated. Futhermore, because modes are deleted, the result is always to remove force from the system. Hence the solution found in this way will be an *upper bound* to the solution expected by making $p = n$.

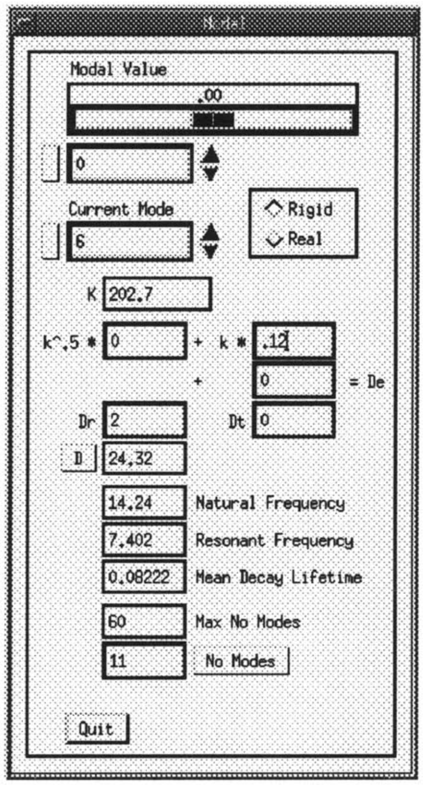
From [Bat82], it is known that the number of modes to be used to model a system depends on the characteristics of the structure itself and the spatial distribution and frequency content of the loading. Bathe gives the example of earthquake loading where in some cases only the ten lowest modes need to be analyzed, although the order of the modeled system n may be larger than 1000. On the other hand, for high frequency blast loading, p may need to be as large as $2n/3$. For the frog gastrocnemius discussed in Section 6.1, the mode shape vector corresponding to the simple longitudinal extension and shortening expected upon isotonic muscle activation was found, and p set to be 2-3 times this value. For the example of Section 6.1, ϕ_{14} corresponds to this mode and p is set to 40. The forces developed upon contraction have a relatively smooth characteristic as seen in the plots of Section 2.2.3.

Bathe has developed an error measure for mode truncation that uses U^p, the displacement response predicted by mode superposition when only p modes are included,

$$\epsilon^p(t) = \frac{\| R(t) - [M\ddot{U}^p(t) + KU^p(t)] \|}{\| R(t) \|}$$

Unfortunately, this is difficult to compute given the details of the modal analysis programs developed here. First, $\epsilon^p(t)$ does not include the effects of damping. Damping is a part of our analysis, but not through the explicit construction of a global damping matrix C. Second, the forces corresponding to rigid body motion would have to be summed in some way with the $M\ddot{U}^p(t)$ terms in order to take care of all inertia components. Futhermore, though $\epsilon^p(t)$ measures how well the nodal point loads are balanced by inertia and elastic forces, it is still not entirely clear where the line should be drawn to be able to say that $\epsilon^p(t)$ should be below some prescribed value.

Hence, the approach taken is to provide a simple interactive method, based on the techniques of Section 5.5, to allow the user to set p for a given simulation. Relevant engineering quantities, such as the natural frequency for a mode, the resonant frequency, and mean decay lifetime are also presented so that the user can make an informed decision for p based on what is known about the loading conditions to be used. A dialog box that provides this functionality is shown in Figure 3.10.



Modal

Modal Value

.00

0

Current Mode

6

Rigid

Real

K 202.7

k^.5 * 0 + k * .12

+ 0 = De

Dr 2 Dt 0

D 24.32

14.24 Natural Frequency

7.402 Resonant Frequency

0.08222 Mean Decay Lifetime

60 Max No Modes

11 No Modes

Quit

Figure 3.10: Modal popup menu

Chapter 4

Musculoskeletal Structure

The idea of capturing the complete internal anatomy of frog and human legs through MRI or other digitizing processes was discussed in Section 2.3. The important information to be preserved was the relative *in vivo* geometry of distinct muscle masses and bone. To dynamically simulate a whole musculoskeletal structure, however, is certainly not a simple task. Many problems are introduced by considering the whole system. Besides the highly non-linear internal muscle forces discussed in Section 2.2.3, new forces must be calculated for the fascia attachments modeling contact between bone, muscles and skin. A simulated skeleton can be driven by muscle forces calculated at the tendon attachment points. Higher-level coordination and reflexes can be modeled through a computational central nervous system. Though much of the implementation of these problems will be left as "future work", this chapter will try to point out some useful representations and present a framework around which the musculoskeletal structure can be captured by extending the techniques heretofore developed.

4.1 Skeleton Kinematics

4.1.1 Denavit and Hartenberg Joints

Using the formalism developed by mechanical engineers and robotics researchers, jointed figures generally are considered to be networks of linked manipulators. The legs and arms of a human skeleton, for instance, can be described as manipulators attached to a common fixed reference frame centered on the body. The hands and feet in this case play the role of *end effectors.*

A very common coordinate frame representation in use is that of Denavit and Hartenberg [Pau81]. The DH notation represents each of serially linked joints with four real numbers that specify a new position and orientation relative to the previous coordinate frame. Paul establishes the relationship between joints $n-1$ and n with the following series of rotations and translations,

1. rotate about z_{n-1}, an angle, θ_n
2. translate along z_{n-1}, a distance d_n
3. translate along rotated $x_{n-1} = x_n$ a length a_n
4. rotate about x_n, the twist angle α_n

The DH representation explicitly defines joints that are either revolute or prismatic. To model a spherical joint, a system of three coincident revolute joints, oriented so that any two of the defined axes of rotation are not parallel, is used. The treatment here, as the emphasis is on approximating human anatomical systems, will deal almost exclusively with revolute joints. The movement of genuine biological joints, of course, is much more complicated and is defined by the contact of shapen articular cartilage bone ends held together

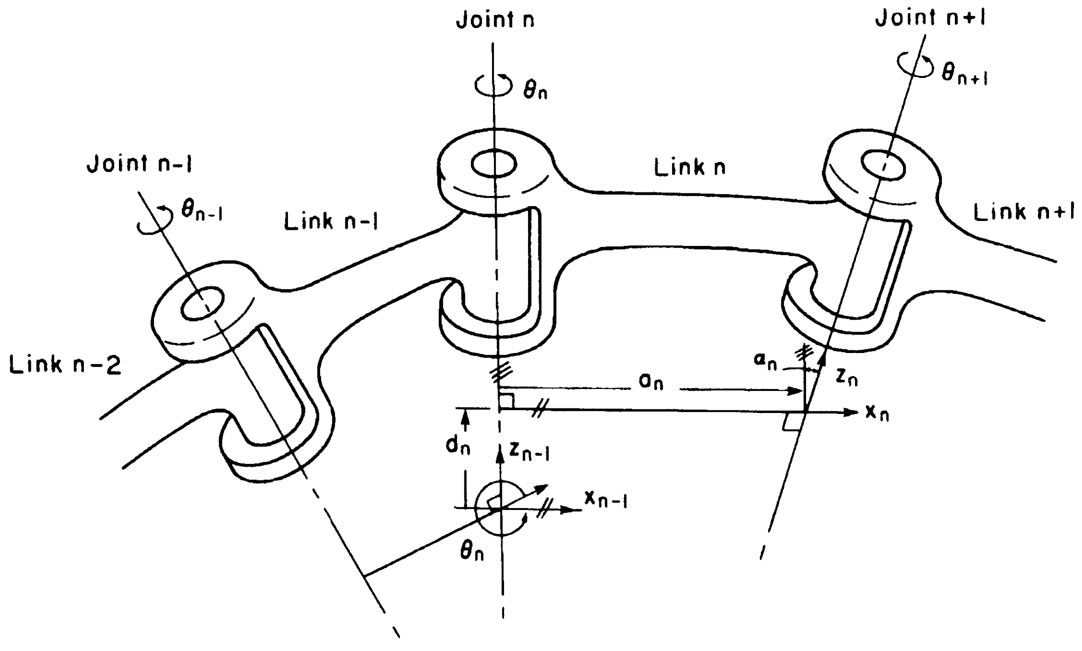


Figure 4.1: Denavit and Hartenberg link parameters [Pau81]

by ligaments [Kel71]. A single DH quadruple is used to represent each degree of freedom of the manipulator. A simple lever arm can be described with one DH quadruple, while a spherical joint requires three. The human upper arm can be idealized as a kinematic system composed of five links and eleven degrees of freedom, and so requires eleven DH quadruples to represent the joint coordinate frames. These eleven degrees of freedom correspond to three DOFs each at the clavicle, scapula and humerous and one DOF at both the ulna and radius.

A further complication is admitted because it is not always possible to rotate the subsequent $(n+1)$ coordinate frame to line up as desired since the twist angle α produces a rotation only around x_n. Thus, it is sometimes necessary to introduce "dummy" joints that do not move, but only serve to modify the coordinate frame orientation. The Denavit-Hartenberg parameters used for the left arm model are presented in Table 4.1. The dummy joints are indicated in the *frozen* column.

joint	name	θ d a α	frozen
0	clavicle	0 0 0 90	0
1		10 0 0 90	0
2		0 0 5.3 0	0
3	scapula	60 0 0 90	0
4		100 0 0 90	0
5		0 0 2.3 0	0
6	humerous	0 0 0 -90	0
7		80 0 0 90	0
8		50 0 0 90	0
9		90 0 10 0	1
10	ulna	100 0 0 90	0
11	radius	-90 0 0 90	0
12		90 0 8.6 90	1
13	wrist	0 0 0 90	0
14		0 0 6.6 0	0

Table 4.1: DH parameters for human left arm

While these kind of joint systems do not inherently contain any muscle or dynamics information, they are particularly convenient vehicles on which to perform kinematic simulations. Methods that have been proposed to perform locomotive control of such articulated systems include the use of finite state machines arranged in a hierarchal control structure by Zeltzer, Jacobian control and dynamic simulation.

Zeltzer in his PhD thesis [Zel84] talks about the task of controlling a multilink biological motor system as a *degrees of freedom* problem. That is, there are many more degrees of freedom that need to be specified than can be comfortably conceptualized. His solution is to use a form of *functional abstraction* by introducing "local motor programs" (LMP's) that directly control specific classes of motions for a particular set of joints. These LMP's, can be viewed physiologically as low-level "reflex processes" which are regulated by higher-level controllers, or *skills*. At the highest level is a *task manager*, that takes goal descriptions from the user—"Go to the door and open it", and decomposes them into lists of component skills. In addition to the work of Zeltzer, this kind of system has been implemented in [Str91] and [Joh91].

4.1.2 Inverse Kinematics

One way of implementing LMP's is with inverse kinematics, a well-known technique developed by robotics researchers that solves the degrees of freedom problem for a manipulator arm by requiring only the specification of the end effector's position and orientation. The essential component of this method is the Jacobian matrix, which directly relates the end effector linear and angular velocities $\frac{dx}{dt}$ to rotary joint velocities $\frac{d\theta}{dt}$. In his thesis, Ribble [Rib82] uses inverse kinematics to control a walking human skeleton, and specifies hardware to perform the needed computations quickly.

The shape of the Jacobian is n by m where n is the number of known end effector velocities and m is the number of free joints in a single manipulator. The number of knowns is usually six, if both end effector linear and angular velocities are specified, or three if only linear velocities are available. As was seen before, a typical manipulator has between four and twenty degrees of freedom.

So, Jacobian Control of a robot arm implies finding rotary joint velocities from goal end effector velocities, that is, inverting the Jacobian to find $\frac{d\theta}{dt}$ from $\frac{dx}{dt}$. But, as the Jacobian is not generally a square matrix, the Moore-Penrose pseudoinverse of the Jacobian matrix is typically used. The pseudoinverse leads to the *minimum norm* solution for a particular under or over-constrained linear system. To model an entire skeleton, a separate Jacobian is needed for each end effector that will be independently motile. Thus, specifying skeleton motion becomes a problem of correctly solving for the paths that each of the hands and feet will follow. Because the system of equations defined in this way is in most cases underconstrained, a further degree of control over the motion can be obtained by specifying a *secondary goal* vector that controls the relative values of the final velocities computed for each DH joint.

For example, Figure 4.2 shows the skeleton arm swinging from back to front. The motion was defined by using the soccer balls as beginning and ending constraint points for the fingertip end effector.

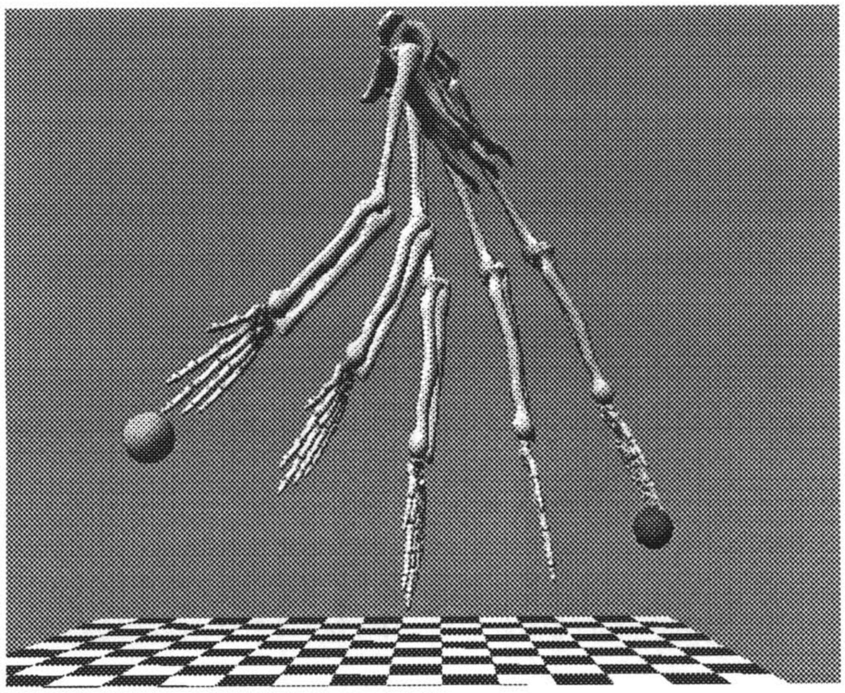


Figure 4.2: Inverse kinematically controlled skeleton motion

Finally, dynamic simulations of articulated bodies have been done in the computer graphics community by Armstrong and Green [AG85], Wilhelms [Wil87], Bruderlin [BC89] and McKenna [McK90]. Forward dynamics can be done if the limb inertias can be calculated and relevant external forces are known. Featherstone [Fea87] writes the equations of motion for an articulated robot as,

$$Q = I(q)\ddot{q} + C(\dot{q}, q, x)$$

where Q is the vector of torques acting at the joints, $q, \dot{q}$, and $\ddot{q}$ are the joint variables and derivatives for the robot and C is a vector of forces *not* dependent on the robot's acceleration, including gravity and other externally applied forces. In general, dynamic simulations produce more realistic, but harder to control motions than purely kinematic simulations.

4.2 Muscle Coordination

Dynamic biomechanic models of musculoskeletal structures have been made in [AD85], [Hat76], [McM84]. With these models, the modus is usually to study individual limbs such as the leg, rather than examining the body in toto. One important topic of such research is to examine how muscles can be controlled to produce specific motions. For example, Audu [AD85] discusses a simple lower limb linkage with one degree of freedom at the hip and another at the knee. He compares the effect of different lumped muscle models, most variations of Hill's, that control the leg in performing a *minimum time kick*, illustrated in Figure 4.3. The minimum time kicking problem was posed in [Hat76]. The leg initially hangs free and is then kicked forward as quickly as possible to a predetermined position. At the end, both θ_1 and θ_2 are specified and the knee velocity $\dot{\theta}_2$ is zero. The solution to this problem lays out control histories for the five muscle actuators so that the kicking time is at a minimum and the final joint angles are met within some tolerance. The strategy that Audu outlines is based on techniques from optimal control theory for finding the best way to activate the muscles.

Another approach to producing coordinated muscle group action has been suggested

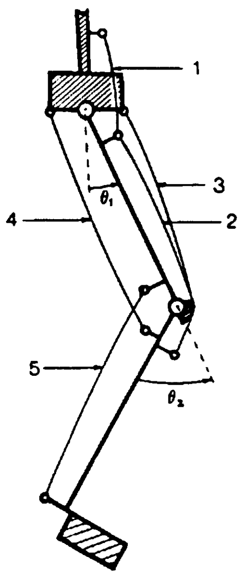


Figure 4.3: Lower limb model for minmum time kicking problem. The angle θ_1 defines the hip. The single DOF θ_2 is at the knee. Lines 1 through 5 define muscle groups acting on the limb. [AD85]

by Wood [WMJ89] [Woo76]. His method seems more easily modularized and extensible to modeling an entire 3D musculoskeletal system than that from Audu, and so is presented here. Wood develops *anatomy matrices* for musculature that relate muscle forces to torques about specific joints. This can be written for the j-th joint and k-th muscle as,

$$A_m(jk) = ((X_k - Z_j) \otimes L_k) \cdot U_j$$

where from Figure 4.4 it can be seen that $A_m(jk)$ is the (time-dependent) scalar *lever arm* which converts muscle force $F_m(k)$ into a scalar torque $M_m(j)$ about joint (U_j, Z_j). The line-of-action for the muscle force $F_m(k)$ is defined by the unit vector L_k, and the point of attachment is X_k. If there are several muscle forces acting about joint j, then the total net moment for musculature $M_m(j)$ about j is given by the summation over k,

$$M_m(j) = A_m(jk)F_m(k) \tag{4.1}$$

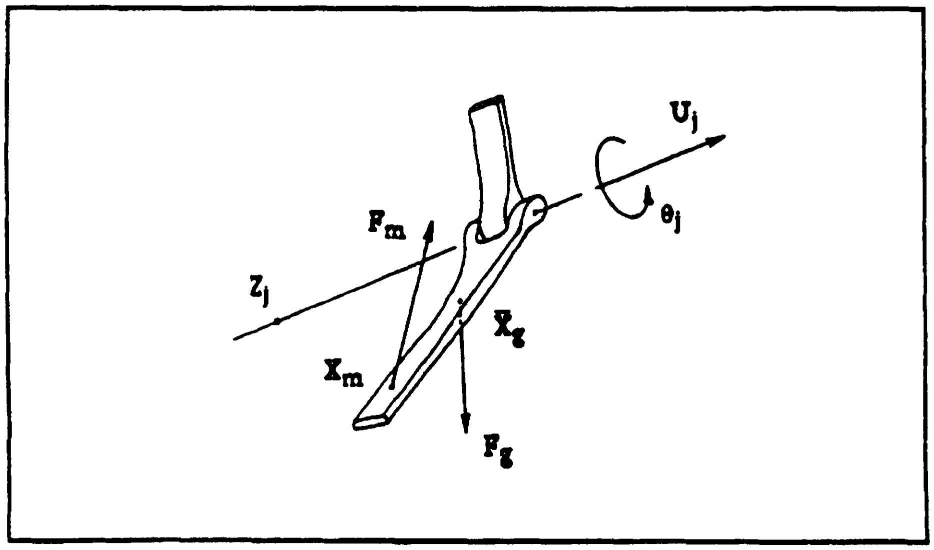


Figure 4.4: Muscle acting about the revolute joint defined by unit vector U_j passing through point Z_j. [Woo76]

This relationship can be extended to handle the effects of gravitational and ligament forces in an analogous fashion by constructing additional matrices. Wood, in his research of prosthetic control, estimated muscle forces from EMG signals collected by electrodes connected to the skin. A linear recruitment relationship, as well as a Hill-type muscle model were used to find these forces. These are then transformed into joint torques by the anatomy matrix, and the torque value used to control the joint actuators of an artificial limb.

By inverting the matrix in Equation 4.1, values for muscle force $F_m(k)$ can be found simply from known joint torques. Thus the *anatomy matrix* can also be used as part of a muscle control mechanism in which a higher-level planning system first estimates the joint moments needed to produce a desired movement. Because the number of joints is typically much less than the number of muscles, $A_m(jk)$ is underconstrained, and so the same pseudoinverse techniques that are described by Ribble to perform skeleton control can then be used to find the vector of muscle forces necessary to produce the specified action. Patriarco [PMSM81] has discussed some of the physiological issues behind different optimization functions (secondary goals) for the determination of muscle forces in this way.

4.3 Attachments, Collisions and Constraints

Besides skeleton representation and muscle coordination, a realistic 3D musculoskeletal model must have a mechanism for enforcing contacts between colliding muscle masses and between muscle and bone through fascia attachments. To meet the goals of computer animation, easy-to-create sequences of complex deformable objects are possible only if techniques for automatically simulating the effects of real physical objects are provided. In Chapter 3, the finite element method was developed so that it could be used to make computer animations of a multitude of visco-elastic models. Here the discussion concerns calculation of the nodal point force vector $R(t)$ so that attachments, collisions and contacts can be realistically simulated.

Platt [PB88], [Pla89] describes many different methods that can be used to constrain the solution of systems of differential equations by calculating constraint forces. One such technique, appropriate for deformable models, is *reaction constraints* (RC's). Reaction constraints are applied to single mass points of a discretized body. RC's work by first calculating the net force active at a point due to physics or other kinds of constraints, call this F_{in}. The RC then projects out the component of F_{in} in the direction of the constraint, yielding $F_{unconstrained}$. Next, $F_{constrained}$ in the direction of the constraint is calculated to yield a critically damped motion that satisfies the constraint. The resultant force to be applied to the point, F_{out} is simply the sum $F_{unconstrained} + F_{constrained}$. The concept is illustrated for a single mass point in Figure 4.5.

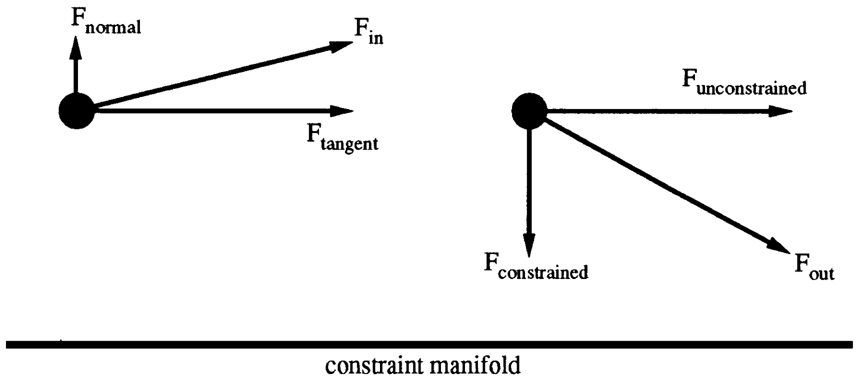


Figure 4.5: The constraint goal is to place the mass point on the line. F_{in} pushes the point up and right. The component F_{normal} is undesirable, $F_{tangent}$ is acceptable. The force $F_{constrained}$ is created by the constraint and sums with $F_{unconstrained}$ to produce F_{out}. *adapted from* [Pla89]

There are two advantages to this technique. First, it is simple to implement. To attach a node of the finite element mesh to a point in space, $F_{constrained}$ is found simply as a spring force. Second, unlike a spring, the desired constraint can be met *exactly* without using a high stiffness value. The reaction constraints executed in the thesis can attach a nodal point to a world space location, and model collisions of the finite element mesh against algebraic surfaces or against arbitrary polyhedral objects. This is done within the **derivs** function by updating the mesh displacements for the current time and measuring how far the constraint is from being satisfied. The nodal force vector $R(t)$ is then calculated to move the mesh points closer to the constraint manifold as discussed above.

The collision analysis software developed for the thesis is a modified version of the algorithm presented in [MW88]. Because a vector normal to the constraint surface is required to generate forces in the proper direction, collisions are always calculated using the mesh edges. An edge-polygon intersection routine from a raytracer is used to determine the polygons that define the collision. The force $F_{constrained}$ is in the direction of the intersected polygon and is scaled by the depth of penetration. Example animations using these constraints are presented in Section 6.2.

Chapter 5

Implementation

5.1 The *3d* Animation/Dynamic Simulation System

One goal of the thesis work is to make a "testbed" system to help script procedures for computer animation, develop algorithms that control dynamic simulation, and that will help us build interactive applications. The software for the resulting program, *3d*, is being developed on an HP TurboSRX graphics workstation running UNIX^{TM}. The approach is to make a simplified version of an interactive command interpreter like BASIC or LISP, that has many special purpose rendering, dynamics, numerical math, and user-interface functions all integrated at a relatively "high-level". The entire *3d* program has a binary image size of only 2.6M which includes the RenderMatic software A-buffer renderer. However, it is the workstation's ability to draw pictures quickly with its special purpose hardware that is the key to the approach taken here in designing the system. The hope is that *3d* will allow simple design of new animations, improved dynamics algorithms and rapid interface prototyping. The work in this area is inspired by *bolio* [ZPS89] [Pie90] and *corpus* [McK90], two other command interpreter programs written for the HP workstation here in the Computer Graphics and Animation Group.

The *3d* program is based on the Tcl embedable, application-independent, "tool command language" from U.C. Berkeley [Ous90]. Tcl is distributed as a C library package and is designed to be used in many different programs. Included with the library is a parser for a simple but fully programmable command language as well as a small collection of built-in functions that support general-purpose language features such as variables, lists, expressions, conditionals, looping and procedure definition. An instance of an application program based on Tcl, such as *3d*, extends the basic set of Tcl commands with any number of application-specific commands. The Tcl library also provides a set of utilities that simplify defining these application-specific commands.

The application-specific code for *3d* has been developed on top of the RenderMatic C library of graphics and rendering software. The command interpreter has over 700 built-in and application-specific functions. There are primitives for rendering and graphics, math, matrix and vector operations, general data structure manipulation, Denavitt and Hartenburg joint description, finite element dynamics and X/Motif interface building. The intent of combining this functionality into a single program is to allow easy prototyping of different graphics-based interactive applications.

There is a two-tiered approach to developing software using *3d*. First, because Tcl allows procedure definition, it is easy to make collections of useful subroutines in the interpreted language that can control object, light and camera motion. One of the main benefits of creating an interpreted layer of language is that development of animation scripts can be very rapid. Second, pieces of code that must run rapidly or interactively can be written in C and then easily imported as new application-specific functions for the command interpreter. It is important to note that these functions can usually be prototyped first as Tcl procedures (`tclprocs`) before implementation in C. In this way a "top down" pro-

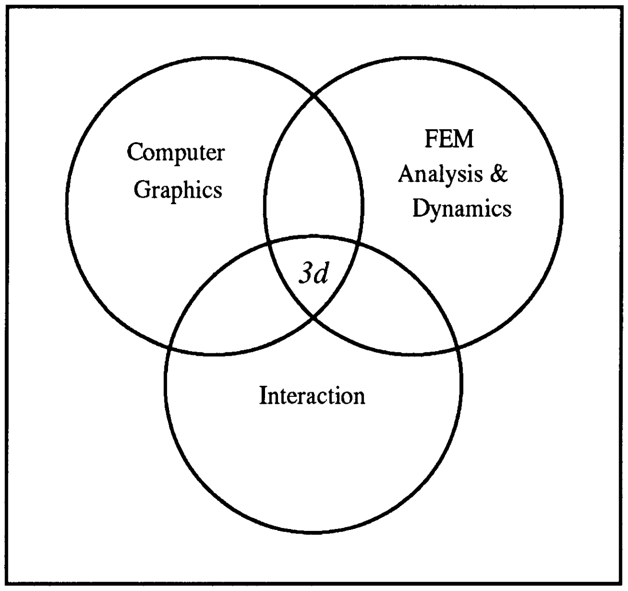


Figure 5.1: The *3d* Animation/Simulation System

gramming style is encouraged. The rest of this chapter will briefly describe the syntax of the Tcl language, the data structures that are used in *3d*, and quickly go over some of the application-specific commands that are available. Lastly, we will describe a simple interactive application defined using the system.

5.1.1 Tcl Language Syntax

This section is largely from [Ous90] and is included so that later examples will make more sense to those unfamiliar with Tcl programming. The Tcl syntax is similar to that of the UNIX shell. A command is simply one or more fields separated by blanks. The first field is the name of the command, which can be either a built-in or application-specific command, or a `tclproc` constructed from a sequence of other commands. Subsequent fields are passed to the command as arguments. A newline (\n) separates commands as do semi-colons (`;`). Every Tcl command returns a string result upon evaluation or the empty string if no return is appropriate.

Additional constructs give Tcl a LISP-like feel. Comments are delineated by a pound sign (#). The backslash character (\) denotes line continuation, or escapes special characters for insertion as arguments. Curly braces ({}) can be used to group complex list arguments. For example, the command

```
set a {leo cga {sprockets cheeba} hoot}
```

has two arguments, "a", and "`leo cga {sprockets cheeba} hoot`". This command sets the variable `a` to the string defined by the second argument.

Square brackets ([]) are used to invoke command substitution. Everything inside of square brackets is treated as a command and evaluated recursively by the interpreter. The result of the command is then substituted as the argument in place of the original square bracketed string. For example

```
mult [plus 1 2] 2
```

returns 6 because the command "`plus 1 2`" returns 3.

Finally, the dollar sign ($) is used for variable substitution. If the dollar appears in an argument string, the subsequent characters are treated as a variable name, and the contents of that variable are then substituted as the argument, in the place of the dollar sign and variable name. For example

```
set a [plus 1 2]
mult $a 2
```

returns 6 as in the previous example because the variable `a` has the value 3.

5.1.2 *3d* Data Types

There are eight primary types built into *3d.* These types are polyhedral objects (`objs`) used primarily as rendering geometries, texture-maps (`maps`) that can be applied to `objs`, Denavitt and Hartenberg joint chains (`dhc`), two-dimensional matrices (`m2d`), one-dimensional vectors of double precision numbers (`vnd`), vectors of integers (`vni`), dynamic objects (`dynobjs`), and strings (`str`). Many of the application-specific *3d* commands operate directly on these types.

For each of these primitive data types, there is a constructor command that creates the **C** data structures that are needed and a special *searchlist* file (`slfile`) that is flagged to indicate its type. The `slfile` name can be used later to refer back to the newly created structures. To illustrate, the constructor command to instance a polyhedral object is

```
io file_name o          -- instance object
```

where "file_name" is the name of an OSU file, and "o" is the name of the `obj slfile` to create in *3d.* When this is invoked at the *3d* prompt

```
3d> io /u/dead/data/cutcube cc
cc
```

an `obj` named `cc` is created from the OSU file "/u/dead/data/cutcube". To examine the names of available `slfiles`, the listing command `ls` can be used

```
3d> ls
- object        cc
```

The intent was to make the `slfile` commands look as much like the UNIX filesystem calls as possible. Directory trees can be made using the `mkdir` command. The animation state can be stored in a hierarchally organized way. Arbitrary combinations of the *3d* types can be created and stored for easy access in directories. This is the way *3d* implements *data structures* made from one or more of the primary types. The dynamic object, dynobj, is created as a directory and makes extensive use of this idea as will be seen.

5.2 Graphics and Rendering

The rendering and graphics application-specific *3d* commands are written using the RenderMatic **C** library, written by myself, Brian Croll and Alex Seiden. Using RenderMatic, it is very easy to setup a scene and control the objects, view, lights and rendering style. For example, to make a "cutcube" object spin, the **C** program is

```
#include <local/3d.h>

main()
{
int i;
OBJECT *obj;

   obj = InstanceObject("/u/dead/data/cutcube");
   View(-7., 20., -10., 0.);

   for (i=0; i<30; i++)
   {  obj_rotate(obj, 'z', 5.);
      RenderMatic();
   }
}
```

To do the same thing using *3d* is even simplier because the commands can be typed directly into the interpreter with no need for ancillary compilation and linking phases.

```
io /u/dead/data/cutcube cc
view -7 20 -10 0

for {set i 0} {$i<30} {set i [plus $i 1]} {
    ro z 5 cc
    render
}
```

5.2.1 Objects

Polyhedral objects are instanced from OSU files by the io command as seen above. An obj is made that other commands can easily access to change the color, shading, position, shape, and other graphical attributes of the underlying polyhedral object. Some of these commands follow

```
io file_name o              -- instance object
to x y z obj                -- translate object
ro axis a obj               -- rotate object
so x y z o1                 -- scale object
reo o1 ...                  -- reorigin object
co [r g b] o                -- color object
shadowo [0|1] o             -- turn shadows [on|off] for object
shade [d s e a] o           -- object shading parameters
facet o1 ...                -- facet object
smooth o1 ...               -- smooth object
centroid obj                -- object wsp-centroid
dumpo [file] o              -- dump point/polygons of object
matrix [m1 ... m16] o       -- local object matrix
phong o1 ...                -- phong shading
gouraud o1 ...              -- gouraud shading
harden o1 ...               -- harden object with local matrix
pt i [x y z] o              -- object vertex
```

5.2.2 Camera

The virtual camera used in RenderMatic and *3d* is that described by Alvy Ray Smith in [Smi84]. The world space coordinates are right-handed with x to the right, y into the screen and z up. Direct control is given to allow manipulation of the viewpoint, view-normal and view-up vectors. Convenience commands are provided to set these view vectors, the field of view, and the view window shear. Because, there is always a single active view, no constructor command is required to implement the synthetic camera. A partial listing of the view commands follows

```
view d a p r              -- set view with ``dist azimuth pitch roll''
fullview x y z a p r      -- vp = (x y z), vn = f(azimuth, pitch, roll)
lookat [o|x y z] roll     -- look at point or object
vheading                  -- current azimuth, pitch and roll
vp [x y z]                -- view point
vn [x y z]                -- view normal
vu [x y z]                -- view up
vdepth [near far]         -- view depth
fov [val]                 -- field of view
vdist [dist]              -- view distance
vwin [cu cv su sz]        -- view window
vpersp [0|1]              -- perspective or orthographic view
vsave                     -- save current view
vrestore                  -- restore to current view
```

5.2.3 Lights

Arbitrary numbers of point or spot lights can be created to illuminate a scene. Control is available to set lighting parameters such as light position and light color. For spot lights, the shade angle can be set, and the light made to cast a shadow in the software renderer. Like the *view*, *lights* are considered a global scene property and so no `slfile` is made when new lights are created. Multiple lights are differentiated by their names.

```
lcreate l                 -- create global light
lclose l                  -- delete light
llights                   -- list all lights
lon l                     -- turn light on
loff l                    -- turn light off
lcolor [r g b] l          -- light color
lpos [x y z] l            -- light position
ldir [x y z] l            -- light direction
lpointat [o|xyz] l        -- point light at xyz or object
lspot l                   -- make light a spotlight
lpoint l                  -- make light a pointlight
lshadeangle [a] l         -- light shade angle
lshadow [name] l          -- light shadow file name
lshadowdim [w h] l        -- light shadow file dimensions
```

```
lshadowden [d] 1        -- light shadow density
```

5.2.4 Rendering

There are two different rendering modes built into *3d.* The hardware renderer is written on top of the HP Starbase Graphics Library, and is used for quick visualization and interactive scene design. The software renderer is the A-buffer scan converter in RenderMatic and is used to draw final versions of scenes to videotape. The `render` command invokes one or the other of these options depending on the state set by the `hardware` command. The extra features of the software renderer include phong shading, anti-aliasing, transparency, haze, shadows, and texture and reflection mapping.

```
render                  -- render scene
hardware [1|0]          -- hardware render
doublebuffer [1|0]      -- render doublebuffered
antialias [1|0]         -- antialiased render
nopasses [num]          -- number of passes to render scene
quick [val]             -- render with obj bboxes if nopolys > val
dither [amplitude]      -- anti-contouring dither amplitude
screensize [x y x1 y1]  -- screen min and max
render2file [im al z]   -- files to create for software render
bg [r g b]              -- background rgb
hazelevel [hl]          -- haze level [0-1]
hazedist [hd]           -- haze distance [near-far]
```

5.3 Numerical Math

Much of the power of *3d* comes from the numerical math capability that it has. There are simple commands that operate on lists of numbers, like

```
3d> sqrt [plus [mult 3 3] [mult 4 4]]
5
```

Or, to make sure the cutcube object cc is centered at the world space origin,

```
to [mult -1 [centroid cc]] cc
```

Additionally, there is a 4x4 matrix stack (`ctm`) that can be used to define affine transformations for `objs` and three `slfile` types that implement a matrix math package.

5.3.1 Matrix Functions

Because most of the finite element equations developed in Chapter 3 are expressed as *matrix* relationships, it is important that *3d* include some method for handling two-dimensional arrays. To accomplish this, three new types are introduced. The `m2d` is a 2D array of double precision numbers, a `vnd` is a one-dimensional vector of double precision numbers, and a `vni` is a vector of integers. Most of the commands that follow can accept various combinations of these three types as input. The exact operation performed will depend on the organization of the input string. For example, `m2copy` can accept either two `m2ds`, two `vnds`, or two `vnis` as input. Other combinations will generate a syntax error.

```
m2create [r c] m        -- create m[r][c] of doubles
vncreate [n] v          -- create v[n] of doubles
vnicreate [n] vi        -- create v[n] of ints
m2dim [r c|n] mv        -- matrix/vec dimensions
m2dump mv               -- print all elements of matrix/vec
m2load {e1... } mv      -- load all elements of matrix/vec
m2write file mv         -- write binary image of mv to file
m2read file mv          -- read binary image of mv from file

m2ele {{r1 r2 rs} {c1 c2 cs}} [x] mv
     -- elements of matrix/vec

m2copy mv1 mv2          -- copy mv1 to mv2
m2equal mv1 mv2         -- mv1 == mv2
m2scale [mv] sca mvr    -- mv[r] * sca = mvr
m2mult mv1 [mv] mvr     -- mv1 * mv[r] = mvr
```

```
m2plus mv1 [mv] mvr      -- mv1 + mv[r] = mvr
m2minus mv1 [mv] mvr     -- mv1 - mv[r] = mvr
m2transpose [m] mr       -- transpose( m[r] ) = mr
vnlength v               -- return length of v
vnsum v                  -- return sum of elements of v
vnnorm [v] vr            -- vr = |v[r]|
vndot v1 v2              -- v1 dot v2
vncross v1 v2 v3         -- v1 cross v2 = v3
m2invert [m] m2          -- m2 = inv( m[2] )
m2lud b A x              -- find vector x such that b = Ax
m2pinsg b A x            -- find vector x such that Ax - b is minimized
m2pisg b A z x           -- find vector x such that, (b-Az) = A(x-z)
m2decompose m vi         -- LU form of m, vi is new row order
m2solve b LU vi x        -- find x given b using LU form of A, ie. b=Ax
m2improv b A LU vi no x -- iteratively improve x from solve no times
```

5.3.2 Netlib Functions

Because C does not have any explicit facility for defining two-dimensional arrays, the two-dimensional matrix functions were implemented as a subroutine library called **libm2.a**. For the harder numerical problems, it is advantageous to turn to the more established FORTRAN programs available through **netlib**. Code from **eispack** is used to find the real and imaginary eigenvalues and eigenvectors of general matrices, or the real eigenvalues and eigenvectors of symmetric matrices. Code for choelsky factorization is taken from **linpack** to solve the *generalized* eigenvalue problem as discussed in Section 3.4.2.

```
m2eig_rg m vr vi mr      -- eigenvals of m in vr/i, eigenvecs in mr
m2eig_rs m vr mr         -- eigenvals/eigenvecs of symmetric matrix m
m2chdc [m] mr            -- cholesky-decomposition( m[r] ) = mr
```

5.4 Dynamic Objects

The system of ordinary differential equations that results from a finite element analysis can be effectively solved using a numerical integrator as discussed in Section 3.4.1. Rather than writing a custom ODE solver, the LSODE code available through **netlib** is employed for this purpose. LSODE solves the initial value problem for stiff or nonstiff systems of first-order ODE's, of size *neq*, of the form

$$\frac{dy_i}{dt} = f_i = f(i, t, y_1, \cdots, y_{neq})$$

The function **f** is the user-defined **derivs** function that characterizes the exact nature of the ODE system. The FORTRAN interface to this call is

subroutine *f* (*neq, t, y, ydot*)
dimension *y*(*neq*), *ydot*(*neq*)

in which the vector function **f** loads **ydot(i)** with **f(i)**.

LSODE, written by Alan Hindmarsh, contains an implementation of the Adams-Bashforth predictor-corrector, which is the LSODE **method** that is primarily used here. The mechanism that *3d* uses to access the numerical integrator is the dynamic object `dynobj`. A `dynobj` is created by the constructor command

```
dyncreate noeqns d       -- make a dynamic object
```

where in this case, **noeqns** is the number of *second-order* differential equations to be solved. One point worth noting is that the predictor-corrector, while relatively fast, does not handle discontinuous input forces very well. In practice, the integrator must be *reinitialized* before such forces can be applied [Lot84].

The command `dyncreate` makes a directory named `d` and fills it with `m2ds`, `vnds`, and `strs` that define the dynamic state. For example, a rigid body can be described as a system of six second-order differential equations. To create a `dynobj` that can handle simple rigid body dynamics, one would type at the *3d* prompt,

```
3d> dyncreate 6 rigidbody
rigidbody

3d> ls rigidbody
- string        .dyn_xddot
s vector        x
s vector        xddot
s vector        xdot
```

where `.dyn_xddot` is a `str` that contains the name of the Tcl **derivs** function that evaluates `xddot` for `rigidbody`. The command defined by `.dyn_xddot` is in essence a *Callback* function (as used extensively in the X Window System) that defines the second-order ODE system for the dynamic object, and is automatically invoked as many times as required by LSODE when a simulation step is taken by the command `dynstep`. It is important to note that the vectors `x`, `xdot` and `xddot` are easily accessed by the matrix commands of the previous section.

Because a `dynobj` is created as a directory structure, it is easy to augment with data types specific to a dynamic situation to be simulated. For example, the solution to the FEM equilibrium equations can be written

$$\ddot{x} = M^{-1}(R - Kx - C\dot{x})$$

which indicates that `m2ds` for the stiffness matrix K, the damping matrix C and for the mass matrix M must be added to the directory `rigidbody` in addition to another `vnd` for the external force. Of course the `.dyn_xddot` Callback must also be modified to solve the FEM equations.

A partial listing of the dynamic object commands follows,

```
dyncreate noeqns d       -- make a dynamic object
dyninit d [sparse]       -- flag integrator for restart
dynstep stepsize d       -- forward simulate
dyntime [new_t] d        -- current simulation time
dyntol [rtol atol] d     -- relative and absolute error tolerance
dynmethod [meth] d       -- 10 is Adams-Basthforth [see lsode src]
dyneqs [no_eqs] d        -- get/set number of equations to be solved
lsodedebug [level]       -- set/get lsode debug level
```

5.5 User Interface

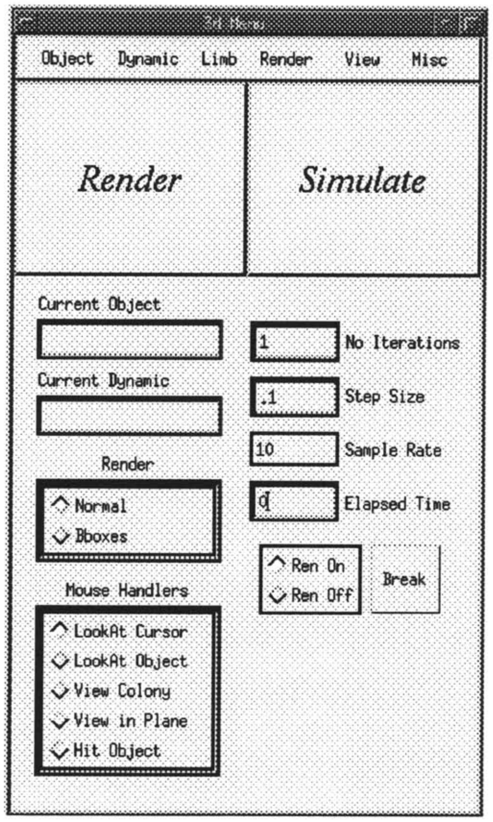
The Tcl language was originally written to be integrated with the widget library of a window system, and can serve two purposes in such a context. These are to configure the *actions* of an application's interface, and to design the *appearance* of that interface. Tcl is used for both purposes by *3d*. The widget set used is OSF/Motif™ and the windowing system is HP's port of X. The user interface is currently defined by three kinds of dialog – through the keyboard, through interactive Dialog Boxes or direct mouse manipulation of objects inside the graphics window. Figure 5.2 shows the primary Motif dialog box used in *3d*.

5.5.1 Motif Widgets

Most of the Motif constructors have been imported as application-specific commands into *3d*. These commands have a very uniform calling sequence as in

```
XmCreateScale parent name args no   -- returns new widget
```

which creates a slider widget. When such a widget is constructed, a hexadecimal pointer address is returned that can be parsed by the other X/Motif interpreter commands. Motif resources can be accessed as documented in [Fou90] and [You90]. *Actions* are defined through the X *Callback* mechanism. A Callback can be any *3d* command or `tclproc`. As



Current Object

Current Dynamic

1	No Iterations
.1	Step Size
10	Sample Rate
0	Elapsed Time

Render: Normal, Bboxes

Mouse Handlers: LookAt Cursor, LookAt Object, View Colony, View in Plane, Hit Object

Ren On, Ren Off

Figure 5.2: *3d* dialog box

a simple example, a small program that creates a PushButton with a simple Callback is presented. This code relies on the tclproc Xmap that allows simple list passing of up to 32 arguments to a widget constructor.

```
proc Xmap { func parent name arglist } {
    set args [argcreate 32]
    set no 0
    foreach i $arglist {
        XtSetArg $no [index $i 0] [index $i 1] $args
        set no [plus 1 $no]
    }
    set widget [$func $parent $name $args $no]
    free $args
    return $widget
}
```

As with all Xt programs, the Toolkit must be initialized before widgets can be created. A RowColumn widget is made as a child of the Toplevel shell, and a PushButton is made that reads "Push Me". The string "Yow!" is echoed when the button is pushed.

```
set top [XtInit Test {}]
set widget [XmCreateRowColumn $top {} {} 0]

Xmap XmCreatePushButtonGadget $widget {} {
    {labelString {  Push Me  }}
    {activateCallback {echo Yow!}}
}
XtRealize $top
```

5.5.2 X Events

The X mouse and window events are also available to *3d*. Callbacks can be setup on a window to listen for mouse clicks, keyboard presses, cursor motion and resize events. The Callback function is handed a list that contains the event information as its argument. In the following example, a window is made and mouse button pushes are requested. In this case, the Callback simply echos the EventList as it is received.

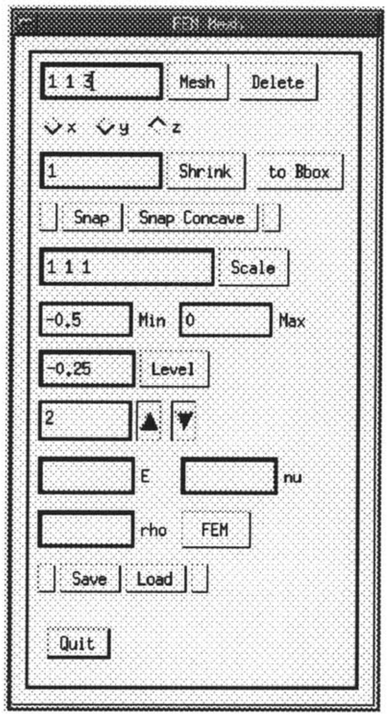
```
set top [XtInit Test {}]
set widget [Xmap XmCreateForm $top {} {{width 250} {height 250}}]
XtRealize $top

set win [XtWindow $widget]
EventCallback $win 5 { echo }
```

5.6 User-Assisted Finite Element Mesh Construction

An example interactive application written using the *3d* system is a user-assisted FEM mesh generator. Totally automatic mesh generation is non-trivial, and is especially hard for a full 3D case [Cha88] [CB89]. Here, the approach taken to this problem is to provide the user with an interactive tool that allows simple construction of finite element meshes that fill a volume defined by polyhedral objects. No effort was made to calculate error indicators that could quantify the quality of the generated mesh as done in [CB89], but in general, a better mesh is produced by using more elements and adaptively fitting the elements to capture the details of the goal geometry. A mesh constructed with the technique implemented in the thesis can be of arbitrary size, but the larger the number of elements used, the slower the dynamics will run. These kind of tradeoffs are discussed in Chapter 3. In fact, it is easy to design a mesh too big for the current simulation system. The dialog box created for the mesh generation process is shown in Figure 5.3.

The steps to make a finite element mesh for the polyhedral Cyberware head model from Section 2.3.1 will be presented. The element used is the twenty-node brick discussed in Section 3.2. For the purposes of computer animation, it is sufficient to coarsely sample the head with only three isoparametric cubes, one for the shoulders, one for the neck and



1 1 3

1

1 1 1

-0.5 Min 0 Max

-0.25 Level

2

Figure 5.3: FEM mesh dialog box

one for the rest of the head. In this arrangement, two element faces are shared, so the number of degrees of freedom to be simulated will be $3 * (3 * 20 - 2 * 8) = 132$. The starting point for the mesh generation process is to fill the volume defined by the bounding box of the Cyberware model with the number of elements set in the *Mesh* text area of the dialog box, see Figure 5.4.

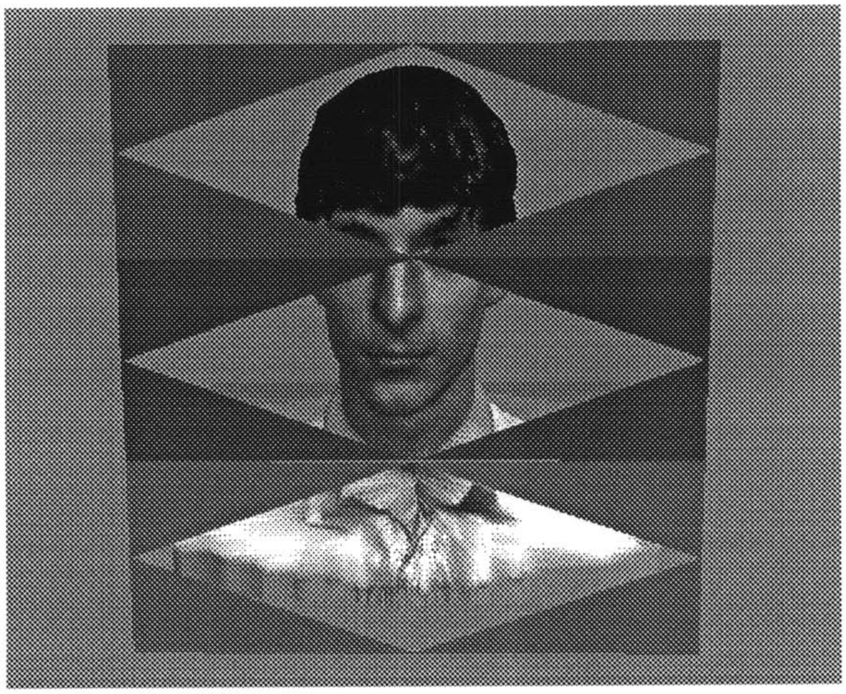


Figure 5.4: Mesh filling Cyberware head bounding box

The z axis is chosen to represent the natural orientation of the object. In this direction, there are seven contour levels defined by the nodal point z values. These contour levels can be easily adjusted through the interface of Figure 5.3. For the Cyberware head, levels two and four are moved so that the middle isoparametric brick encompasses the neck area. The *Snap* push-button recenters the edge midpoints after the element corners are shifted. This intermediate mesh is shown in Figure 5.5.

The points of the mesh are then cylindrically *shrinkwrapped* around the z axis to

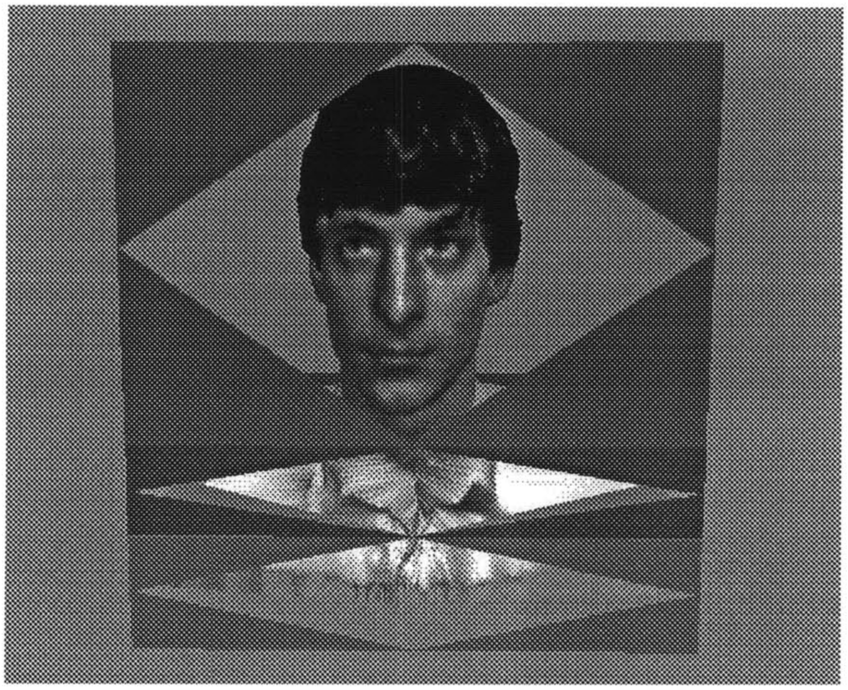


Figure 5.5: Mesh boundaries interactively shifted

intersect the polygonal object. This is done by calculating a *contour* from the points of the polyhedral data for each z level and drawing the outside mesh points in to intersect the contour. This produces the finite element approximation to the Cyberware head. Because the mesh points can initially be relatively distant from the polyhedral model, the procedure is done in two stages. First, the mesh is shrunken only to the bounding boxes defined by the local contours with the *toBbox* push-button. Then the intersection with the contour defined by the geometry data is done. The mesh after shrinkwrapping to the local bounding boxes is shown in Figure 5.6. After shrinking, it is necessary to recenter those element midpoints that make a concave edge with *SnapConcave*, or else a malformed stiffness matrix K may be found as described in [Bat82].

After the finite element approximation has been completed, the mesh information can either be saved to disk, or simulation parameters can be set for running dynamic trials.

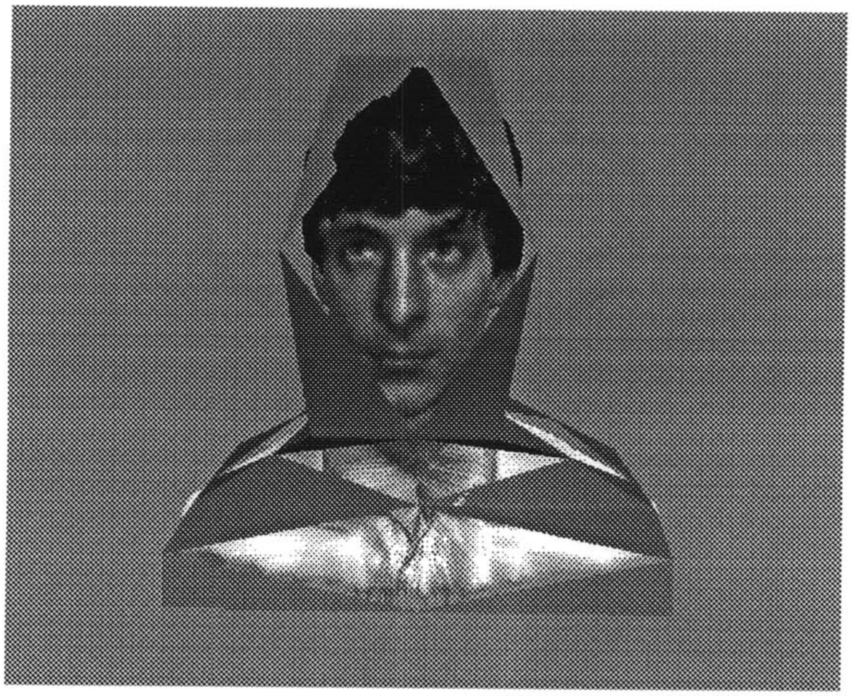


Figure 5.6: Mesh shrunken to head

The parameters Young's modulus E, Poison's ratio ν, and the mass density ρ can be fixed identically for each of the constituent elements, or else set individually to model an inhomogeneous composite material. A `dynobj` is then constructed by calculating the matrices K and M and by performing the modal transformation as in Chapter 3. Furthermore, the polyhedral head is embedded in the isoparametric mesh by performing the local to natural coordinate transformation described in Section 3.2.2. To test the behavior of the finite element model, the Modal dialog box of Figure 3.10 is used to examine the mode-shape deformations defined by the eigenvectors ϕ_i as in Figure 5.7.

This simple mesh construction tool was written in approximately 320 lines of Tcl code. An additional 550 lines were needed for the X interface. Application-specific routines written in **C** were also added to *3d* to calculate the shrinkwrapping contours from the input `obj` and to shrink the mesh points as required, and to do the element midpoint recentering

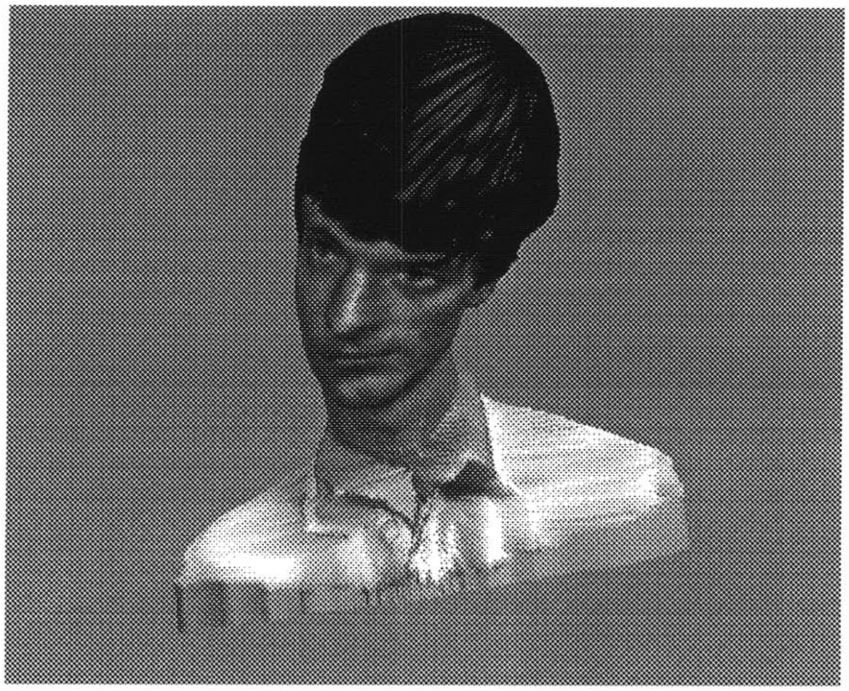


Figure 5.7: Head warped through mode-shape deformation

operations described. In total, with the existing **C** libraries, these routines took only about 600 new lines to implement.

5.7 Validating the FEM Implementation

Before proceeding further, an attempt is made to validate the FEM implementation by comparing simulation results to the analytic solution for an idealized beam. In the next chapter, a similar set of trials will be run to validate the *biomechanical* implementation by reproducing some of the experiments done on whole muscle that lead to the development of Zajac's muscle model.

The problem to be considered is a simple beam subjected to a uniform gravity force. The beam is 6 inches long, 2 inches high and 2 inches wide. The beam is made to have the density of water, which in the English units is .00112287 slugs/in^3, and so weighs .868 lbs on Earth. The acceleration of the Earth's gravity is -386 in/s^2.

The other physical parameters to set are the mechanical material properties of the beam. For a linear, homogeneous, isotropic material, this is done in terms of E and ν. E is Young's modulus and is simply the ratio of stress to strain. The Young's modulus chosen here is 435.11 psi, which approximates some kinds of rubber. For comparison, steel has modulus of elasticity $E = 3 \times 10^7$ psi. Poisson's ratio, ν, is the ratio of lateral contraction to longitudinal extension and so is a dimensionless quantity. A value of $\nu = .5$ is used for materials that are volume preserving. The Poisson's ratio for steel ranges from 0.25 to 0.33, for rubber it is slightly less than 0.50 [Har49]. Note that a look back to Equation 3.3 indicates that a Poisson's ratio of exactly 0.50 will cause a divide by zero and so is not possible. Volume preserving materials are then approximated with ν close to, but not equal to 0.50.

Initially, the beam is undeflected with no loading. At time 0, gravity is turned on and the beam deforms in response. The solutions for the deflection of cantilever beams under various loading conditions are well known. Analytic formulations from [Har49] are used as the control for this experiment. These solutions are based on the assumption that the cross-sectional dimensions are small compared with the beam's length. Consider the beam in Figure 5.8 subjected to a uniform load w all along its length l. At a point x, which is distance $l - x$ from the free end, the bending moment is $w(l-x)^2/2$. The deflection at the free end is written,

$$\delta = y_{x=l} = \frac{wl^4}{8EI}$$

For a beam with rectangular cross-section, I is given by

$$I = \frac{bh^3}{12}$$

where b is the width and h the height of the cross-section. In this case, $b = h = 2$ and so

$I = \frac{4}{3}$.

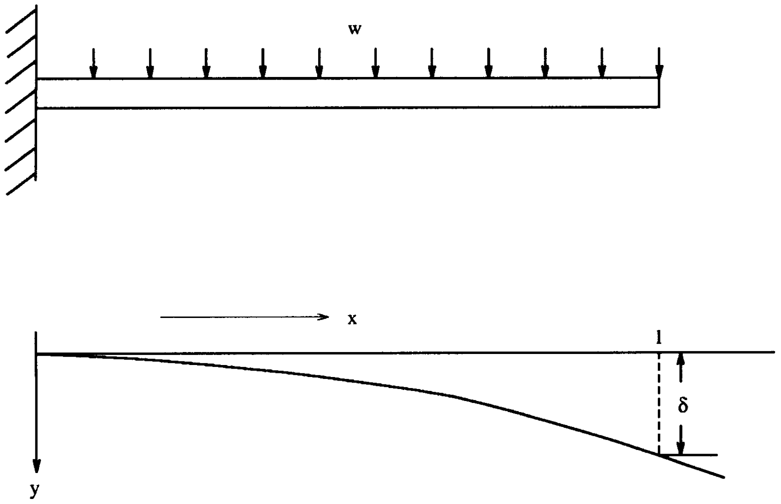


Figure 5.8: Uniformly loaded cantilever beam. *adapted from* [Den Hartog]

The load w is simply the force per cross-sectional slice of the beam, and is found by calculating the total force due to gravity and dividing by the length of the beam. The total mass is $24 \times .00112287 = .02694888$ slugs,

$$w = -386 \times .02694888/6 = 1.73371128 \, \mathrm{lb}$$

And so the analytic solution for the amount of deflection at the tip of the beam is,

$$\delta = \frac{1.73371128 * 6^4}{8 * 435.11 * 1.3333} = -.484121076 \, \mathrm{in}$$

For experiment, two different finite element discretizations of the beam are simulated, one with three elements and the second with six elements. The particular element used is the twenty-node isoparametric brick discussed in Section 3.2. After matrix assembling, the three element beam has 44 nodes and 132 degrees of freedom. The six element beam has 80 nodes and 240 degrees of freedom. The effect of mode truncation is examined by performing a number of trials with different numbers of modes for each beam. Figure 5.9 plots the eigenvalues for the two beams.

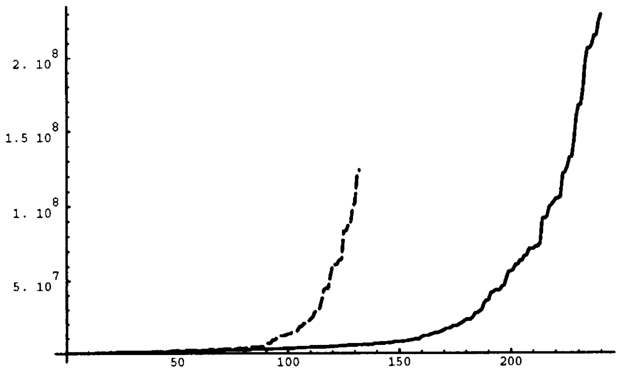


Figure 5.9: Plot of eigenvalues for 3 and 6 element beams. 3 element plot is dashed, 6 element plot is solid

For the three element beam, three separate simulation trials are run for the deflection, once with 30 modes, once with 100 modes, and finally with all 132 modes. The tabulated statistics are, the total number of steps taken by the LSODE integrator for half a second of total simulation, the number of **derivs** function evaluations needed, the total real time required for the simulation, and the beam deflection at time .5 seconds.

no. of modes	frac. of total	LSODE steps	func. evals.	evals. per step	simulation time (mins)	deflection (inches)	percent error
30	.227	16791	30356	1.81	45	.3046	37
100	.758	16904	30681	1.82	63	.4351	10
132	1.	16695	30244	1.81	70	.4493	7.2

Table 5.1: Runtime and tip deflection for 3 element beam.

As expected, the error for the deflection goes down as the number of modes increases. Also note that the FEM results are all stiffer, or represent an *upper-bound* to the analytically predicted results. The time course of the tip deflection is plotted in Figure 5.10.

For the second trial, a six element beam is used instead of the three element beam.

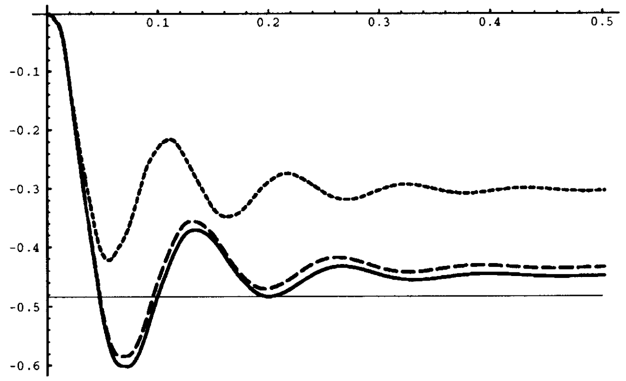


Figure 5.10: Deflection for 3 element beam. Length vs. time. 30 mode plot is dotted, 100 mode plot is dashed and 132 mode plot is solid.

The same behavior as regards the number of modes is expected, but the final results should better approximate the analytic deflection values.

no. of modes	frac. of total	LSODE steps	func. evals.	evals. per step	simulation time (mins)	deflection (inches)	percent error
30	.125	16073	28996	1.80	56	.3304	32
54	.225	15716	28311	1.80	61	.4035	17
100	.417	17047	30996	1.82	90	.4525	6.5
182	.758	17191	31306	1.82	119	.4792	1.0
240	1.	23142	24635	1.06	117	.4881	.82

Table 5.2: Runtime and tip deflection for 6 element beam.

Again, the final deflection is better approximated by the simulation with the most modes. The deflection for this case is plotted in Figure 5.11.

The cost in calculating the **derivs** function for this problem is dominated by the matrix multiplications for the modal forces, $R_{modal} = \Phi^T R$, and updating the nodal positions, $u = \Phi x$, as discussed in Section 3.4.4. For the three element beam, Φ has rank 132 and the

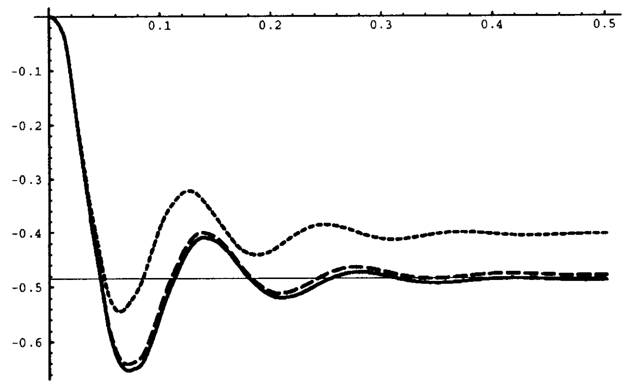


Figure 5.11: Deflection for 6 element beam. Length vs. time. 54 mode plot is dotted, 182 mode plot is dashed and 240 mode plot is solid.

matrix multiplies take about .023 seconds each. For the six element beam, Φ has rank 240 and the large matrix multiplies take roughly .081 seconds each. The initial and final states for the 6 element beam used in the experiment are shown in Figure 5.12 and Figure 5.13.

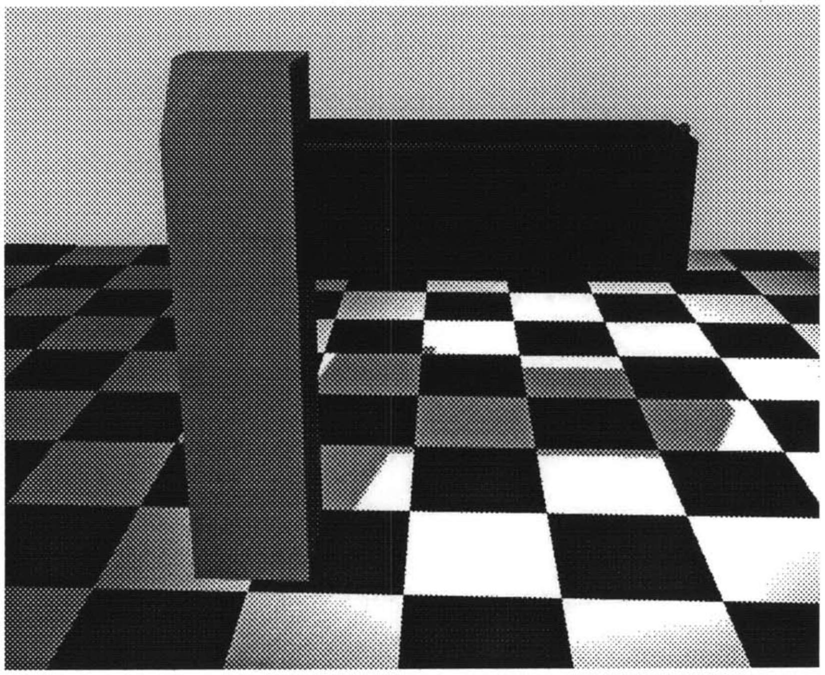


Figure 5.12: Beam made from 6 twenty-node elements, initial state

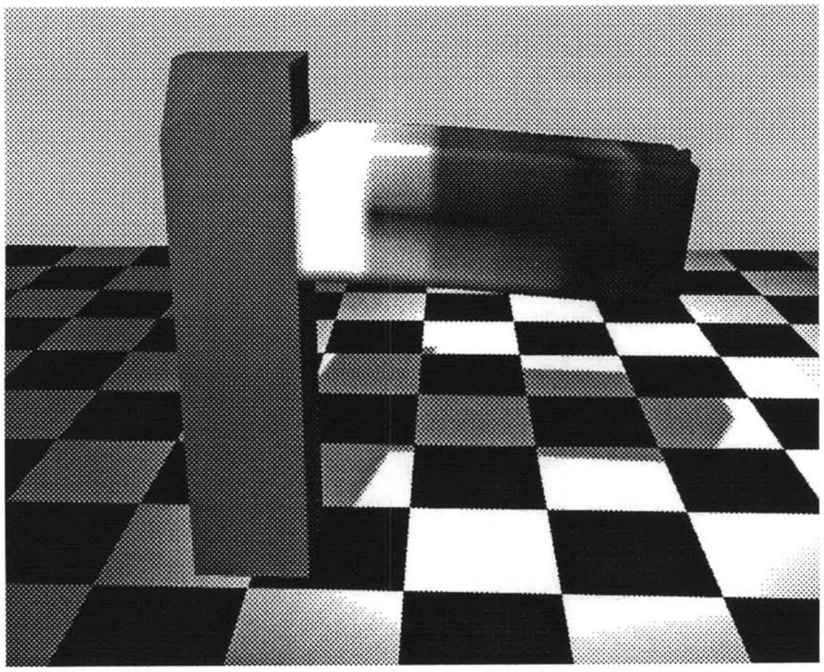


Figure 5.13: Beam made from 6 twenty-node elements, final state

Chapter 6

Methods and Results

6.1 Frog Gastrocnemius Simulation

Models that simulate the action of muscle on a wide variety of levels have been proposed for a wide variety of purposes; the model that is implemented clearly depends on the desired objective. For instance, much research is being done to investigate the fundamental physical and chemical mechanisms of the actin-myosin crossbridge attachments within muscle fibers. Individual fibers can be extracted from frogs, subjected to periodic displacement changes and the visco-elastic properties measured as in [CGT86] and [JBDT88]. For this research, Blangé and Stienen have proposed a model of single muscle fibers as an infinitesimally thin rod composed of a repeating sequence of springs and damping elements, see Figure 6.1. Alternately, engineers who design artificial limbs to improve or replace lost motor function often make calculations of muscle force based on kinematic body movements and on measured EMG activity. This kind of muscle model was discussed in Section 4.2 in reference to the work of Wood.

Zajac in [ZTS86] and [Zaj89] discusses a *dimensionless* musculotendon model that

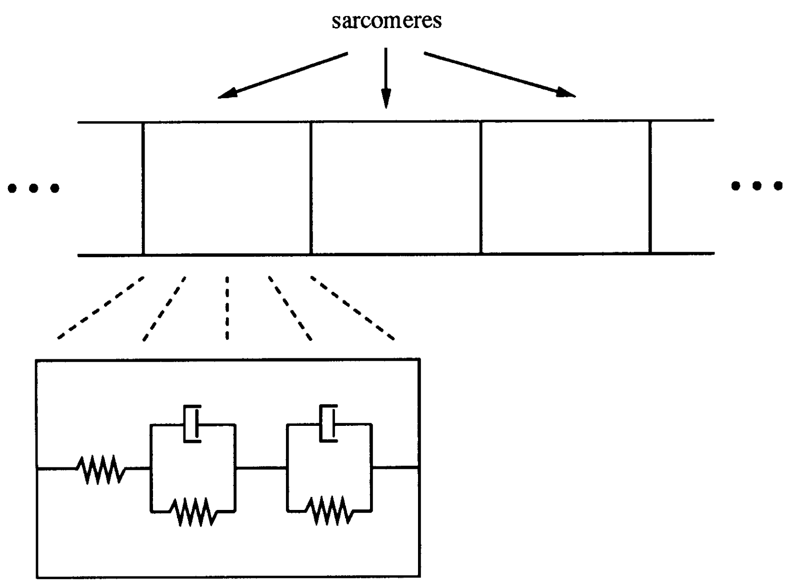


Figure 6.1: Model of single muscle fiber

is a refinement of the classic Hill model, appropriate for studies of intermuscular coordination. Details of my implementation of this model can be found in Section 2.2.3. Delp in [Del90] uses Zajac's muscle functions in a *surgical simulation* system that calculates the *static* effects of tendon transfer surgery. When human limbs are impaired, motor function can sometimes be recovered through surgical reconstruction. These reconstructions, however, often compromise the capacity of muscles to generate the amount of force needed to coordinate natural body activity. Surgeons typically rely on experience to make decisions about a given procedure. Delp has created a tool that can *pre-operatively* evaluate the effect of a procedure on muscle function. The result is that the computer *model* can assist in understanding the biomechanics of a reconstructive surgery. Computer graphics is an essential part of computer models for surgery simulation or *BioCAD* systems—both to visualize the 3D musculoskelatal geometry and to facilitate communication between engineer and surgeon.

When doing computer animation, we are primarily interested in simulation for the purpose of automatically generating changes in shape and position. A major supposition of this thesis is that in the case of graphically simulating a contracting muscle, it is possible to produce convincing shape changes by accurately simulating the forces involved. Furthermore, through simulation we will be able to analyze which elements of the muscle model are important factors in determining the shape. A natural application for such a model would be in a surgical simulator such as the one developed by Delp.

A major difference between the muscle model proposed here and the ones discussed above is that our's must be truly 3-dimensional to make computer animations of human characters. Other muscle models are essentially one-dimensional—Delp used simple straight lines to visualize the muscle origin and insertion points. Clearly a 3D model is a requirement for computer graphics, but we will also see that there may be biomechanical advantages because of the extra complexity and realism of the model.

Because our goal requires us to model a complex, dynamic, irregular, elastic volume, the underlying formulation is based on the finite element method discussed at length in Chapter 3. In that chapter, we discussed how to make linear deformable visco-elastic models from data represented polyhedrally, how forces can be applied to the node points of the resulting FEM mesh, and how to forward simulate these models through the use of numerical differential equation solvers. In this way, we argue that we have met the twin goals of computer animation and creating a biomechanically valid simulation of muscle.

6.1.1 Force-based Finite Element Muscle Model

The muscle model was constructed by dissecting the gastrocnemius from an anesthetized frog, measuring the top and side dimensions, then Swivel was used to make a polyhedral model with 576 polygons, see Section 2.3. The user-assisted finite element mesh generator described in Section 5.6 was then used to interactively construct the mesh shown in Figure 6.2. Four twenty-node isoparametric bricks are used to approximate the gastrocnemius. The exact number of elements used represents a tradeoff of the quality of the simulation versus simulation time as discussed in Section 5.6. The model then has 56 nodes or 168 total degrees of freedom.

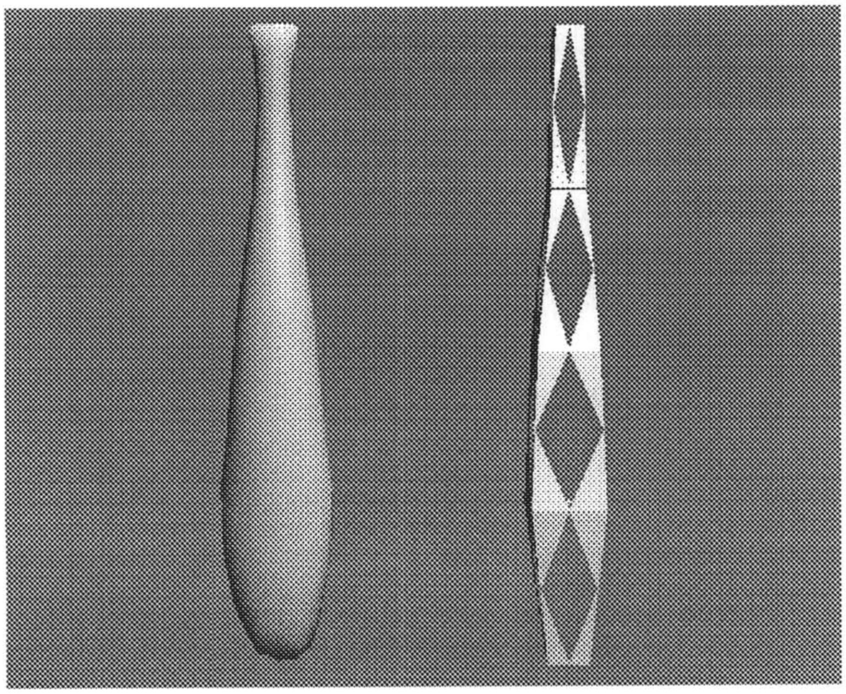


Figure 6.2: Finite element mesh used for frog gastrocnemius

To simulate a contraction, force generators are added to the node points of the finite element mesh that act along the longitudinal direction of the muscle. Wired in this way, there are eight generators per twenty-node brick, for a total of 32 for the whole muscle. Tendons are constructed in a similar fashion, see Figure 6.3.

To facilitate the simulation, the `dynobj` data structure from Section 5.4 is augmented with information for the muscle state. Using the notation of Section 5.4, for the muscle fibers, we add

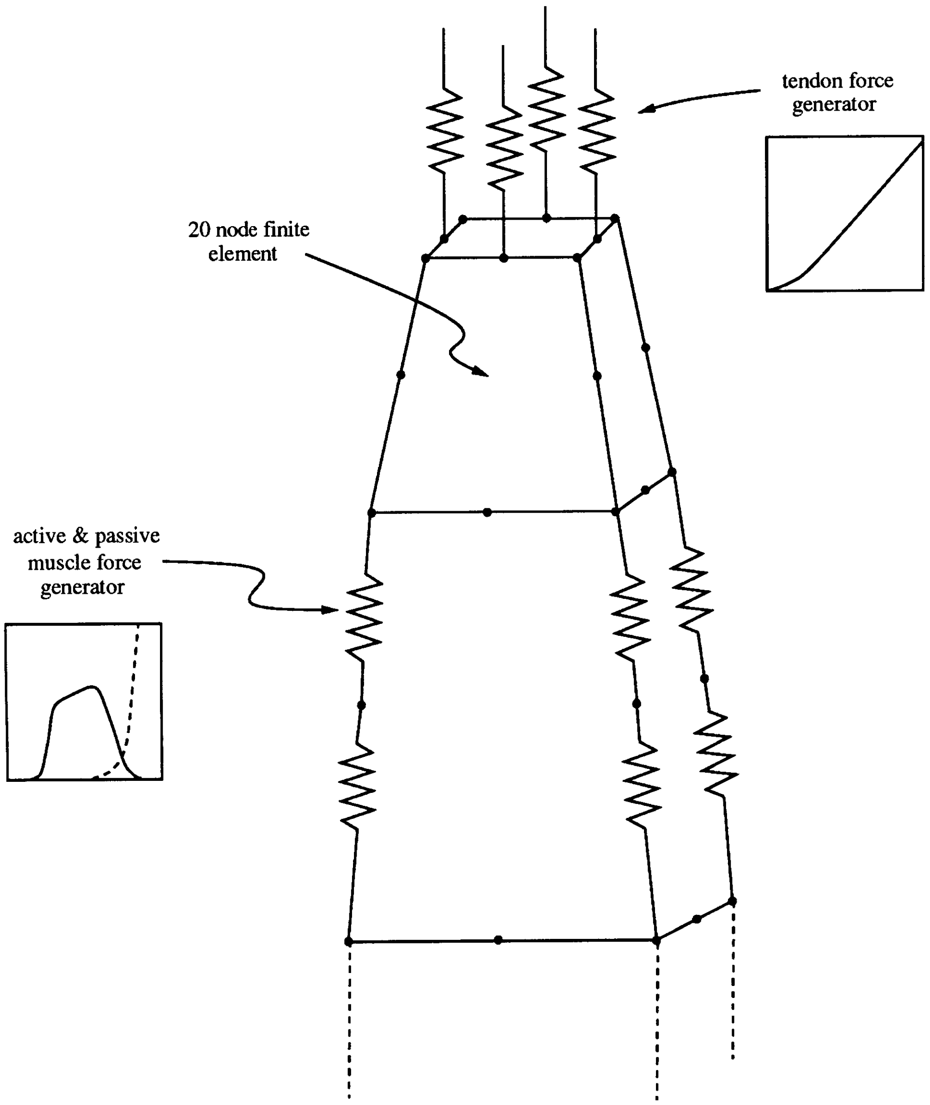


Figure 6.3: Force generators for muscle and tendon

1. `fibers`, integer vector `vni`, of length (nofibers*2): encodes which nodes are attached
2. `fiber_len0`, double vector `vnd`, of length (nofibers): fiber rest length
3. `fiber_len`, double vector `vnd`, of length (nofibers): current fiber length
4. `fiber_time`, string `str`: simulation time for which `fiber_len` was calculated

In addition, three other `str` parameters define the muscle, as per Zajac's dimensionless model. The maximum isometric active force is `fiber_F0a`, the passive force scalar is `fiber_F0p` and the maximum normalized fiber velocity is `fiber_Vmax`.

To calculate the force within each of the muscle fibers to be applied to the finite element mesh nodal points in the **derivs** function, an application-specific routine `muscle_zajacF` was added to *3d*,

```
muscle_zajacF F0a F0p Vmax wspF d
     -- Add active and passive muscle force to ‘wspF’ using Zajac model
```

which accepts the maximum active and passive forces and the maximum normalized fiber velocity as inputs. The world space force vector `wspF` ($R(t)$ from Section 3.4.4) for the dynamic object `d` is accumulated with the muscle fiber forces. These fiber forces are calculated by `muscle_zajacF` as follows.

For the four element approximation to the frog gastrocnemius in Figure 6.2, named `gastroc`, the fibers are numbered,

```
3d> m2dump /gastroc/fibers
0 16 1 17 2 18 3 19 16 4 17 5 18 6 19 7 20 28 21 29 22 30 23 31 28 0
29 1 30 2 3 1 3 32 40 33 41 34 42 35 43 40 20 41 21 42 22 43 23 44 52
45 53 46 54 47 55 52 3 2 53 33 54 34 55 35
```

where nodes {`0 16`} define the first muscle fiber, {`1 17`} the second, and so on.

The current world space mesh nodal positions are found for each fiber. These are subtracted to yield a world space length. From this the normalized fiber length $\tilde{l}^M$ is,

```
lm = len / vnd_get( fiber_len0, i );
```

The normalized length is used to find the isometric active and passive force functions, and their derivatives, as discussed in Section 2.2.3,

```
zajac_passiveforce( lm, &Fmp, &dFm );
zajac_activeforce( lm, &Fma, &dFm );
```

The fiber velocity is determined through the following first-order approximation. Before the next call to `dynstep`, the lengths for each of the muscle fibers are found and saved in `fiber_len`. The simulation time before the next call to `dynstep` is stored in `fiber_time`. When `dynstep` is then used to take the next step, the instantaneous simulation time is accessed by `muscle_zajacF` as

```
DynObject_GetTime( d, &time );
```

and the normalized velocity $\tilde{v}_r^{CE}$ computed,

```
if (time == fiber_time) velocity = 0.;
else
{   velocity = (len - vnd_get(fiber_len,i)) / (time - fiber_time);
    velocity /= vnd_get( fiber_len0, i );
    velocity /= Vmax;
}
```

from which the scale factor from the force-velocity curve is looked-up,

```
zajac_forcevelocity( velocity, &Fvc );
```

and the force generated by the fiber from the normalized quantities is,

```
force  = Fmp * F0p;
force += Fma * F0a * Fvc;
force *= .5;
```

The final scale by .5 is done because half the force is applied to the first node point of the fiber, and the other half of the force is summed with the second fiber node point. If the world space vector that defines the fiber orientation is d1, and in1 and in2 are the two fiber indices, then the world space fiber force is accumulated into wspF as,

```
VVAscale( d1, force );

vnd_setplus( wspF, in1*3 + 0, d1[0] );
vnd_setplus( wspF, in1*3 + 1, d1[1] );
vnd_setplus( wspF, in1*3 + 2, d1[2] );

vnd_setplus( wspF, in2*3 + 0, -d1[0] );
vnd_setplus( wspF, in2*3 + 1, -d1[1] );
vnd_setplus( wspF, in2*3 + 2, -d1[2] );
```

The tendon fibers are modeled in much the same way as the muscle fibers, the main difference being that tendons connect mesh node points to external world space locations. For tendon, the dynobj data structure is further augmented with,

1. tendons, integer vector vni, of length (notendons): encodes which nodes are tendon attached
2. tendon_len0, double vector vnd, of length (notendons): the tendon rest length
3. tendon_wsppt, double vector vnd, of length (notendons*3): the world space attachment points
4. tendon_force, double vector vnd, of length (notendons*3): the amount of force generated by the tendon fibers

There is also a `str` that scales the normalized tendon force, `tendon_F0`. The force generated by each of the tendon fibers, is found by the application-specific routine `tendon_zajacF`.

```
tendon_zajacF F0 wspF d -- Add tendon force to 'wspF' using Zajac model
```

which accumulates the tendon fiber forces into the world space force vector `wspF`. These forces are calculated by `tendon_zajacF` as below.

For the four element approximation, `gastroc`, the tendon fibers are placed at the edge midpoints of the bottom and top faces of the mesh assemblage,

```
3d> m2dump /gastroc/tendons
49 51 48 50 13 15 12 14 0 3 11 20 23 27 32 35 39
```

To calculate the tendon force, begin with the world space location of the node to which the tendon fiber is fixed. Subtract the position of the external attachment point defined by `tendon_wsppt`, and find the length `len` and direction of the resulting vector `d1`. Using the rest length of the current tendon fiber, the strain ε^T is,

```
strain = (len - len0) / len0;
```

Then the strain is used to look up the normalized tendon force. The vector `d1` is scaled by the force,

```
zajac_tendonforce( strain, &Ft, &dFt );
force = Ft * F0t;
```

This is repeated for each tendon fiber and again the computed force is summed into the world space external force vector `wspF` for the finite element mesh. In addition, the force generated by each of the tendon fibers is saved in the vector `tendon_force`. This is because the total force generated by the muscle is the sum of the reaction forces created by the tendon attachments. This total force is, of course, an important measure and is one the quantities plotted in the subsequent experiments.

The two application-specific routines muscle_zajacF and tendon_zajacF are invoked by the standard **derivs** function in *3d* for the modally transformed finite element equations. The name of the *Callback* function that defines the effect of the muscle and tendon fibers is set within the gastroc structure,

```
sset $d/.dyn_forces muscle_contract
```

and the .dyn_forces Callback is invoked within **derivs** as

```
foreach f [sset $d/.dyn_forces] { eval $f $wspF $d }
```

where the Tcl procedure muscle_contract is

```
proc muscle_contract { wspF d } {
    set F0a [mult [sset $d/fiber_F0a] [sset $d/.activation]]
    set F0p [sset $d/fiber_F0p]
    set Vmax [sset $d/fiber_Vmax]
    set F0t [sset $d/tendon_F0]

    muscle_zajacF $F0a $F0p $Vmax $wspF $d
    tendon_zajacF $F0t $wspF $d
}
```

The effects of the muscle activation function $a(t)$, directly scales the maximum active fiber force fiber_F0a. This is the mechanism used to turn the muscle on and off.

To test the biomechanical validity of the muscle model developed here, two well-known experiments are simulated. First, tension-length curves will be plotted of both active and relaxed muscle. Second, Gasser and Hill's quick-release experiment will be simulated to reveal the dynamic time course of the forces generated by the finite element muscle model.

6.1.2 Tension-Length Experiment

The first experiment plots the characteristic tension-length relationship produced by the FEM muscle model. This is easy to do by attaching the top set of tendons to various positions to control the overall muscle length, measuring the amount of force generated by the whole muscle, then fully activating the muscle and measuring the force again. The setup for this experiment is illustrated in Figure 6.4. The muscle is attached with tendon to both the bone and the horseshoe shaped "clamp" in the figure.

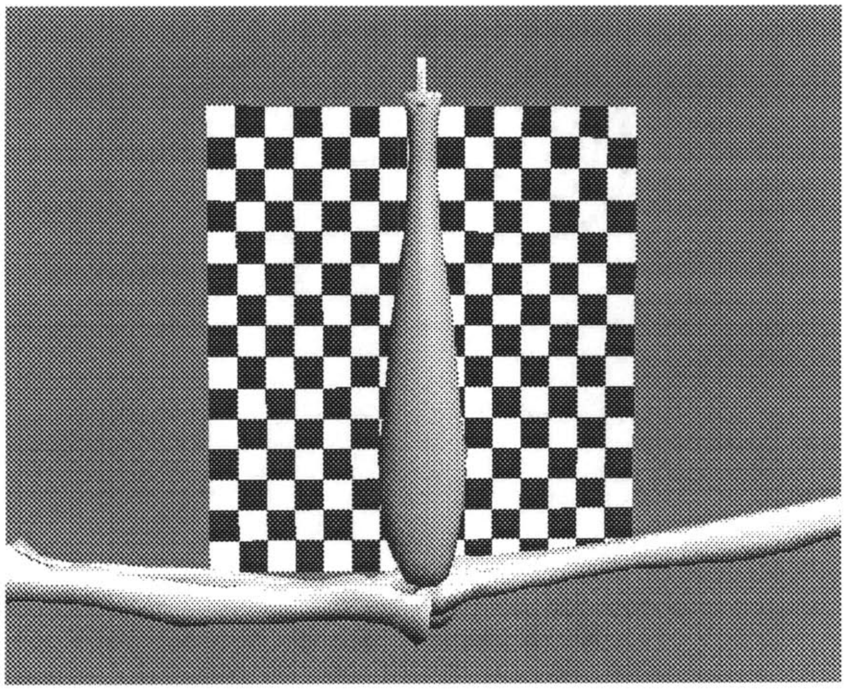


Figure 6.4: Setup for tension-length experiment

Up to now, most of the development has concerned computing the non-linear muscle force functions defined by biomechanical models to apply at the finite element mesh nodal points. To fully specify the FEM model, however, the passive mechanical characteristics of muscle must also be set. Grieve and Armstrong in [GA88] publish stress-strain curves derived from compressing plugs taken from pig muscle. These curves are also non-linear, the plugs becoming stiffer the more they are compressed. The range of values for Young's modulus they present are from close to 0 up to 2.745×10^3 Pa at a strain of 40%. Or, in **cgs** units, 2.745×10^4 dyne/cm^2.

A major assumption made in the thesis, is that in terms of these passive elastic properties, muscle can be approximated as a linear, homogeneous, isotropic material. This assumption is necessary because of simplifications inherent in the derivation of the stiffness matrix in Section 3.1.1, but could be easily relaxed by implementing better constitutive models [Bat82]. In any case, I feel this simplification is justified when examining muscle contraction because the contraction forces are orders of magnitude larger than the passive mechanical forces, and these non-linear, anisotropic forces are indeed modeled. An intermediate, convenient value is then chosen for E. Poisson's ratio, ν, is set to approximate a volume preserving material. The density, ρ, of muscle was found by Grieve and Armstrong through careful weighing. Gravity, g, is turned off so as not to confound the force measurements. The physical constants for the simulation are as follows,

$$\mathrm{E} = 1000\ \mathrm{dyne/cm^2} \qquad \nu = .49 \qquad \rho = 1.04\ \mathrm{g/cm^3} \qquad \mathrm{g} = 0\ \mathrm{cm/s^2}$$

The frog muscle rest length is 6 cm long. The maximum isometric force generated by the gastrocnemius measured in the frog lab (Section 2.3), was 2.77 N or 2.77×10^5 dynes. This force is distributed evenly among the 32 fibers and so the fiber force is 8656.2 dynes.

The tension-length experiment is performed by first measuring the force for the passive muscle. This is done by examining the force generated in the tendons attached to the clamp through the `tendon_force` vector. The muscle is activated and the force measured again. The whole process is repeated for the muscle set to different initial lengths. The resulting tension-length curves are presented in Figure 6.5.

An examination of this plot shows a good correspondence to the published biological

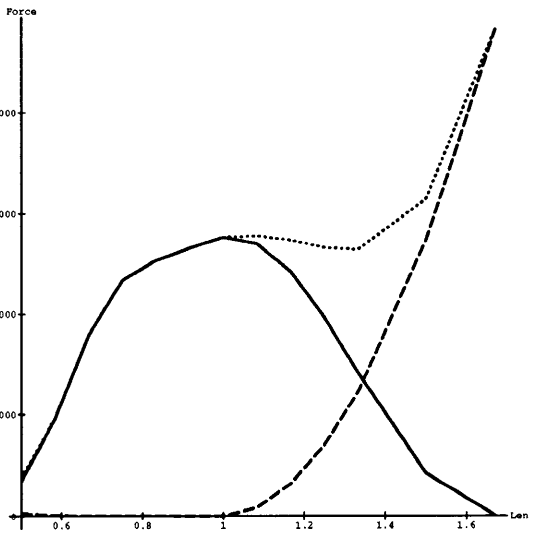


Figure 6.5: Simulation results for tension-length experiment. Isometric muscle force vs. normalized length. Dotted line is total isometric force, dashed line is the force from passive muscle and the solid line is the developed force.

observations (see Figure 2.10), with a local maximum for the muscle at its rest length. The other result taken from this simulation, interesting from the point of view of computer graphics, is that very little changes in shape are produced by a muscle undergoing purely isometric contraction. Another of the simplifying assumptions made in the muscle model is that all the fibers are homogeneous both in terms of the amount of force they can develop, and in their response to stimulus from the neural controller ($u(t)$ in Figure 2.37). The small observed shape change is then expected since the actions of all the series and parallel fibers should produce a net zero resultant force at the internal mesh faces, while producing a large force at the origin and insertion ends which is then cancelled by the tendon forces. Hence, to produce larger shape changes for the purposes of computer animation requires either the muscle to lengthen or shorten, or for the contraction to work against shape changes due to external forces such as gravity.

6.1.3 Quick Release Experiment

The second experiment simulated is the quick release procedure carried out by Gasser and Hill to examine the velocity dependent effects within the muscle. The muscle is stimulated and made to work isometrically. Then the muscle is suddenly released and hits a 'knot' or new position constraint after a certain amount of time. A plot of force versus time for this experiment is presented in Figure 6.6. The muscle begins at the rest length, 6 cm or $\tilde{l}^M = 1$. The amount of shortening that takes place is 1.3 cm so that the final normalized muscle length is $\tilde{l}^M = 0.7833$.

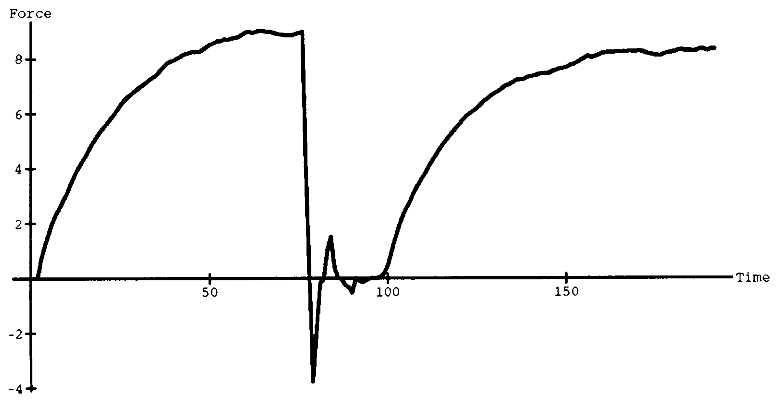


Figure 6.6: Simulation results for Gasser-Hill quick release experiment. Force vs. time. Force is scaled to match plot of Figure 2.12.

Figure 6.6 shows a good correspondence with Gasser and Hill's plots of Section 2.2.2. In that section, the conclusion drawn from the shape of the curves is that the slow rise in the force both when the muscle is activated, and after the quick release, indicate that it is caused by a biochemical damping effect, rather than a central nervous system control mechanism. Furthermore, I feel the shape of this force function justifies another of the simplifying assumptions made in the model, namely that the effects of excitation-contraction dynamics (see Figure 2.37) are negligible for this application because they occur on a much shorter time scale than what is being simulated.

The original intent was to model the velocity-dependent damping characteristics of muscle through setting the *Rayleigh* parameters of the FEM equations as in Section 3.4.1. However, because the fiber forces are so much larger than the internal forces produced by the passive visco-elastic material, this approach caused the numerical integrator (LSODE from Section 5.4) to fail. The importance of including this damping is illustrated by the following example.

As discussed above, a change of shape in the simulated muscle is only predicted for a situation in which the contraction works against shape changes produced by external forces. Such a case is presented in Figure 6.7, in which the relaxed frog muscle bows downwards due to gravity. Upon contraction, the muscle pulls taut between its attachment points. If however, velocity dependent effects are not included, or the maximum velocity, $\tilde{v}_r^{CE}$, is set too high, the muscle will oscillate back and forth, much as a plucked string, in an amusing, albeit unrealistic manner. The arrows in the figure show the direction, but not the scale, of the world space forces acting at the finite element mesh node points.

6.1.4 Human Gastrocnemius Simulation

The FEM based muscle model is used in a similar way to simulate contraction of the medial gastrocnemius from a human subject, the polyhedral data for which is presented in Section 2.3. This muscle is approximated with a four element mesh made of twenty-node bricks as in the frog simulations. The medial gastrocnemius model is 24 cm long. The maximum isometric force used is 1113 N from Delp. Each of the 32 fibers generates 34.78 N or 3.48×10^6 dynes. The passive mechanical parameters for the FEM mesh are set as before,

$$\mathrm{E} = 2000\,\mathrm{dyne/cm^2} \qquad \nu = .49 \qquad \rho = 1.04\,\mathrm{g/cm^3} \qquad \mathrm{g} = -980\,\mathrm{cm/s^2}$$

To model fascia attachments to the rest of the leg, tendons are defined for nodes on

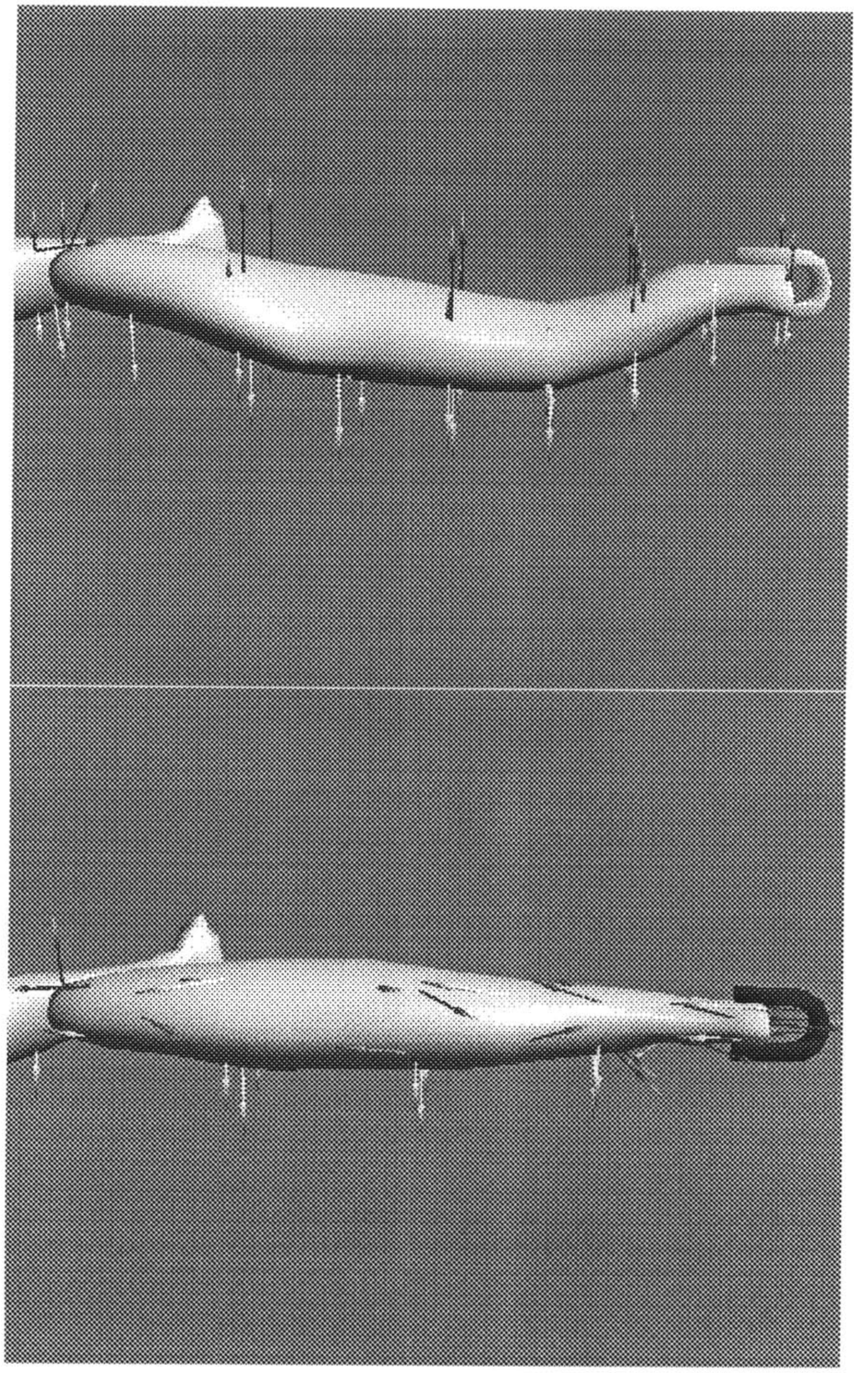


Figure 6.7: Relaxed muscle deforms due to gravity. Active muscle pulled taut.

the backside of the muscle. These are set so that the muscle can move freely, but not by a large amount. Furthermore, a reaction constraint (see Section 4.3) is defined so that the muscle will not penetrate the soleus. For the animation, the leg is rotated 45 degrees to horizontal and the relaxed muscle deforms due to gravity. Figure 6.8 shows that muscle contraction causes the gastrocnemius to pull taut as expected. Again, the arrows only show the direction in which the world space forces are acting.

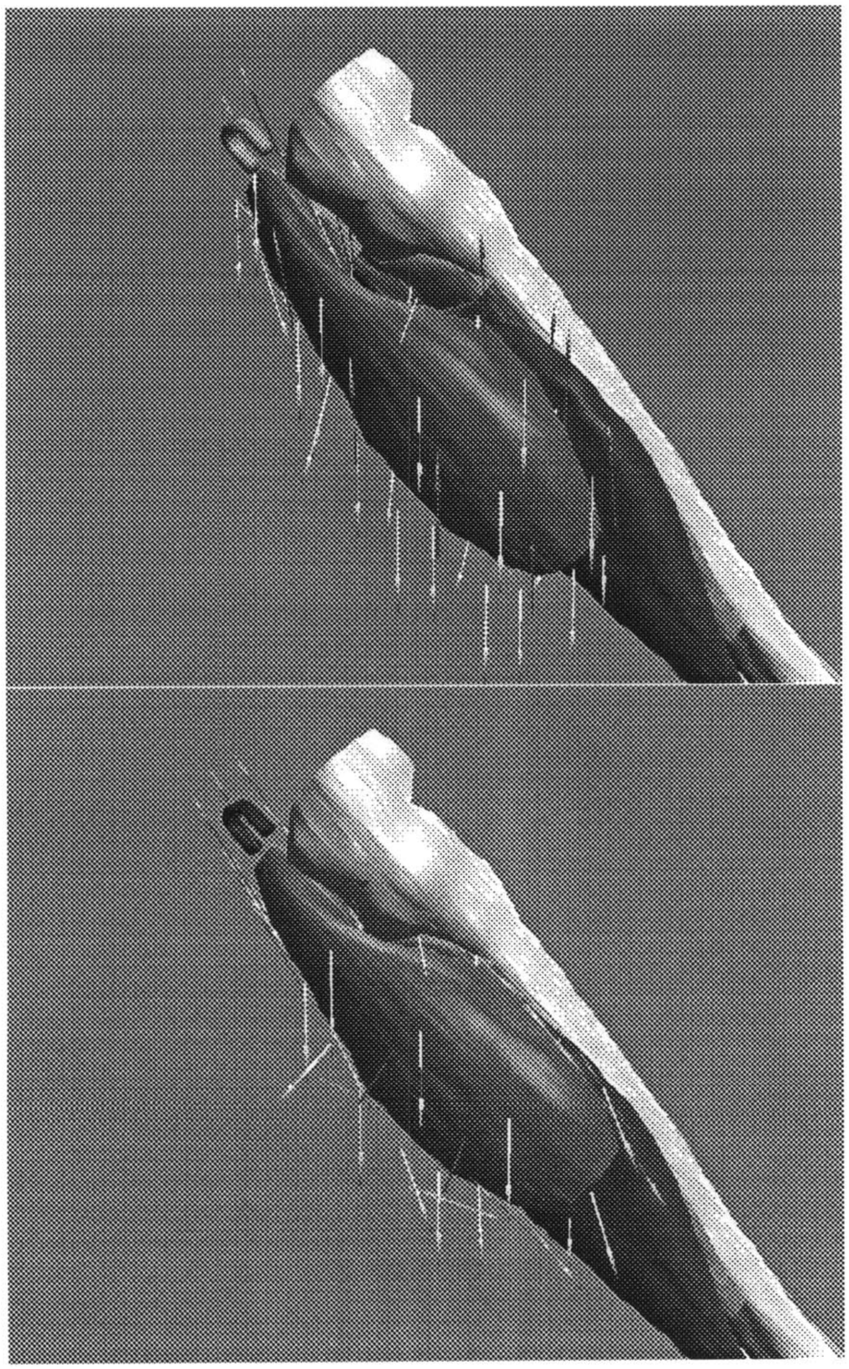


Figure 6.8: Human gastrocnemius deformed due to gravity, then pulled taut.

6.1.5 Future Work

Besides net force, the second quantity measured in both frog experiments was the volume of the finite element mesh throughout the stages of the simulation. For a twenty-node isoparametric cube, the volume is easy to compute by performing a numerical *gauss integration* of the determinant of the Jacobian matrix as discussed in Section 3.2. Plots are presented of the normalized volume, equal to the current volume divided by the rest volume in Figure 6.9 for the tension-length experiment, and in Figure 6.10 for the quick-release experiment. The rest volume of the gastrocnemius is 1.054 cm^3.

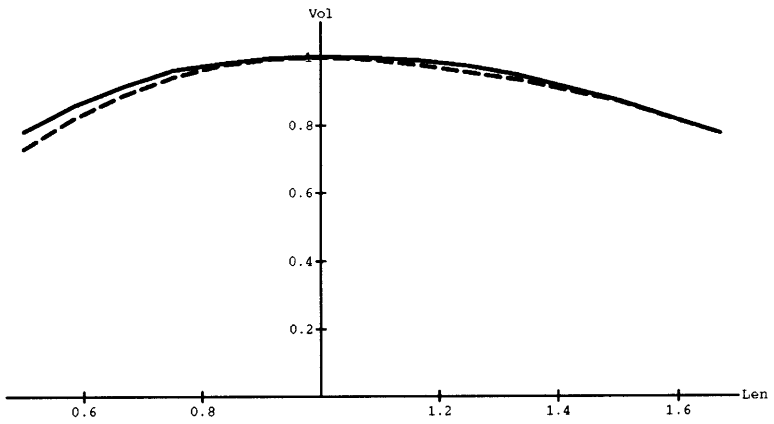


Figure 6.9: Simulation results for tension-length experiment. Normalized element volume vs. length. Solid line is volume for passive stretch, dashed line is for active muscle.

Even though Poisson's ratio was specified to approximate a volume preserving material, Figure 6.9 indicates this criterion for a muscle was not met; the likely reason being the *small-strain* approximation that was made in solving for the stiffness matrix K (see Section 3.1.1). To do a large-strain analysis requires both more complicated strain measures and more complicated constitutive relationships than those developed here [Bat82]. As a temporary workaround to this problem, a simple *volume constraint* was implemented that applies an instantaneous first-order correction to the mesh volume. Figure 6.11 shows the effect of this correction for the tension-length experiment.

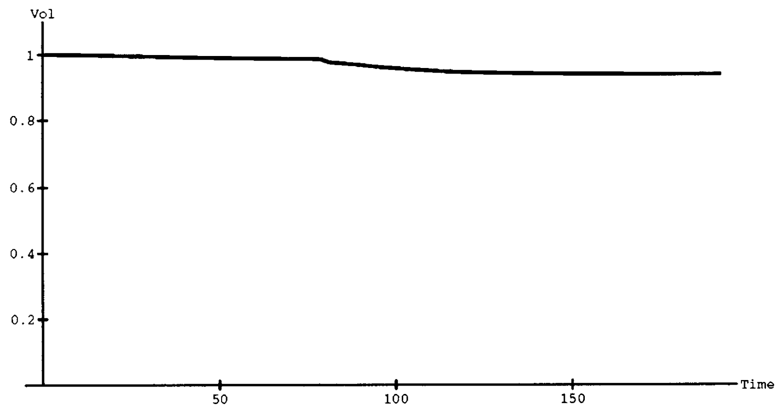


Figure 6.10: Simulation results for quick-release experiment. Normalized element volume vs. time.

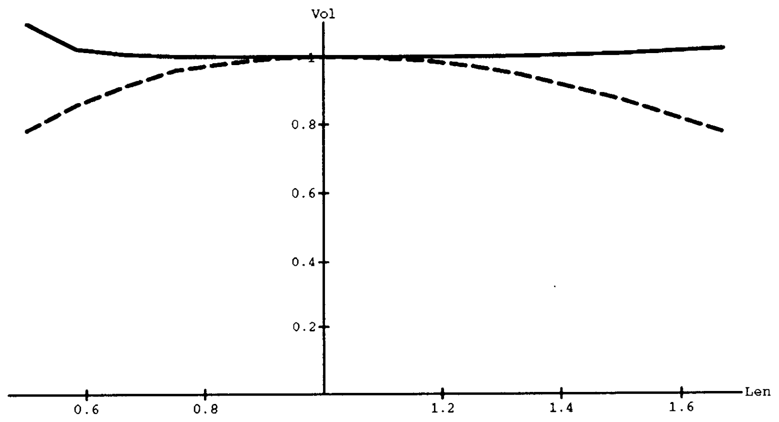


Figure 6.11: Simulation results for tension-length experiment with *volume constraint.* Normalized element volume vs. length. Solid line is volume for passive stretch with constraint. Dashed line is passive stretch without constraint.

We mentioned that another of the simplifying assumptions made in the muscle model is that all of the fibers are homogeneous in terms of the amount of force they can develop. The final experiment done using the frog gastrocnemius model examines the effect of simulated fiber force inhomogeneities on the total force produced. Here the force exerted by an individual fiber is scaled by the local volume measured by the containing finite element. Thus, the wider elements, which represent local volumes with greater physiologic cross-section (see Section 2.2) will produce more force than the fibers within the thinner elements at the tendon ends. The tension-length experiment was then repeated and the results plotted in Figure 6.12. The shape of the plot indicates inefficiencies of muscle function caused by uneven fiber lengths. The fibers within the thinner elements become overly stretched and are not able to produce as much force as they otherwise would. The resulting tension-length curve also does not exhibit the local maximum predicted by Zajac's model. This force inefficiency manifested itself in an interesting redistribution of the muscle mass in which a change of shape is produced even for an isometric case.

Though I am convinced that the results from this simulation are accurate in terms of the forces and changes in shape produced, at this time, no claim is made that real muscles exhibit this behavior. But, it does seem like a reasonable phenomena that might bear looking into. What this experiment does reveal is the kinds of investigations involving non-uniform fiber effects that can be performed with the FEM model developed in the thesis. Other experiments to try include staggering the activation pattern across the muscle, or varying the fiber length and orientation through the muscle.

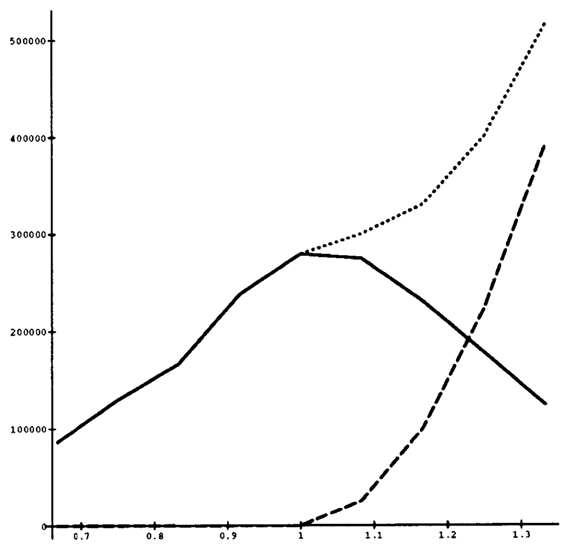


Figure 6.12: Simulation results for tension-length experiment with fiber inhomogeneities. Isometric muscle force vs. normalized length. Dotted line is total isometric force, dashed line is the force from passive muscle and the solid line is the developed force.

6.2 Computer Animation

6.2.1 Using the FEM for Computer Animation

Physical simulation has only in the past four or five years been utilized as a way of generating computer animation. As a form of engineering analysis, however, it has had a long and varied history. The displacement based finite element method that we have been exploring here at the Computer Graphics and Animation Group to simulate the physics of deformable objects has its beginnings in the early 1950's. Since then there has been a large amount of cross-disciplinary research and a very large number of publications on the subject.

While it is possible to create very realistic looking keyframe animations by using splines [Las87], this is often hard and tedious. To meet the goals of computer animation, physical simulation can provide a mechanism for automatically generating complex motions and changes in shape. The reason the FEM was chosen as the underlying formulation for the muscle model is that our goals include not only geometric visualization, but also building computational models that can be simulated and used to predict the behavior of real structures. In this way, the methods developed to allow fast graphical visualization and forms of interactive manipulation can carry over to interesting applications such as the surgery simulation systems presented by Delp [Del90] and Pieper [Pie92].

To do the thesis work, a software system *3d* (see Chapter 5) was written that can dynamically model elastically deformable objects based on the finite element method. The effects that can be employed include simulation of rigid body motion, simulation of deformation, collision with arbitrary polyhedrally represented objects, friction, and constraint satisfaction. An effort has been made to find solutions algorithmically and numerically suitable for graphics workstations.

Both for computer animation and simulation environments, we would like to be able to use familiar geometric representations—polygon and spline based patch descriptions—as rendering vehicles. On the other hand, the dynamic simulations that have been discussed are calculated based on a finite element mesh. A finite element implementation built on isoparametric elements can be used to target both of these functions. Like the free-form deformations described by Sederburg [SP86], a method has been developed in which the twenty-node isoparametric brick defines a free-form deformation that can be used to warp the points of polyhedral or patch objects very easily. Furthermore, the twenty-node brick is an extremely effective element on which to build a 3D finite element implementation and is widely used in engineering analysis.

To define assemblies of meshes based on polyhedral data, a user-assisted mesh generator has been written. The principle of virtual work is used to derive the stiffness and mass matrices that govern the dynamic behavior of the mesh. The modal transformation is performed to the matrix equations so that the differential equations defining the dynamic response can be more quickly and effectively solved. Mode truncation bandlimits the frequency response of the dynamic system resulting in fewer problems with stiff input forces. *Reaction constraints* as presented by Platt [PB88] have been extended for use in the FEM framework so that forces that match the goals of computer animators can be easily defined.

Two example computer animations are presented to illustrate using the finite element method to make animations of elastically deformable objects with controlled motions.

6.2.2 Jell-O® Dynamics

An animation is created of a visco-elastic cube of "Jello" discretized by eight twenty-node brick elements. The Jello has length 2 cm on a side. The geometric polyhedral model is a cube made with 100 polygons per face. All told, the geometry is defined by 602 points and 600 polygons. Though papers have been published regarding the optical properties of

Jello [Hec87], no one has yet examined its material properties (other than to say it *wiggles*). Jello is made mostly of water and so is assumed to be largely incompressible, and to have the density of water. For now, Young's modulus is chosen solely on the basis of the final appearance of the animation.

$$E = 1000 \text{ dyne/cm}^2 \qquad \nu = .49 \qquad \rho = 1 \text{ g/cm}^3 \qquad g = -980 \text{ cm/s}^2$$

The other objects in the scene are a funnel, a grid floor and a cutcube "rock". The initial conditions are such that the Jello begins off screen, above the funnel with zero velocity. The simulation begins, gravity causes the Jello to fall into the funnel, hit the cutcube rocks and drop off the grid floor. The funnel opening is set so that the Jello will have to squeeze and wiggle its way out. Collision constraints for the Jello are set for the grid and cutcube. The funnel rendered in the final animation is double-sided and so is inappropriate for the collision analysis algorithm implemented here. Thus a collision "stand-in" object is used to define the constraint for the funnel. Figure 6.13 shows the Jello falling through the funnel stand-in. Figure 6.14 is a finished software rendered scene, complete with transparency, lighting, and shadow effects.

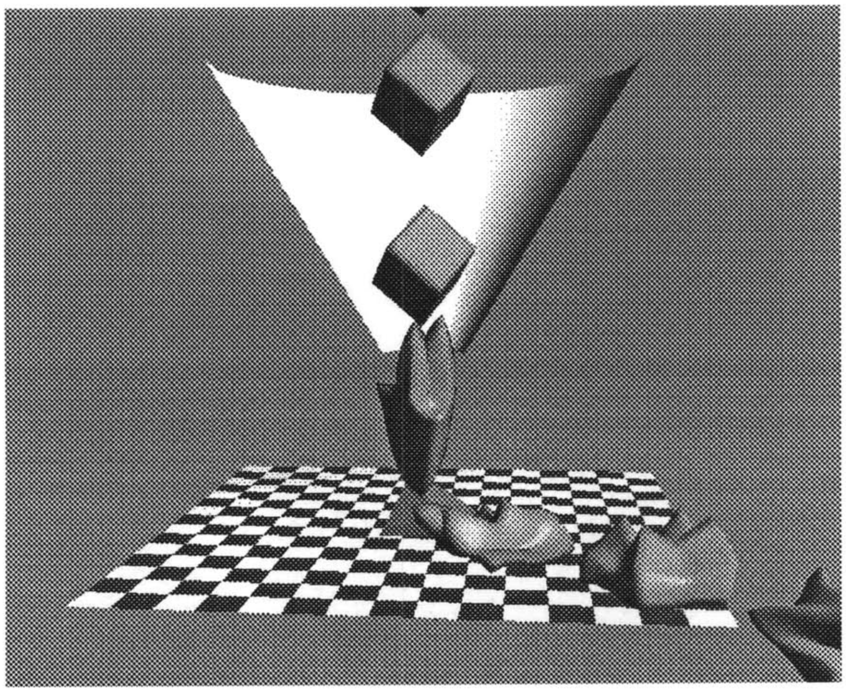


Figure 6.13: Jello motion

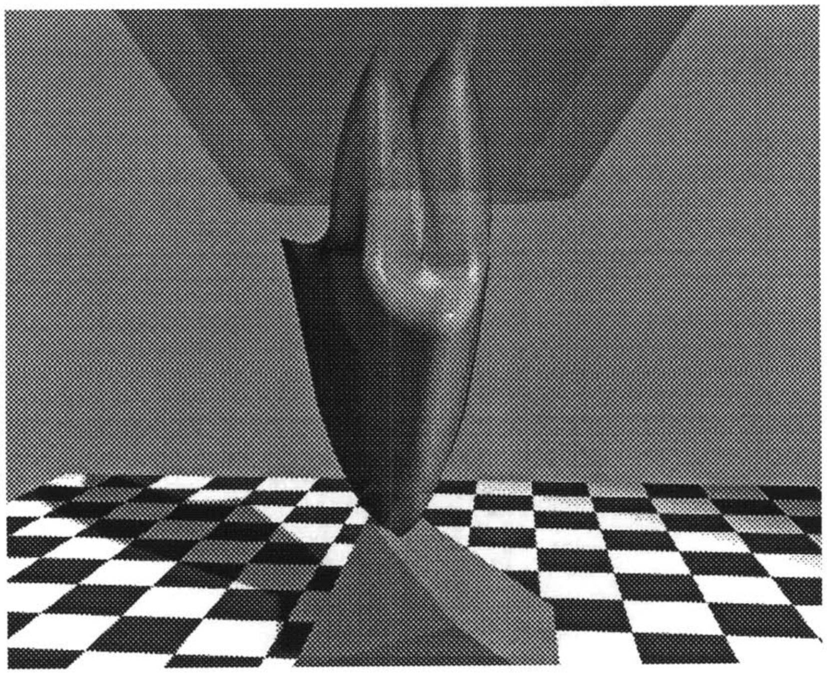


Figure 6.14: Scene from finished Jello animation

6.2.3 GED Puff

A computer animation is made of a "Puff" character bouncing across the floor. The polyhedral Puff object was created by Steve Strassman and made its first appearance in the award-winning short *Grinning Evil Death* by Mikey McKenna and Bob Sabiston. The Puff is discretized with three twenty-node brick elements. The Puff motion is created through the action of constraints. A small (30 line) program was written to kinematically control the up, down, and twisting motion of a rigid skeleton. The Puff is tied to the skeleton by reaction constraints at four of the mesh nodal points. A constraint is also used to make a collision with the grid floor. In this case, all simulation parameters are chosen for the sake of the final animation, and so that the computation time is minimized.

$$\mathrm{E} = 20\ \mathrm{dyne/cm^2} \qquad \nu = .3 \qquad \rho = .1\ \mathrm{g/cm^3} \qquad \mathrm{g} = -1\ \mathrm{cm/s^2}$$

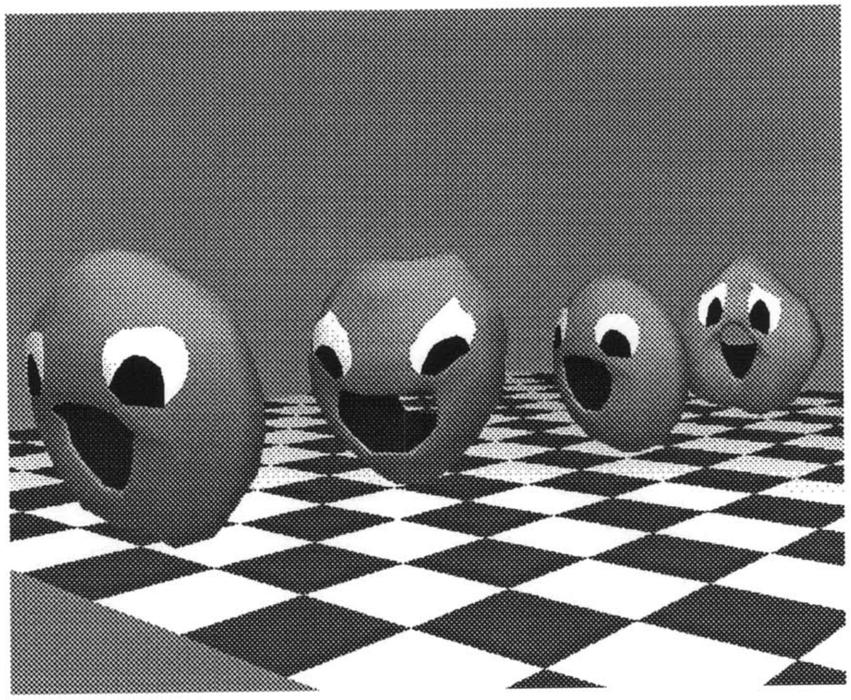


Figure 6.15: Blooby Puff bounce

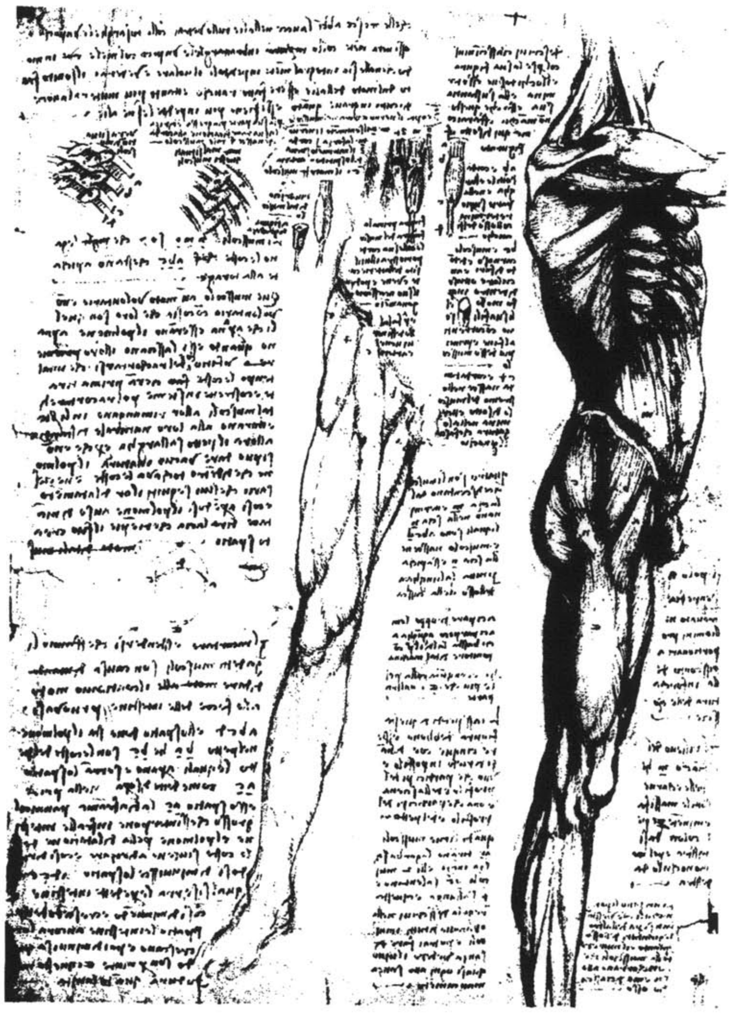
Chapter 7

Conclusions

For computer animation, the next step beyond kinematic simulations of skeleton movement must include realistic modeling and rendering of the muscle and skin. A finite element model of muscle has been developed that can be used both to simulate muscle forces and to visualize the dynamics of muscle contraction. Biomechanically, we have taken an existing model of muscle function from Zajac, added complexity by making it 3D, and shown that under certain circumstances it still behaves like a muscle. We have tried to validate the model by doing biomechanical experiments and plotting out key quantities.

It is much harder to validate the shape changes produced by the force-based FEM muscle model. Videos of live frog and human subjects have been made to serve as check-points for the simulations, but these can do so only qualitatively. The clearest way to verify the shape changes produced by the muscle model would be to make a whole series of MRI reconstructions, with the limb held in different states of isometric tension while measuring the muscle force wherever possible. A series of reconstructions made in this way would be appropriate data against which to compare simulation results. This would be a big project, and just barely doable with the current MRI technology.

However, by emphasizing the *physical model*, I think we have taken a step beyond the original computer graphics goal of making a virtual actor, up to the goal of making an *artificial person.* By studying the anatomy, the form will be revealed through the function.



from *Leonardo Da Vinci, Engineer and Architect*
The Montreal Museum of Fine Arts

Acknowledgements

I dedicate this thesis to my parents, who supported and encouraged me through the years, to my brother and sister, to my wife Betsy and our daughter Darcy for providing love and comfort, and to all the guys in the SnakePit, especially Stevie and Mikey, for making this such a fun and worthwhile experience.

Thanks, of course, go to many, many people, starting with my advisor, David Zeltzer for creating the Computer Graphics and Animation Group and starting me off on this research problem. The story of how muscles work is one of the most fascinating topics there is. The members of my thesis committee, Joe Rosen who gave up so much of his time, John Williams and Steve Benton all provided invaluable vision and encouragement. Special thanks go to Simon Gitzer for sharing with me his experience and understanding of muscle function.

And thanks to everyone else, without whom I never would have finished, David Sturman, Steve Drucker, Tinsley Galyean, Margaret Minsky, Peter Schroeder, Mike Halle, Linda Peterson, walter, lacsap, hank, march, eman, ferdi, kenji, mat, bconn, brian, alex, irfan, sclaroff, kk, shigeru, janette, judith, gitta, jh, klug, barons, ken, leo, lpeeps, yoshi, russ, ...

Bibliography

[AD85] M. L. Audu and D. T. Davy. The influence of muscle model complexity in musculoskeletal motion modeling. *Journal of Biomechanical Engineering*, 107:147–157, 1985.

[AG85] William W. Armstrong and Mark Green. The dynamics of articulated rigid bodies for purposes of animation. *Proceedings, Graphics Interface '85*, pages 407–415, 1985.

[Bat82] Klaus-Jürgen Bathe. *Finite Element Procedures in Engineering Analysis.* Prentice-Hall, 1982.

[BC89] Armin Bruderlin and Thomas W. Calvert. Goal-directed, dynamic animation of human walking. *ACM Computer Graphics*, 23(3):233–242, 1989.

[BOT79] N.I. Badler, J. O'Rourke, and H. Toltzis. A spherical representation of a human body for visualizing movement. *Proceedings IEEE*, 67(10):1397–1402, October 1979.

[CB89] Soo-Won Chae and Klaus-Jürgen Bathe. On automatic mesh construction and mesh refinement in finite element analysis. *Computers & Structures*, 32(3/4):911–936, 1989.

[CGT86] G. Cecchi, P.J. Griffiths, and S. Taylor. Stiffness and force in activated frog skeletal muscle fibers. *Biophysical Journal*, 49:437–451, 1986.

[Cha88] Soo-Won Chae. *On the Automatic Generation of Near-Optimal Meshes for Three-Dimensional Linear-Elastic Finite Element Analysis.* PhD thesis, Massachusetts Institute of Technology, 1988.

[CHP89] John E. Chadwick, David R. Haumann, and Richard E. Parent. Layered construction for deformable animated characters. *ACM Computer Graphics*, 23(3):243–252, 1989.

[CW74] F. D. Carlson and D. R. Wilkie. *Muscle Physiology.* Prentice-Hall, Inc., 1974.

[Del90] Scott L. Delp. *Surgery Simulation: A Computer Graphics System to Analyze and Design Musculoskeletal Reconstructions of the Lower Limb.* PhD thesis, Stanford University, 1990.

[Fea87] Roy Featherstone. *Robot Dynamics Algorithms.* Kluwer Academic Publishers, 1987.

[Fet82] William A. Fetter. A progression of human figures simulated by computer graphics. *IEEE Computer Graphics and Applications*, pages 9–13, November 1982.

[Fou90] Open Software Foundation. *OSF/Motif™ Programmer's Reference.* Prentice-Hall, Inc., 1990.

[GA88] Andrew P. Grieve and Cecil G. Armstrong. Compressive properties of soft tissues. In G. de Groot, A. P. Hollander, P. A. Huijing, and G. J. van Ingen Schenau, editors, *Biomechanics XI-A*, International Series on Biomechanics, Amsterdam, 1988. Free Unversity Press.

[GH24] H.S. Gasser and A.V. Hill. The dynamics of mucular contraction. *Royal Society of London Proceedings*, 96:398–437, 1924.

[GMTT89] Jean-Paul Gourret, Nadia Magnenat-Thalman, and Daniel Thalman. Simulation of object and human skin deformations in a grasping task. *ACM Computer Graphics*, 24(3):21–30, 1989.

[Har49] J.P. Den Hartog. *Strength of Materials.* Dover Publications, 1949.

[Hat76] H. Hatze. The complete optimization of the human motion. *Mathematical Biosciences*, 28:99–135, 1976.

[Hec87] Paul S. Heckbert. Ray tracing jell-o® brand gelatin. *ACM Computer Graphics*, 21(4):73–74, 1987.

[HH54] H. Huxley and J. Hanson. Changes in the cross-striations of muscle during contraction and stretch and their structural interpretation. *Nature*, 173:973–976, 1954.

[HN54] A.F Huxley and R. Niedergerke. Structural changes in muscle during contraction. *Nature*, 173:971–973, 1954.

[JBDT88] D.W.G Jung, T. Blangé, H. DeGraaf, and B.W. Treijtel. Elastic properties of relaxed, activated and rigor muscle fibers measured with microsecond resolution. *Biophysical Journal*, 54:897–908, 1988.

[Joh91] Michael Boyle Johnson. Build-a-Dude: Action selection networks for computational autonomous agents. Master's thesis, Massachusetts Institute of Technology, 1991.

[Kel71] David L. Kelley. *Kinesiology: Fundamentals of Motion Description.* Prentice Hall, Inc., 1971.

[Kom86] Koji Komatsu. Human skin model capable of natural shape variation. Laboratories Note 329, NHK, Tokyo, March 1986.

[Las87] John Lasseter. Principles of traditional animation applied to 3d computer animation. *ACM Computer Graphics*, 21(4):35–44, 1987.

[LC87] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[Lot84] Per Lotstedt. Numerical simulation of time-dependent contact and friction problems in rigid body mechanics. *SIAM Journal of Scientific Statistical Computing*, 5(2), 1984.

[LW27] A. Levin and J. Wyman. The viscous elastic properties of muscle. *Royal Society of London Proceedings*, 101:218–243, 1927.

[Max83] Delle Rae Maxwell. Graphical marionette: a modern-day Pinocchio. Master's thesis, Massachusetts Institute of Technology, 1983.

[McK90] Michael McKenna. A dynamic model of locomotion for computer animation. Master's thesis, Massachusetts Institute of Technology, 1990.

[McM84] Thomas A. McMahon. *Muscles, Reflexes, and Locomotion.* Princeton University Press, 1984.

[MhLH90] S. McKenna, Y. harvill, A. Louie, and D. Huffman. *Swivel 3D Professional User's Guide.* Paracomp, Inc., 1987-1990.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. *ACM Computer Graphics*, 22(4):289–298, 1988.

[Ous90] J.K. Ousterhout. Tcl: An embeddable command language. *1990 Winter USENIX Conference Proceedings*, 1990.

[Pau81] Richard P. Paul. *Robot Manipulators: Mathematics, Programming and Control.* The MIT Press, Cambridge, Massachusetts, 1981.

[PB88] John C. Platt and Alan H. Barr. Constraint methods for physical models. *ACM Computer Graphics*, 22(4):279–288, 1988.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[Pie90] Steven Donald Pieper. More than skin deep: Physical modeling of facial tissue. Master's thesis, Massachusetts Institute of Technology, 1990.

[Pie92] Steven Donald Pieper. *CAPS: Computer-Aided Plastic Surgery.* PhD thesis, Massachusetts Institute of Technology, 1992.

[Pla89] John Platt. *Constraint Methods for Neural Networks and Computer Graphics.* PhD thesis, California Institute of Technology, April 1989. Department of Computer Science, Caltech-CS-TR-89-07.

[PMSM81] A.G. Patriarco, R.W. Mann, S.R. Simaon, and J.M. Mansour. An evaluation of the approaches of optimization models in the prediction of muscle forces during human gait. *Journal of Biomechanics*, 14(8):513–525, 1981.

[PW89] Alex Pentland and John Williams. Good vibrations: Modal dynamics for graphics and animation. *ACM Computer Graphics*, 23(3):215–222, 1989.

[Rib82] Eric Alan Ribble. Synthesis of human skeletal motion and the design of a special-purpose processor for real-time animation of human and animal figure motion. Master's thesis, Ohio State University, 1982.

[Roc83] K. C. Rockey. *The Finite Element Method: A Basic Introduction.* Wiley, 2 edition, 1983.

[Smi84] Alvy Ray Smith. The viewing transformation. Technical report, Computer Graphics Project, Computer Division, Lucasfilm Ltd., Marin, California, May 1984.

[SP86] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *ACM Computer Graphics*, 20(4):151–160, 1986.

[Str91] Steven Henry Strassmann. *Desktop Theater: Automatic Generation of Expressive Animation.* PhD thesis, Massachusetts Institute of Technology, 1991.

[SZ90] Peter Schröder and David Zeltzer. The virtual erector set. *ACM Computer Graphics*, 24(2):23–32, 1990.

[TF88] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. *ACM Computer Graphics*, 22(4):269–278, 1988.

[VSL75] Arthur J. Vander, James H. Sherman, and Dorothy S. Luciano. *Human Physiology - The Mechanisms of Body Function.* McGraw Hill, 1975.

[Wil56] D. R. Wilkie. Measurement of the series elastic component at various times during a single muscle twitch. *Journal Physiology*, 134:527–530, 1956.

[Wil87] Jane Wilhelms. Using dynamic analysis for realistic animation of articulated bodies. *IEEE Computer Graphics and Applications*, pages 12–27, June 1987.

[Wil90] Lance Williams. 3d paint. *ACM Computer Graphics*, 24(2):225–233, 1990.

[WMJ89] J.E. Wood, S.G. Meek, and S.C. Jacobsen. Quantization of human shoulder anatomy for prosthetic arm control, part i. *Journal of Biomechanics*, page 273, 1989.

[Woo76] John Everett Wood. *Theoretical Formalism for the Kinesiological Trajectories of a Computer Simulated Neuro-Musculo-Skeletal System.* PhD thesis, Massachusetts Institue of Technology, 1976.

[You90] Douglas A. Young. *The X Window System® Programming and Applications with Xt: OSF/Motif® Edition.* Prentice-Hall, Inc., 1987-1990.

[Zaj89] F. E. Zajac. Muscle and tendon: Properties, models, scaling, and application to biomechanics and motor control. *Critical Reviews in Biomedical Engineering*, 17:359–411, 1989.

[Zel84] David Zeltzer. *Representation and Control of Three Dimensional Computer Animated Figures.* PhD thesis, Ohio State University, 1984.

[ZPS89] D. Zeltzer, S. Pieper, and D. Sturman. An integrated graphical simulation platform. *Proc. Graphics Interface '89*, 134:266–274, June 1989.

[ZTS86] F. E. Zajac, E.L. Topp, and P.J. Stevenson. A dimensionless musculotendon model. *Proceedings IEEE Engineering in Medicine and Biology*, 1986.

Appendix A

Isoparametric Interpolation

The following is a C implementation of the blending functions for a 20 node isoparametric cube element.

```
/*
 *  c20-map.c                    dead@media-lab 02/20/90
 *
 *  c20_nodes array added by stevie 9/13/90
 */
#include <stdio.h>
#include <math.h>

#ifndef lint
static char copyright[] =
"Copyright (c) 1989, 1990 by David T. Chen.  All rights reserved.";
#endif

typedef int VectorI[3];
typedef float Vector[3];
typedef float Matrix[4][4];

typedef double (*PFD)();

static VectorI UNIT_NODES[20] =
{  {1, 1, 1},    {-1, 1, 1},    {-1, -1, 1},    {1, -1, 1},
   {1, 1, -1},   {-1, 1, -1},   {-1, -1, -1},   {1, -1, -1},
   {0, 1, 1},    {-1, 0, 1},    {0, -1, 1},     {1, 0, 1},
   {0, 1, -1},   {-1, 0, -1},   {0, -1, -1},    {1, 0, -1},
    {1, 1, 0},    {-1, 1, 0},    {-1, -1, 0},    {1, -1, 0}
};

/*
*  c20_nodes: Pointer to an array of node pointers.
*  If the node pointer is null, correponding interpolation
*  functions will not be included.        -Stevie 9/13/90
*/
static char **c20_nodes = NULL;
#define  c20_node_exists(i) ((c20_nodes) ? (c20_nodes[(i)]) : ((char *)1))

c20_node_set_array(nodes)
char **nodes;
{  c20_nodes = nodes;
}

/*
===================================================
Interpolation functions
   G(), dG()
===================================================
*/
static double G( beta, beta_i )
double beta;
int beta_i;
{
double ans;

   if (beta_i == 0) ans = 1. - (beta * beta);
   else if (beta_i == 1) ans = .5 * (1 + beta);
   else if (beta_i == -1) ans = .5 * (1 - beta);
   else
   {  fprintf( stderr, "G: bad beta_i <%d>\n", beta_i );
      ans = 0.;
   }
   return( ans );
}

static double dG( beta, beta_i )
double beta;
int beta_i;
{
double ans;

   if (beta_i == 0) ans = -2. * beta;
   else if (beta_i == 1) ans = .5;
   else if (beta_i == -1) ans = -.5;
   else
   {  fprintf( stderr, "dG: bad beta_i <%d>\n", beta_i );
      ans = 0.;
   }
```

```
    return( ans );
}

/*
===================================================
Function g and the partials with respect to r, s, t
    g(), dgdr(), dgds(), dgdt()
===================================================
*/
static double g( r, s, t, i )
double r, s, t;
int i;
{
double ans;

    if (i > 19 || i < 0) return( 0. );
    if ( !c20_node_exists(i) ) return( 0. );

    ans  = G( r, UNIT_NODES[i][0] );
    ans *= G( s, UNIT_NODES[i][1] );
    ans *= G( t, UNIT_NODES[i][2] );
    return( ans );
}

static double dgdr( r, s, t, i )
double r, s, t;
int i;
{
double ans;

    if (i > 19 || i < 0) return( 0. );
    if ( !c20_node_exists(i) ) return( 0. );

    ans  = dG( r, UNIT_NODES[i][0] );
    ans *=  G( s, UNIT_NODES[i][1] );
    ans *=  G( t, UNIT_NODES[i][2] );
    return( ans );
}

static double dgds( r, s, t, i )
double r, s, t;
int i;
{
double ans;

    if (i > 19 || i < 0) return( 0. );
    if ( !c20_node_exists(i) ) return( 0. );

    ans  =  G( r, UNIT_NODES[i][0] );
    ans *= dG( s, UNIT_NODES[i][1] );
    ans *=  G( t, UNIT_NODES[i][2] );
    return( ans );
}

static double dgdt( r, s, t, i )
double r, s, t;
int i;
{
double ans;

    if (i > 19 || i < 0) return( 0. );
    if ( !c20_node_exists(i) ) return( 0. );

    ans  =  G( r, UNIT_NODES[i][0] );
    ans *=  G( s, UNIT_NODES[i][1] );
    ans *= dG( t, UNIT_NODES[i][2] );
    return( ans );
}

/*
====================================================
Function h and the partials with respect to r, s, t
    c20_h(), c20_dhdr(), c20_dhds(), c20_dhdt()
====================================================
*/
static double c20_h_BLEND( r, s, t, i, name, g )
double r, s, t;
int i;
char *name;
PFD g;
{
double ans;

    if (i > 19 || i < 0)
    { fprintf( stderr, "%s: bad i <%d>\n", name, i );
       return( 0. );
    }
    ans = (*g)(r,s,t,i);

    if (i == 0)
    {  ans += -.5 * ((*g)(r,s,t,8) + (*g)(r,s,t,11) + (*g)(r,s,t,16));
    }
    else if (i == 1)
    {  ans += -.5 * ((*g)(r,s,t,8) + (*g)(r,s,t,9) + (*g)(r,s,t,17));
    }
    else if (i == 2)
    {  ans += -.5 * ((*g)(r,s,t,9) + (*g)(r,s,t,10) + (*g)(r,s,t,18));
    }
    else if (i == 3)
    {  ans += -.5 * ((*g)(r,s,t,10) + (*g)(r,s,t,11) + (*g)(r,s,t,19));
    }
    else if (i == 4)
```

```
        {  ans += -.5 * ((*g)(r,s,t,12) + (*g)(r,s,t,15) + (*g)(r,s,t,16));
        }
        else if (i == 5)
        {  ans += -.5 * ((*g)(r,s,t,12) + (*g)(r,s,t,13) + (*g)(r,s,t,17));
        }
        else if (i == 6)
        {  ans += -.5 * ((*g)(r,s,t,13) + (*g)(r,s,t,14) + (*g)(r,s,t,18));
        }
        else if (i == 7)
        {  ans += -.5 * ((*g)(r,s,t,14) + (*g)(r,s,t,15) + (*g)(r,s,t,19));
        }
        return( ans );
    }

    double c20_h( r, s, t, i )
    double r, s, t;
    int i;
    {  return( c20_h_BLEND(r, s, t, i, "c20_h", g) );
    }

    double c20_dhdr( r, s, t, i )
    double r, s, t;
    int i;
    {  return( c20_h_BLEND(r, s, t, i, "c20_dhdr", dgdr) );
    }

    double c20_dhds( r, s, t, i )
    double r, s, t;
    int i;
    {  return( c20_h_BLEND(r, s, t, i, "c20_dhds", dgds) );
    }

    double c20_dhdt( r, s, t, i )
    double r, s, t;
    int i;
    {  return( c20_h_BLEND(r, s, t, i, "c20_dhdt", dgdt) );
    }

    /*
    ==================================================
    Coordinate transformation from natural to local space.
    Input twenty nodal points and an array of natural coordinate points.
    Output points in local space. 'num' is array length.
    ==================================================
    */
    c20_Natural2Local( node_points, natural, local, num )
    Vector *node_points, *natural, *local;
    int num;
    {
    int i, j;
    double r, s, t, x, y, z, h;
    for (i=0; i<num; i++)
    {
       r = natural[i][0]; s = natural[i][1]; t = natural[i][2];
       x = y = z = 0.;

       for (j=0; j<20; j++)
       {  h = c20_h( r, s, t, j );
          x += h * node_points[j][0];
          y += h * node_points[j][1];
          z += h * node_points[j][2];
       }
       local[i][0] = x; local[i][1] = y; local[i][2] = z;
    }
}
```

Appendix B

Fast Interpolation

The following is a C implementation of the blending functions for a 20 node isoparametric cube element.

```
/*
*  c20-fast.c                    dead@media-lab 02/20/90
*
*  Interpolate through a 20 node isoparametric cube.
*  Math from stevie and Mathematica
*/

typedef float Vector[3];

/*
==================================================
c20_X()
==================================================
*/
double c20_X( C, r, s, t )
double *C, r, s, t;
{
double r2, s2, t2, ans;

   r2 = r*r; s2 = s*s; t2 = t*t;

   ans = C[0]                + C[1]   * r * s
      + C[2]   * s * t2     + C[3]   * r * s * t2
      + C[4]   * r * t2     + C[5]   * r * s * t
```

```
        + C[6]   * t2          + C[7]   * r2 * t
        + C[8]   * r2 * s      + C[9]   * r2 * s * t
        + C[10]  * r2          + C[11]  * s * t
        + C[12]  * s           + C[13]  * r * t
        + C[14]  * s2 * t      + C[15]  * r
        + C[16]  * s2          + C[17]  * r * s2 * t
        + C[18]  * r * s2      + C[19]  * t;

    return( ans );
}

double c20_dXdr( C, r, s, t )
double *C, r, s, t;
{
double r2, s2, t2, ans;

    r2 = 2 * r; r = 1.;
    s2 = s * s; t2 = t * t;

    ans = C[1]  * r * s         + C[3]   * r * s * t2
        + C[4]  * r * t2        + C[5]   * r * s * t
        + C[7]  * r2 * t        + C[8]   * r2 * s
        + C[9]  * r2 * s * t    + C[10]  * r2
        + C[13] * r * t         + C[15]  * r
        + C[17] * r * s2 * t    + C[18]  * r * s2;

    return( ans );
}

double c20_dXds( C, r, s, t )
double *C, r, s, t;
{
double r2, s2, t2, ans;

    s2 = 2 * s; s = 1.;
    r2 = r * r; t2 = t * t;

    ans = C[1]  * r * s         + C[2]   * s * t2
        + C[3]  * r * s * t2    + C[5]   * r * s * t
        + C[8]  * r2 * s        + C[9]   * r2 * s * t
        + C[11] * s * t         + C[12]  * s
        + C[14] * s2 * t        + C[16]  * s2
        + C[17] * r * s2 * t    + C[18]  * r * s2;

    return( ans );
}

double c20_dXdt( C, r, s, t )
double *C, r, s, t;
{
double r2, s2, t2, ans;
```

```
    t2 = 2 * t; t = 1.;
    r2 = r * r; s2 = s * s;

    ans = C[2]  * s * t2        + C[3]  * r * s * t2
        + C[4]  * r * t2        + C[5]  * r * s * t
        + C[6]  * t2            + C[7]  * r2 * t
        + C[9]  * r2 * s * t    + C[11] * s * t
        + C[13] * r * t         + C[14] * s2 * t
        + C[17] * r * s2 * t    + C[19] * t;

    return( ans );
}

/*
===============================================
c20_make_polynomial()
===============================================
*/
c20_make_polynomial( nodes, index, C )
Vector *nodes;
int index;
double C[];
{
    C[0] =
        - 0.25*nodes[0][index] - 0.25*nodes[1][index]
        - 0.25*nodes[2][index] - 0.25*nodes[3][index]
        - 0.25*nodes[4][index] - 0.25*nodes[5][index]
        - 0.25*nodes[6][index] - 0.25*nodes[7][index]
        + 0.25*nodes[8][index] + 0.25*nodes[9][index]
        + 0.25*nodes[10][index] + 0.25*nodes[11][index]
        + 0.25*nodes[12][index] + 0.25*nodes[13][index]
        + 0.25*nodes[14][index] + 0.25*nodes[15][index]
        + 0.25*nodes[16][index] + 0.25*nodes[17][index]
        + 0.25*nodes[18][index] + 0.25*nodes[19][index];

    C[1] =
        0.25*nodes[16][index] - 0.25*nodes[17][index]
        + 0.25*nodes[18][index] - 0.25*nodes[19][index];

    C[2] =
        0.125*nodes[0][index] + 0.125*nodes[1][index]
        - 0.125*nodes[2][index] - 0.125*nodes[3][index]
        + 0.125*nodes[4][index] + 0.125*nodes[5][index]
        - 0.125*nodes[6][index] - 0.125*nodes[7][index]
        - 0.25*nodes[16][index] - 0.25*nodes[17][index]
        + 0.25*nodes[18][index] + 0.25*nodes[19][index];

    C[3] =
        0.125*nodes[0][index] - 0.125*nodes[1][index]
        + 0.125*nodes[2][index] - 0.125*nodes[3][index]
    + 0.125*nodes[4][index] − 0.125*nodes[5][index]
    + 0.125*nodes[6][index] − 0.125*nodes[7][index]
    − 0.25*nodes[16][index] + 0.25*nodes[17][index]
    − 0.25*nodes[18][index] + 0.25*nodes[19][index];

C[4] =
    0.125*nodes[0][index] − 0.125*nodes[1][index]
    − 0.25*nodes[16][index] + 0.25*nodes[17][index]
    + 0.25*nodes[18][index] − 0.25*nodes[19][index]
    − 0.125*nodes[2][index] + 0.125*nodes[3][index]
    + 0.125*nodes[4][index] − 0.125*nodes[5][index]
    − 0.125*nodes[6][index] + 0.125*nodes[7][index];

C[5] =
    0.125*nodes[0][index] − 0.125*nodes[1][index]
    + 0.125*nodes[2][index] − 0.125*nodes[3][index]
    − 0.125*nodes[4][index] + 0.125*nodes[5][index]
    − 0.125*nodes[6][index] + 0.125*nodes[7][index];

C[6] =
    0.125*nodes[0][index] + 0.125*nodes[1][index]
    − 0.25*nodes[16][index] − 0.25*nodes[17][index]
    − 0.25*nodes[18][index] − 0.25*nodes[19][index]
    + 0.125*nodes[2][index] + 0.125*nodes[3][index]
    + 0.125*nodes[4][index] + 0.125*nodes[5][index]
    + 0.125*nodes[6][index] + 0.125*nodes[7][index];

C[7] =
    0.125*nodes[0][index] + 0.125*nodes[1][index]
    − 0.25*nodes[10][index] + 0.25*nodes[12][index]
    + 0.25*nodes[14][index] + 0.125*nodes[2][index]
    + 0.125*nodes[3][index] − 0.125*nodes[4][index]
    − 0.125*nodes[5][index] − 0.125*nodes[6][index]
    − 0.125*nodes[7][index] − 0.25*nodes[8][index];

C[8] =
    0.125*nodes[0][index] + 0.125*nodes[1][index]
    + 0.25*nodes[10][index] − 0.25*nodes[12][index]
    + 0.25*nodes[14][index] − 0.125*nodes[2][index]
    − 0.125*nodes[3][index] + 0.125*nodes[4][index]
    + 0.125*nodes[5][index] − 0.125*nodes[6][index]
    − 0.125*nodes[7][index] − 0.25*nodes[8][index];

C[9] =
    0.125*nodes[0][index] + 0.125*nodes[1][index]
    + 0.25*nodes[10][index] + 0.25*nodes[12][index]
    − 0.25*nodes[14][index] − 0.125*nodes[2][index]
    − 0.125*nodes[3][index] − 0.125*nodes[4][index]
    − 0.125*nodes[5][index] + 0.125*nodes[6][index]
    + 0.125*nodes[7][index] − 0.25*nodes[8][index];
```

```
C[10] =
   0.125*nodes[0][index] + 0.125*nodes[1][index]
   - 0.25*nodes[10][index] - 0.25*nodes[12][index]
   - 0.25*nodes[14][index] + 0.125*nodes[2][index]
   + 0.125*nodes[3][index] + 0.125*nodes[4][index]
   + 0.125*nodes[5][index] + 0.125*nodes[6][index]
   + 0.125*nodes[7][index] - 0.25*nodes[8][index];

C[11] =
   -0.25*nodes[10][index] - 0.25*nodes[12][index]
   + 0.25*nodes[14][index] + 0.25*nodes[8][index];

C[12] =
   -0.125*nodes[0][index] - 0.125*nodes[1][index]
   - 0.25*nodes[10][index] + 0.25*nodes[12][index]
   - 0.25*nodes[14][index] + 0.25*nodes[16][index]
   + 0.25*nodes[17][index] - 0.25*nodes[18][index]
   - 0.25*nodes[19][index] + 0.125*nodes[2][index]
   + 0.125*nodes[3][index] - 0.125*nodes[4][index]
   - 0.125*nodes[5][index] + 0.125*nodes[6][index]
   + 0.125*nodes[7][index] + 0.25*nodes[8][index];

C[13] =
   0.25*nodes[11][index] + 0.25*nodes[13][index]
   - 0.25*nodes[15][index] - 0.25*nodes[9][index];

C[14] =
   0.125*nodes[0][index] + 0.125*nodes[1][index]
   - 0.25*nodes[11][index] + 0.25*nodes[13][index]
   + 0.25*nodes[15][index] + 0.125*nodes[2][index]
   + 0.125*nodes[3][index] - 0.125*nodes[4][index]
   - 0.125*nodes[5][index] - 0.125*nodes[6][index]
   - 0.125*nodes[7][index] - 0.25*nodes[9][index];

C[15] =
   -0.125*nodes[0][index] + 0.125*nodes[1][index]
   + 0.25*nodes[11][index] - 0.25*nodes[13][index]
   + 0.25*nodes[15][index] + 0.25*nodes[16][index]
   - 0.25*nodes[17][index] - 0.25*nodes[18][index]
   + 0.25*nodes[19][index] + 0.125*nodes[2][index]
   - 0.125*nodes[3][index] - 0.125*nodes[4][index]
   + 0.125*nodes[5][index] + 0.125*nodes[6][index]
   - 0.125*nodes[7][index] - 0.25*nodes[9][index];

C[16] =
   0.125*nodes[0][index] + 0.125*nodes[1][index]
   - 0.25*nodes[11][index] - 0.25*nodes[13][index]
   - 0.25*nodes[15][index] + 0.125*nodes[2][index]
   + 0.125*nodes[3][index] + 0.125*nodes[4][index]
   + 0.125*nodes[5][index] + 0.125*nodes[6][index]
   + 0.125*nodes[7][index] - 0.25*nodes[9][index];
```

```
    C[17] =
      0.125*nodes[0][index] - 0.125*nodes[1][index]
      - 0.25*nodes[11][index] - 0.25*nodes[13][index]
      + 0.25*nodes[15][index] - 0.125*nodes[2][index]
      + 0.125*nodes[3][index] - 0.125*nodes[4][index]
      + 0.125*nodes[5][index] + 0.125*nodes[6][index]
      - 0.125*nodes[7][index] + 0.25*nodes[9][index];

    C[18] =
      0.125*nodes[0][index] - 0.125*nodes[1][index]
      - 0.25*nodes[11][index] + 0.25*nodes[13][index]
      - 0.25*nodes[15][index] - 0.125*nodes[2][index]
      + 0.125*nodes[3][index] + 0.125*nodes[4][index]
      - 0.125*nodes[5][index] - 0.125*nodes[6][index]
      + 0.125*nodes[7][index] + 0.25*nodes[9][index];

    C[19] =
      -0.125*nodes[0][index] - 0.125*nodes[1][index]
      + 0.25*nodes[10][index] + 0.25*nodes[11][index]
      - 0.25*nodes[12][index] - 0.25*nodes[13][index]
      - 0.25*nodes[14][index] - 0.25*nodes[15][index]
      - 0.125*nodes[2][index] - 0.125*nodes[3][index]
      + 0.125*nodes[4][index] + 0.125*nodes[5][index]
      + 0.125*nodes[6][index] + 0.125*nodes[7][index]
      + 0.25*nodes[8][index] + 0.25*nodes[9][index];
}
```