Haptic Rendering and Psychophysical Evaluation of a Virtual Three-Dimensional Helical Spring

Vinithra Varadharajan

Master's Thesis Report

CMU-RI-TR-07-21

May 24^{th}, 2007

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This Masters thesis presents the development of a new deformable object for haptic interaction in the form of a 3D helical spring. This haptic and visual simulation is based on an analytical model of a quasistatic spring. The model provides a real-time computationally efficient method for rendering a deformable spring using a magnetic levitation haptic device. The solution includes equations for reaction forces and resisting moments experienced during compression, elongation, shear and tilting of the spring. The system is used to conduct psychophysical experiments that quantify human perception and discriminability of spring stiffness magnitude with and without vision and demonstrates the effectiveness of the device and the simulation for rendering springs. Experimental results show that spring magnitude perception follows a linear trend, and presence of vision enables better discrimination between different spring stiffnesses.

Acknowledgments

I would first like to thank my adviser, Prof. Ralph Hollis, for giving me the opportunity to work in the field of haptics, and for his guidance and support. I have benefited from his wealth of knowledge and have become a better researcher. I have also grown professionally under his observation. A huge thank you to Dr. Bertram Unger for the many hours he has spent discussing and refining my ideas, always being available to answer my questions, for imparting me with knowledge on every topic from haptics to medicine to world cultures, and for being a great friend. Thank you to Prof. Roberta Klatzky for her pschophysical expertise and to Prof. Robert Swendsen for lending his assistance in extending the spring model. Thanks to my other committee members Prof. Nancy Pollard and Clark Haynes. Thanks to Dr. Ben Brown for his assistance with the spring model.

I would also like to thank my friends at the Robotics Institute for making my graduate school experience fun and memorable. In particular, I would like to thank Ayorkor Mills-Tettey for always being there for me. Thanks also to my other friends around the world for their support and friendship. Finally, I would like to thank my parents and my sister for their undying support and motivation, and for always believing in me. My every achievement is the product of their love and encouragement.

This work was partially supported by National Science Foundation grant IIS-0413085 and by a Google Anita Borg scholarship.

Contents

1 Introduction

There has been substantial interest in the virtual environment community in the haptic and visual modeling of deformable objects [6], [18]. One method of rendering elastic behavior in these models is by using a network of "Mass-Spring" elements [7]. Such models over-simplify the behavior of a deformable object. Another popular method is that of Finite Element Analysis (FEA) [3]. While FEA accurately captures continuous and nonlinear deformable behavior, it is also computationally intensive and unsuitable for real-time simulations. James and Pai have suggested pre-calculation methods [10], [11] to make FEA models computationally efficient. Such pre-calculated models have been used by Barbič and James [2] to identify the principal components of the deformation model and combine them appropriately to render a range of deformations and Mahvash and Hayward have used them in combination with interpolation techniques [15]. While these methods render real-time deformations, they compromise on accuracy and fidelity of the haptic interaction. Also, the values of the parameters are specified during pre-calculation, making it difficult to change the model during run-time. In this report, a method of rendering real-time, realistic and accurate haptic and visual 3D helical springs based on a quasistatic analytical model is presented. Equations that define the behavior of the spring during compression, elongation, shear and tilting, and that predict the buckling point are provided. Such a simulation is an initial step in the use of a magnetic levitation haptic device (MLHD) to render deformable objects. The simulation allows parameterization of the spring structure (e.g., length; coil diameter), material (e.g., Young's modulus) and inherent qualities (e.g., compression rigidity constant). Users can feel the results of parametric variations using free exploration and these variations are mapped into perceptible properties. Here we consider how users freely explore and perceive one fundamental property of a spring: its stiffness.

Interaction with a deformable object using a MLHD is equivalent to active exploration of a compliant object by contact with a rigid surface. Srinivasan and LaMotte [16] showed that humans can be quite effective at discriminating the compliance of objects and surfaces. Sensory information for this task comes from two sources: the skin (cutaneous) and the muscles, tendons and joints (kinesthesis). Srinivasan and LaMotte [16] also showed that kinesthesis is required for the discrimination among levels of compliance when springs are covered with a rigid surface. Furthermore, LaMotte [13] showed that people can discriminate stiffness even while wielding a tool when allowed active control. Vision also contributes to stiffness perception, as shown by Wu et al. [19], sometimes compensating for systematic bias in the haptic system. Indeed, Bingham et al. [5] showed that vision alone is sufficient for identifying spring motion, raising the issue of whether rendered forces will add to the realism of a stiffness model in which visual feedback of spring motion is present.

We used two classic psychophysical procedures to characterize perception of stiffness and to gauge the effectiveness of the spring simulation. One is magnitude estimation, which assesses how internal responses to the stiffness co-vary with rendered values. The other is the just noticeable difference procedure (JND), which assesses how perceptible small differences in stiffness are and how the perceptibility varies with the base stiffness value. One common finding in many perceptual domains is that the JND threshold is a constant proportion of the base value, following what is commonly called Weber's law. That proportion is called the Weber fraction. Together, these measures describe stiffness perception over a broad range of supra-threshold values and at the limits of discrimination. Given the demands of fabricating real spring samples and the complexities of these procedures, it would not be possible to do such a study without the

rendering capabilities of a MLHD, which makes it possible to generate high-resolution stiffness values over a broad range in the context of the ongoing experiment.

The first section of this document provides the specifications of a MLHD that lends itself to render a realistic haptic spring. Next, the theory behind the 3D helical spring model, application of the theory to render the simulation, and handling of real-world issues are presented. The first subsection derives the equations for forces, moments and shape of the spring during compression that are stated in [14]. It also provides details on how this model was extended to derive the equations during elongation. The following two subsections explain how the theory was adapted to create the haptic and visual simulations. The last part of this section explains how the device was calibrated along the vertical axis. The next section describes the setup of the psychophysical experiments, their execution, and the obtained results. The last section interprets the results, lists the key contributions of the work, and suggests some future directions.

2 Magnetic Levitation Haptic Device

A MLHD consists of a handle that is attached to a magnetically levitated flotor, which has the shape of a hemisphere. Three photodiode sensors and LED markers are used to monitor the position of the flotor. A MLHD provides maximum stiffness, k_{max}, of approximately 25 N/mm in translation and 50.0 Nm/rad in rotation. This stiffness refers to the performance limit in unilateral constraints while rendering rigid surfaces. The maximum force and the maximum moment generated ranges between 55 N to 140 N and between 6.3 Nm to 12.2 Nm respectively depending on the axis. These characteristics make a MLHD appropriate for rendering springs since it is necessary in some cases to simulate high stiffnesses associated with "hard" springs and high moments experienced during buckling. Interaction with a 3D spring involves forces along the three axes of translation and moments about the three axes of rotation and thus requires a device with 6 degrees of freedom (DoFs). A MLHD satisfies this requirement since the 6-DoF motion of its handle has a range approximately that of comfortable fingertip motion with the wrist stationary ($\pm$12 mm translation and $\pm 7^{\circ}$ rotation in all directions). In addition, a MLHD provides real-time position and orientation information with resolutions of 5-10 μm and high position bandwidth ($\approx$125 Hz at $\pm$3 dB) [4].

3 Rendering a 3D Helical Spring

A 3D helical spring was haptically rendered using a MLHD. A corresponding visual simulation was rendered on a computer monitor. An analytical quasistatic model proposed by T.M. Lowery [14] was used to haptically and visually render the 3D helical spring. The model assumes a close-coiled helical compression spring with wire of circular cross section, and upper and lower end plates. The model provides a set of equations to predict the interactive forces and moments experienced at the plate ends in two dimensions in response to deformation of the spring. This model was extended to three dimensions and the derivations of the expressions are presented in the next few subsections. The extension to 3D was implemented by considering two 2D planes, $X-Y$ and $Y-Z$, and combining the computed forces and moments.

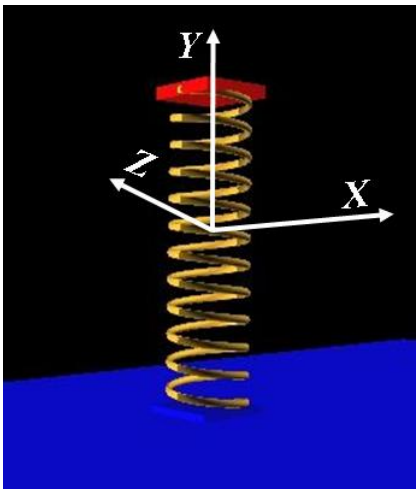

Figure 1: System of axes.

3.1 Derivation of Spring Equations

Lowery's [14] main assumption is that the spring behaves like an elastic rod. Haringx [9] explained that if during compression a helical spring has a sufficiently small pitch, the deformation of each coil of the spring can be approximated to that of an unclosed circular ring lying in a flat plane. Based on this approximation, the compressed helical spring can be replaced by a number of similar rings connected by perfectly rigid elements as shown in Fig. 2.

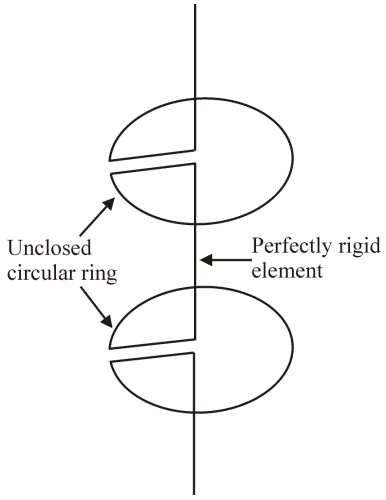

Figure 2: Approximation of coil of deformed helical spring as an unclosed ring connected by rigid elements.

Consider a plane of one such ring at $\{C_X, C_Y, C_Z\}$. Its cross-section without any additional forces acting on it is shown in Fig. 3. Let the moments, normal force, and shear forces at any point $\{C_X, C_Y, C_Z\}$, be represented as $M_{Z,X}$, N, and $Q_{X,Z}$. Subscripts for

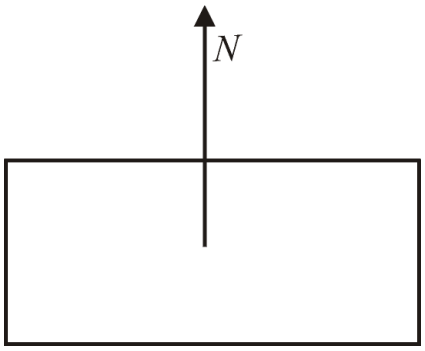

Figure 3: Plane of an unclosed ring.

forces refer to the respective translation axes and subscripts for moments refer to the respective rotation axes. Upon application of moments, the planes in which the rings are situated bisect the angle between two successive rigid elements. The plane is thus perpendicular to the spring centerline, as shown in Fig. 4a. Application of transverse forces deflect the ring in its plane such that the rigid elements are displaced in the direction of the transverse force and parallel to themselves, as shown in Fig. 4b. The planes of the rings are no longer perpendicular to the spring centerline.

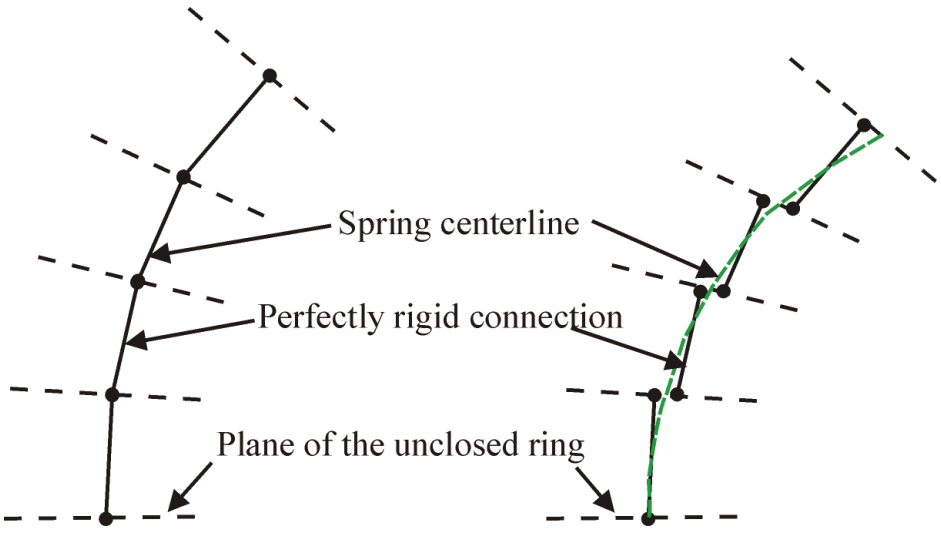

Figure 4: (a) Spring distorted by bending moments, (b) Spring distorted by bending moments and transverse forces.

Hence, the application of a vertical force, P, horizontal forces, $H_{X,Z}$, and moments, $M_{Z,X}$ cause the plane to bend by angles, $\psi_{Z,X}$, and the slope angles caused by shear alone are $\phi_{Z,X}$ about the Z and X axes respectively as shown in Fig. 5. The free length of the spring, L_0, changes to the compressed length, L. $L < L_0$ during compression and $L > L_0$ during elongation. At end conditions, $\psi_{Z,X}$ equals either $\psi_{Uz,x}$ or $\psi_{Lz,x}$.

A triangle of forces can be drawn as shown in Fig. 6. It can be seen that

$$\overrightarrow{N} = \overrightarrow{P} - \overrightarrow{H_{X,Z}}. \tag{1}$$

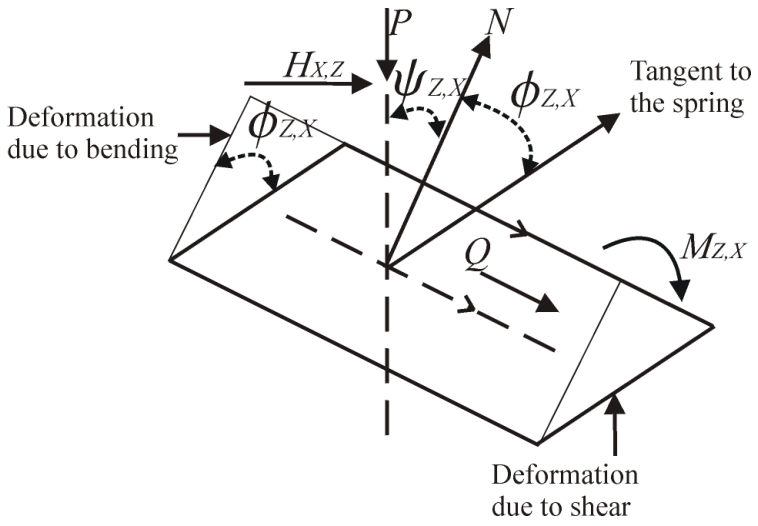

Figure 5: Bending and shearing of plane upon application of moments and forces.

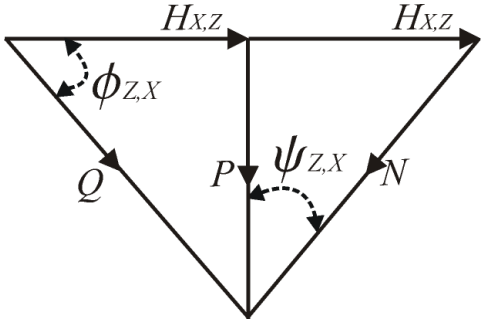

Figure 6: Triangle of forces acting on a coil plane.

The components of $\overrightarrow{P}$ and $\overrightarrow{H_{X,Z}}$ along $\overrightarrow{N}$ are $\overrightarrow{P}\cos(\psi_{Z,X})$ and $-\overrightarrow{H_{X,Z}}\sin(\psi_{Z,X})$. Therefore,

$$|\overrightarrow{N}| = |\overrightarrow{P}|\cos(\psi_{Z,X}) - |\overrightarrow{H_{X,Z}}|\sin(\psi_{Z,X}). \tag{2}$$

The direction of $\overrightarrow{N}$ is the same as that of the tangent to the spring. Also,

$$\overrightarrow{Q_{X,Z}} = \overrightarrow{P} + \overrightarrow{H_{X,Z}}. \tag{3}$$

Similarly, taking the components of $\overrightarrow{P}$ and $\overrightarrow{H_{X,Z}}$ along $\overrightarrow{Q_{X,Z}}$,

$$|\overrightarrow{Q_{X,Z}}| = |\overrightarrow{P}|\sin(\psi_{Z,X}) + |\overrightarrow{H_{X,Z}}|\cos(\psi_{Z,X}). \tag{4}$$

The direction of $\overrightarrow{Q_{X,Z}}$ is perpendicular to that of $\overrightarrow{N}$. When the angles are assumed to be small,

$$|\overrightarrow{N}| \approx |\overrightarrow{P}| - |\overrightarrow{H_{X,Z}}|\ \psi_{Z,X}, \text{ and} \tag{5}$$

$$|\overrightarrow{Q_{X,Z}}| \approx |\overrightarrow{P}|\ \psi_{Z,X} + |\overrightarrow{H_{X,Z}}|. \tag{6}$$

Table 1: Table of symbols

L_0	Free length of spring
L	Compressed length of spring
n	Number of active coils
d	Wire Diameter
D	Mean Coil Diameter
E	Young's Modulus
G	Shear Modulus
α	Rigidity constant with respect to bending
β	Rigidity constant with respect to shear
γ	Rigidity constant with respect to compression
k	Spring constant as defined by Hooke's law
P	Vertical reaction force
q	Buckling factor
$H_{X,Z}$	Horizontal reaction forces along X and Z axes respectively
$M_{Ux,z}$	Resisting moment at the upper end plate about X and Z axes respectively
$M_{Lx,z}$	Resisting moment at the lower end plate about X and Z axes respectively
ψ_X	Pitch angle of upper end plate = Angle of rotation about X axis
ψ_Z	Roll angle of upper end plate = Angle of rotation about Z axis
$\psi_{Ux,z}$	Angles of the upper spring centerline end about the X and Z axis respectively
$\psi_{Lx,z}$	Angles of the lower spring centerline end about the X and Z axis respectively
$\psi_{X,Z}$	Angle of spring centerline caused by bending alone about the X and Z axis respectively
$\phi_{X,Z}$	Slope of spring centerline caused by shear alone about the X and Z axis respectively
N	Force normal to cross-section of an equivalent elastic rod
$Q_{X,Z}$	Shear force along the X and Z axes respectively
$M_{X,Z}$	Moment about the X and Z axes respectively
V	Maximum vertical translation of MLHD
$T_{X,Y,Z}$	Translation along X, Y and Z axes respectively
$S_{X,Y,Z}$	Sensor data on translation along X, Y and Z axes respectively
$P_{buckling}$	Vertical force at which buckling occurs
$L_{buckling}$	Compressed length of the spring at which buckling occurs
$C_{X,Y,Z}$	3D coordinates of a point on the spring centerline

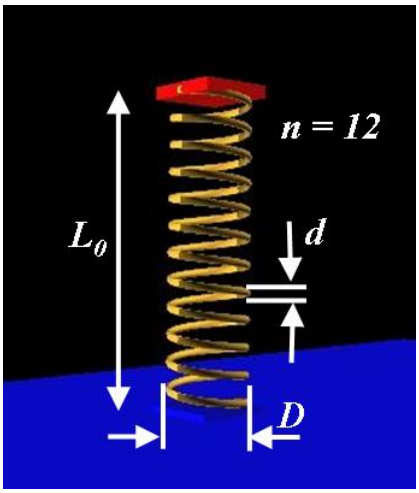

Figure 7: Physical parameters defining a spring.

From this point on, I will refer to $|\overrightarrow{K}|$ as simply K. The rigidity constants with respect to bending α, shear β, and compression γ, are given by

$$\alpha = \frac{L_0 d^4 E}{32nD\left(1+\frac{E}{2G}\right)}, \tag{7}$$

$$\beta = \frac{L_0 d^4 E}{8nD^3}, \text{ and} \tag{8}$$

$$\gamma = \frac{L_0 d^4 G}{8nD^3}, \text{ where} \tag{9}$$

α, β, and γ are the constants of proportionality relating applied moment and resulting curvature, applied shear force and resulting shear deformation, and applied vertical force and resulting compression, respectively [See Table I and Fig. 7 for symbol meanings]. The vertical reaction force is given by

$$\boxed{P = \gamma\frac{(L_0 - L)}{L_0}}, \tag{10}$$

which is the familiar spring equation based on Hooke's law, where spring constant

$$k = \frac{\gamma}{L_0}. \tag{11}$$

The moment, $M_{Z,X}$, acting at any point $\{C_X, C_Y, C_Z\}$ is the sum of the vertical force, P, acting at a perpendicular distance of $C_{X,Z}$, horizontal forces, $H_{X,Z}$, acting at a perpendicular distance of $(L - C_Y)$, and the moments, $M_{Uz,x}$, acting at the upper end plate of the spring, as shown in Fig. 8.
Therefore,

$$M_{Z,X} = C_{X,Z}P + H_{X,Z}\,(L - C_Y) + M_{Uz,x}. \tag{12}$$

At the upper end point when $C_X = 0, C_Y = L, C_Z = 0$,

$$M = M_{Uz,x}. \tag{13}$$

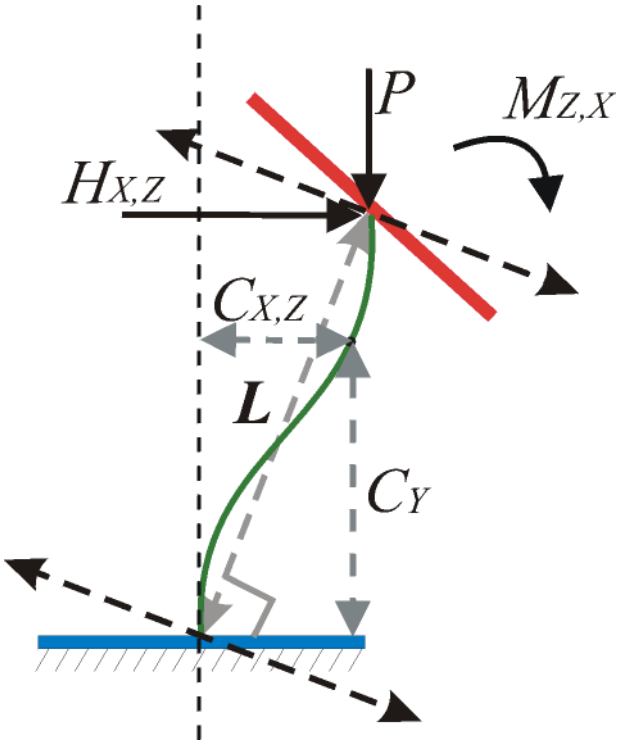

Figure 8: 2D view of forces and moments.

At the lower end point when $C_X = 0, C_Y = 0, C_Z = 0$,

$$M = H_{X,Z}L + M_{Uz,x}. \tag{14}$$

These two expressions match our definition of $M_{Uz,x}$ and $M_{Lz,x}$ as in Table 1. Moments ($M_{Z,X}$) and the rigidity constant for bending (α) are related by,

$$\frac{d\ \psi_{Z,X}}{d\ C_Y} \approx \frac{M_{Z,X}}{\alpha} = \frac{C_{X,Z}P + H_{X,Z}\ (L - C_Y) + M_{Uz,x}}{\alpha}. \tag{15}$$

Shear forces ($Q_{X,Z}$) and the rigidity constant for shear (β) are related by,

$$\phi_{Z,X} = \frac{Q_{X,Z}}{\beta} \approx \frac{P\psi_{Z,X}}{\beta} + \frac{H_{X,Z}}{\beta}. \tag{16}$$

The slope of the spring centerline is equal to the sum of the slopes due to bending and shear. Using (16),

$$-\frac{d\ C_{X,Z}}{d\ C_Y} = \psi_{Z,X} + \phi_{Z,X} = \left(1 + \frac{P}{\beta}\right)\psi_{Z,X} + \frac{H_{X,Z}}{\beta}. \tag{17}$$

Differentiating this equation with respect to C_Y and substituting (15),

$$-\frac{d^2\ C_{X,Z}}{d\ C_Y{}^2} = \left(1 + \frac{P}{\beta}\right)\left[\frac{PC_{X,Z}}{\alpha} + \left(\frac{H_{X,Z}(L - C_Y) + M_{Uz,x}}{\alpha}\right)\right]. \tag{18}$$

A spring buckles during compression when it deforms suddenly and nonlinearly along the vertical axis. The buckling factor is given by

$$q^2 = \frac{P}{\alpha}\left(1+\frac{P}{\beta}\right). \tag{19}$$

The buckling factor predicts the buckling of parallel plates by

$$qL_0 = 2\pi. \tag{20}$$

P is positive during compression and negative during elongation. Since α and β are positive constants, q^2 is also positive during compression and negative during elongation. At this point, the derivation diverges into compression and elongation.

3.1.1 Compression

P and q^2 are positive during compression. Rearranging (19),

$$1+\frac{P}{\beta} = \frac{q^2\alpha}{P}. \tag{21}$$

Substituting (21) in (18) and rearranging the terms,

$$\frac{d^2\, C_{X,Z}}{d\, C_Y{}^2} + q^2 C_{X,Z} = -q^2\left[\frac{H_{X,Z}(L-C_Y)+M_{Uz,x}}{P}\right]. \tag{22}$$

Let

$$A(C_Y) = \frac{H_{X,Z}(L-C_Y)+M_{Uz,x}}{P}. \tag{23}$$

Substituting (23) in (22),

$$\frac{d^2\, C_{X,Z}}{d\, C_Y{}^2} + q^2 C_{X,Z} = -q^2 A(C_Y). \tag{24}$$

A particular solution of the nonhomogeneous linear differential equation (24) is

$$C_p = -A(C_Y), \tag{25}$$

since

$$\frac{d^2\, A(C_Y)}{d\, C_Y{}^2} = 0, \text{ and} \tag{26}$$

$$\frac{d^2\, C_p}{d\, C_Y{}^2} + q^2 C_p = -q^2 A(C_Y). \tag{27}$$

The complementary function of (24) is determined by using the auxiliary equation of the second order homogeneous linear differential equation,

$$m^2 + q^2 = 0. \tag{28}$$

This implies that

$$m = \pm qi. \tag{29}$$

Hence, the complementary function is given by

$$C_{X,Z} = B\cos(qC_Y) + D\sin(qC_Y). \tag{30}$$

Using the superposition principle for nonhomogeneous equations, the general solution of (22) is obtained by combining the particular solution, (25), and the complementary function, (30),

$$C_{X,Z} = -A(C_Y) + B\cos(qC_Y) + D\sin(qC_Y). \tag{31}$$

Using the lower boundary condition,

$$B = A(0) = \frac{H_{X,Z}L + M_{Uz,x}}{P}. \tag{32}$$

Using (32) with the upper boundary condition,

$$-\frac{M_{Uz,x}}{P} + \frac{H_{X,Z}L + M_{Uz,x}}{P}\cos(qL) + D\sin(qL) = 0. \tag{33}$$

Solving for D,

$$D = \frac{M_{Uz,x}}{P\sin(qL)} - \frac{H_{X,Z}L + M_{Uz,x}}{P\tan(qL)}. \tag{34}$$

Therefore, the complete solution of (24) in the case of compression is

$$\begin{aligned} C_{X,Z} = &-\frac{H_{X,Z}(L - C_Y) + M_{Uz,x}}{P} + \frac{H_{X,Z}L + M_{Uz,x}}{P}\cos(qC_Y) + \\ &+ \left(\frac{M_{Uz,x}}{P\sin(qL)} - \frac{H_{X,Z}L + M_{Uz,x}}{P\tan(qL)}\right)\sin(qC_Y). \end{aligned} \tag{35}$$

Simplifying this expression by pulling out the common factor of $\left(\frac{1}{P}\right)$, and using the trigonometric identity

$$\frac{1}{\sin(\theta)} - \frac{1}{\tan(\theta)} = \tan\left(\frac{\theta}{2}\right), \tag{36}$$

$$\begin{aligned} C_{X,Z} = \frac{1}{P}\Bigg[&M_{Uz,x}\left(\tan\left(\frac{qL}{2}\right)\sin(qC_Y) + \cos(qC_Y) - 1\right) + \\ &+ H_{X,Z}L\left(\frac{-\sin(qC_Y)}{\tan(qL)} + \cos(qC_Y) + \frac{C_Y}{L} - 1\right)\Bigg]. \end{aligned} \tag{37}$$

This expression gives the X and Z coordinates of a point along the spring centerline. 2π number of points per coil are used to define the spring centerline. The number of turns in a spring, n, stays constant with change in compressed length, L. What changes is the vertical height of each turn. Therefore, the Y coordinate, C_Y, for each of these $2n\pi$ points is given by

$$\boxed{C_Y = \frac{i * L}{2n\pi}, \quad i : 0 \rightarrow 2n\pi} \ , \text{ and} \tag{38}$$

The next step is to find an expression for $\psi_{Z,X}$. Using (37) and differentiating $C_{X,Z}$ with respect to C_Y, we get

$$\begin{aligned} \frac{d\,C_{X,Z}}{d\,C_Y} = \frac{1}{P}\Bigg[&M_{Uz,x}\left(q\tan\left(\frac{qL}{2}\right)\cos(qC_Y) - q\sin(qC_Y)\right) + \\ &+ H_{X,Z}L\left(\frac{-q\cos(qC_Y)}{\tan(qL)} - q\sin(qC_Y) + \frac{1}{L}\right)\Bigg]. \end{aligned} \tag{39}$$

Using (17), we get

$$\psi_{Z,X} = -\left(\frac{d\,C_{X,Z}}{d\,C_Y} + \frac{H_{X,Z}}{\beta}\right)\left(\frac{\beta}{P+\beta}\right). \tag{40}$$

Substituting (39) in (40), and rearranging the terms, we get

$$\begin{aligned}\psi_{Z,X} = -\frac{1}{P}\left(\frac{\beta}{P+\beta}\right)\Bigg[M_{Uz,x}\left(q\tan\left(\frac{qL}{2}\right)\cos(qC_Y) - q\sin(qC_Y)\right) + \\ + H_{X,Z}L\left(\frac{-q\cos(qC_Y)}{\tan(qL)} - q\sin(qC_Y)\right)\Bigg] - \frac{H_{X,Z}}{P}.\end{aligned} \tag{41}$$

Taking the reciprocal of (19), and substituting in (41),

$$\begin{aligned}\psi_{Z,X} = -\frac{1}{\alpha q}\Bigg[M_{Uz,x}\left(\tan\left(\frac{qL}{2}\right)\cos(qC_Y) - \sin(qC_Y)\right) - \\ - H_{X,Z}L\left(\frac{\cos(qC_Y)}{\tan(qL)} + \sin(qC_Y)\right)\Bigg] - \frac{H_{X,Z}}{P}.\end{aligned} \tag{42}$$

[Note: Expression (42) is very similar to the expression given by Lowery [14] for $\psi_{Z,X}$ if α is substituted with its expression in (7). The main difference is that the term $\left(-\frac{H_{X,Z}}{P}\right)$ is within the square braces in [14]. We believe that this is the typo because our derivation tells otherwise, and also such an expression as given by [14] cannot lead to forthcoming expressions, which are identical with those given by [14].]

The value of $\psi_{Z,X}$ at the lower boundary condition is

$$\psi_{Z,X}(0) = \psi_{Lz,x} = -\frac{1}{\alpha q}\left[M_{Uz,x}\tan\left(\frac{qL}{2}\right) - H_{X,Z}L\cot(qL)\right] - \frac{H_{X,Z}}{P}. \tag{43}$$

The value of $\psi_{Z,X}$ at the upper boundary condition is

$$\begin{aligned}\psi_{Z,X}(L) = \psi_{Uz,x} &= -\frac{1}{\alpha q}\Bigg[M_{Uz,x}\left(\tan\left(\frac{qL}{2}\right)\cos(qL) - \sin(qL)\right) - \\ &\quad - H_{X,Z}L\left(\frac{\cos(qL)}{\tan(qL)} + \sin(qL)\right)\Bigg] - \frac{H_{X,Z}}{P} \qquad (44) \\ &= \frac{1}{\alpha q}\left[M_{Uz,x}\tan\left(\frac{qL}{2}\right) + H_{X,Z}L\csc(qL)\right] - \frac{H_{X,Z}}{P}, \text{ since} \\ &\quad \tan\left(\frac{\theta}{2}\right)\cos\theta - \sin\theta = -\tan\left(\frac{\theta}{2}\right). \qquad (45)\end{aligned}$$

Adding $\psi_{Lz,x}$ and $\psi_{Uz,x}$,

$$\psi_{Lz,x} + \psi_{Uz,x} = \frac{H_{X,Z}L}{\alpha q}\cot\left(\frac{qL}{2}\right) - \frac{2H_{X,Z}}{P}, \text{ since} \tag{46}$$

$$\cot(\theta) + \csc(\theta) = \cot\left(\frac{\theta}{2}\right). \tag{47}$$

Solving for $H_{X,Z}$,

$$\boxed{H_{X,Z} = \frac{P(\psi_{Lz,x} + \psi_{Uz,x})}{\frac{LP}{q\alpha\tan\left(\frac{qL}{2}\right)} - 2}}\ . \tag{48}$$

Using the value of $H_{X,Z}$ in (44), $M_{Uz,x}$ is solved for as

$$\boxed{M_{Uz,x} = \left[\left(\frac{H_{X,Z}}{P} + \psi_{Uz,x}\right)\alpha q - \frac{H_{X,Z}L}{\sin(qL)}\right]\cot\left(\frac{qL}{2}\right)} \ . \tag{49}$$

The moment acting at the lower end plate, $M_{Lz,x}$, was previously computed in (14). It is restated here for the sake of convenience

$$\boxed{M_{Lz,x} = H_{X,Z}L + M_{Uz,x}} \ .$$

3.1.2 Elongation

The value of q^2 is negative during elongation. Consider a variable, $\widetilde{q}$, such that

$$\widetilde{q}^{\,2} = \pm q^2. \tag{50}$$

During compression when $q^2 > 0$, $\widetilde{q}^{\,2} = |q^2|$, and during elongation when $q^2 < 0$, $\widetilde{q}^{\,2} = -|q^2|$. In other words, during both compression and elongation

$$\widetilde{q}^{\,2} = \left|\frac{P}{\alpha}\left(1 + \frac{P}{\beta}\right)\right|. \tag{51}$$

The elongation equivalent of (24) is

$$\frac{d^2\, C'_{X,Z}}{d\, C_Y{}^2} - \widetilde{q}^{\,2} C'_{X,Z} = \widetilde{q}^{\,2} A(C_Y). \tag{52}$$

The particular solution of this nonhomogeneous differential equation stays the same as $C_p = -A(C_Y)$. The auxiliary equation is however,

$$m^2 - \widetilde{q}^{\,2} = 0, \tag{53}$$

which implies that

$$m = \pm\widetilde{q}. \tag{54}$$

The corresponding complementary function is

$$C'_{X,Z} = B' \cosh(\widetilde{q}C_Y) + D' \sinh(\widetilde{q}C_Y), \tag{55}$$

and the general solution of (52) is

$$C'_{X,Z} = -A(C_Y) + B' \cosh(\widetilde{q}C_Y) + D' \sinh(\widetilde{q}C_Y). \tag{56}$$

Using the lower boundary condition,

$$B' = B = A(0) = \frac{H_{X,Z}L + M_{Uz,x}}{P}. \tag{57}$$

Using the upper boundary condition,

$$D' = \frac{M_{Uz,x}}{P\sinh(qL)} - \frac{H_{X,Z}L + M_{Uz,x}}{P\tanh qL}. \tag{58}$$

The complete solution is therefore,

$$C_{X,Z}^{'} = -\frac{H_{X,Z}(L-C_Y)+M_{Uz,x}}{P} + \frac{H_{X,Z}L+M_{Uz,x}}{P}\cosh(\widetilde{q}C_Y) + \\ + \left(\frac{M_{Uz,x}}{P\sinh(\widetilde{q}L)} - \frac{H_{X,Z}L+M_{Uz,x}}{P\tanh(\widetilde{q}L)}\right)\sinh(\widetilde{q}C_Y). \tag{59}$$

The simplified complete expression for $C_{X,Z}^{'}$, or $C_{X,Z}$ during elongation, using the trigonometric identity

$$\frac{1}{\sinh(\theta)} - \frac{1}{\tanh(\theta)} = -\tanh\left(\frac{\theta}{2}\right) \quad \text{is} \tag{60}$$

$$C_{X,Z}^{'} = \frac{1}{P}\left[M_{Uz,x}\left(-\tanh\left(\frac{\widetilde{q}L}{2}\right)\sinh(\widetilde{q}C_Y)+\cosh(\widetilde{q}C_Y)-1\right) + \\ + H_{X,Z}L\left(\frac{-\sinh(\widetilde{q}C_Y)}{\tanh(\widetilde{q}L)}+\cosh(\widetilde{q}C_Y)+\frac{C_Y}{L}-1\right)\right]. \tag{61}$$

The expression for C_Y stays the same as in the case of compression, (38). Differentiating $C_{X,Z}^{'}$ with respect to C_Y,

$$\frac{d\,C_{X,Z}^{'}}{d\,C_Y} = \frac{1}{P}\left[M_{Uz,x}\left(-\widetilde{q}\tanh\left(\frac{\widetilde{q}L}{2}\right)\cosh(\widetilde{q}C_Y)+\widetilde{q}\sinh(\widetilde{q}C_Y)\right) + \\ + H_{X,Z}L\left(\frac{-\widetilde{q}\cosh(\widetilde{q}C_Y)}{\tanh(\widetilde{q}L)}+\widetilde{q}\sinh(\widetilde{q}C_Y)+\frac{1}{L}\right)\right]. \tag{62}$$

Substituting (62) in (40), and rearranging the terms,

$$\psi_{Z,X}^{'} = -\frac{1}{P}\left(\frac{\beta}{P+\beta}\right)\left[M_{Uz,x}\left(-\widetilde{q}\tanh\left(\frac{\widetilde{q}L}{2}\right)\cosh(\widetilde{q}C_Y)+\widetilde{q}\sinh(\widetilde{q}C_Y)\right) + \\ + H_{X,Z}L\left(\frac{-\widetilde{q}\cosh(\widetilde{q}C_Y)}{\tanh(\widetilde{q}L)}+\widetilde{q}\sinh(\widetilde{q}C_Y)\right)\right] - \frac{H_{X,Z}}{P}. \tag{63}$$

Taking the reciprocal of (51), and substituting in (63),

$$\psi_{Z,X}^{'} = -\frac{1}{\alpha\widetilde{q}^{\,2}}\left[M_{Uz,x}\left(-\widetilde{q}\tanh\left(\frac{\widetilde{q}L}{2}\right)\cosh(\widetilde{q}C_Y)+\widetilde{q}\sinh(\widetilde{q}C_Y)\right) + \\ + H_{X,Z}L\left(\frac{-\widetilde{q}\cosh(\widetilde{q}C_Y)}{\tanh(\widetilde{q}L)}+\widetilde{q}\sinh(\widetilde{q}C_Y)\right)\right] - \frac{H_{X,Z}}{P}. \tag{64}$$

The value of $\psi_{Z,X}^{'}$ at the lower boundary condition is

$$\psi_{Z,X}^{'}(0) = \psi_{Lz,x}^{'} = \frac{1}{\alpha\widetilde{q}^{\,2}}\left[M_{Uz,x}\,\widetilde{q}\,\tanh\left(\frac{\widetilde{q}L}{2}\right)+H_{X,Z}\,L\,\widetilde{q}\,\coth(\widetilde{q}L)\right] - \frac{H_{X,Z}}{P}. \tag{65}$$

The value of $\psi_{Z,X}^{'}$ at the upper boundary condition is

$$\psi_{Z,X}^{'}(L) = \psi_{Uz,x}^{'} = -\frac{1}{\alpha\widetilde{q}^{\,2}}\left[M_{Uz,x}\left(-\widetilde{q}\tanh\left(\frac{\widetilde{q}L}{2}\right)\cosh(\widetilde{q}L)+\widetilde{q}\sinh(\widetilde{q}L\right) + \right. \\ \left. + H_{X,Z}L\left(\frac{-\widetilde{q}\cosh(\widetilde{q}L)}{\tanh(\widetilde{q}L)}+\widetilde{q}\sinh(\widetilde{q}L)\right)\right] - \frac{H_{X,Z}}{P} \\ = -\frac{1}{\alpha\widetilde{q}^{\,2}}\left[M_{Uz,x}\,\widetilde{q}\,\tanh\left(\frac{\widetilde{q}L}{2}\right)-H_{X,Z}\,L\,\widetilde{q}\,\operatorname{csch}(\widetilde{q}L)\right] - \frac{H_{X,Z}}{P}, \text{ since} \tag{66}$$

$$-\tanh\left(\frac{\theta}{2}\right)\cosh\theta+\sinh\theta=\tanh\left(\frac{\theta}{2}\right),\ \text{and} \tag{67}$$

$$-\frac{\cosh(\theta)}{\tanh(\theta)}+\sinh\theta=\frac{-\cosh^2(\theta)+\sinh^2(\theta)}{\sinh(\theta)}=\frac{-1}{\sinh\theta}=-\text{csch}(\theta). \tag{68}$$

Adding $\psi^{'}_{Lz,x}$ and $\psi^{'}_{Uz,x}$,

$$\psi^{'}_{Lz,x}+\psi^{'}_{Uz,x}=\frac{H_{X,Z}\,L\,\widetilde{q}}{\alpha\widetilde{q}^{\,2}}\left[\coth\left(\frac{qL}{2}\right)\right]-\frac{2H_{X,Z}}{P},\ \text{since} \tag{69}$$

$$\coth(\theta)+\text{csch}(\theta)=\coth\left(\frac{\theta}{2}\right). \tag{70}$$

Now, $\widetilde{q}$ is always positive since it is the square-root of a value. Hence, $\widetilde{q}=q$. However, during elongation $\widetilde{q}^2=-q^2$. Substituting these values in (69) and solving for $H_{X,Z}$,

$$\boxed{H^{'}_{X,Z}=\frac{P(\psi^{'}_{Uz,x}+\psi^{'}_{Lz,x})}{\frac{-LP}{q\alpha\tanh\left(\frac{qL}{2}\right)}-2}}\ . \tag{71}$$

Using the value of $H^{'}_{X,Z}$ in (44), $M_{Uz,x}$ during elongation is solved for as

$$\boxed{M^{'}_{Uz,x}=\left[\left(\frac{H^{'}_{X,Z}}{P}+\psi^{'}_{Uz,x}\right)\alpha q+\frac{H^{'}_{X,Z}L}{\sinh(qL)}\right]\coth\left(\frac{qL}{2}\right)}\ . \tag{72}$$

The switch from compression to elongation occurs at $L=L_0$. As $L\to L_0, P\to 0$ and $H_{X,Z}\to\infty$. In order to calculate the horizontal forces at this limit, consider (46)

$$\psi_{Lz,x}+\psi_{Uz,x}=\frac{H_{X,Z}L}{\alpha q}\cot\left(\frac{qL}{2}\right)-\frac{2H_{X,Z}}{P}.$$

Rearranging the terms,

$$\frac{\psi_{Lz,x}+\psi_{Uz,x}}{H_{X,Z}}=\frac{L}{\alpha q}\cot\left(\frac{qL}{2}\right)-\frac{2}{P}. \tag{73}$$

For small values of P, q^2 is also small [See (19)] and so is $\left(\frac{qL}{2}\right)$. Expanding the $\cot\left(\frac{qL}{2}\right)$ term as a series of infinite sums,

$$\begin{aligned}\frac{\psi_{Lz,x}+\psi_{Uz,x}}{H_{X,Z}}&=\frac{L}{\alpha q}\left[\frac{2}{qL}-\frac{1}{3}\left(\frac{qL}{2}\right)+\cdots\right]-\frac{2}{P}\\&=\frac{2}{\alpha}\left[\frac{1}{q^2}-\frac{L^2}{12}+\cdots\right]-\frac{2}{P}.\end{aligned} \tag{74}$$

Substituting the reciprocal of (19) in (74),

$$\begin{aligned}\frac{\psi_{Lz,x}+\psi_{Uz,x}}{H_{X,Z}} &= \frac{2}{\alpha}\left[\frac{\alpha}{P}\left(\frac{\beta}{\beta+P}\right)-\frac{L^2}{12}+\cdots\right]-\frac{2}{P}\\ &= \frac{2}{P}\left[-1+\frac{\beta}{\beta+P}-\frac{PL^2}{12\alpha}+\cdots\right]\\ &= \frac{2}{P}\left[\frac{-P}{\beta+P}-\frac{PL^2}{12\alpha}+\cdots\right]\\ &= -\frac{2}{\beta+P}-\frac{L^2}{6\alpha}+\cdots\end{aligned} \tag{75}$$

Rearranging the terms in (75),

$$H_{X,Z} = \frac{\psi_{Lz,x}+\psi_{Uz,x}}{\left(-\frac{2}{\beta+P}-\frac{L_0^2}{6\alpha}+\cdots\right)}. \tag{76}$$

As $P \to 0$, $L \to L_0$, and

3.2 Haptic Rendering

The setup for the virtual simulation is that of a vertically positioned spring with its lower end plate fixed to the ground, as shown in Fig. 9. The user deforms the spring by applying forces and moments to the handle of a MLHD, which represents the upper end plate.

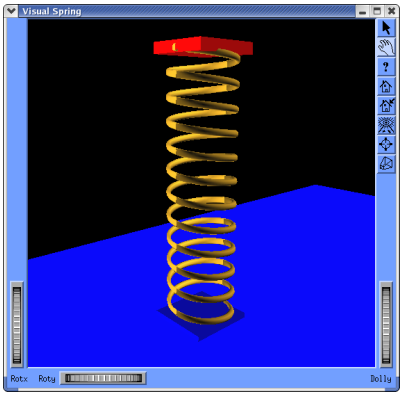

Figure 9: 3D view of undeformed spring.

The servo loop runs at a rate of 1000 Hz and the same rate is used to record the translation and rotation data of the handle. The simulation records the real-time translation and rotation data as input and calculates the forces and moments using the expressions derived in the previous subsections. Expressions such as (15), (16), (42), and (64) are not used explicitly but were presented for the purpose of deriving other expressions. The physical parameters are set to default values at the start of the simulation. The parameters that need to be set and their default values are $L_0 = 25$mm, $d = 0.63$mm, $D = 6.07$mm, $n = 11.8$, $E = 203395$MPa $= 203395$N/mm^2, and $G = 75842$MPa. They

can be modified at run-time using a panel that looks like Fig. 10. This panel also allows change in control parameters such as proportional and dampening constants. These parameters determine the translational and rotational limits of the spring and the behavior of the spring at these limits.

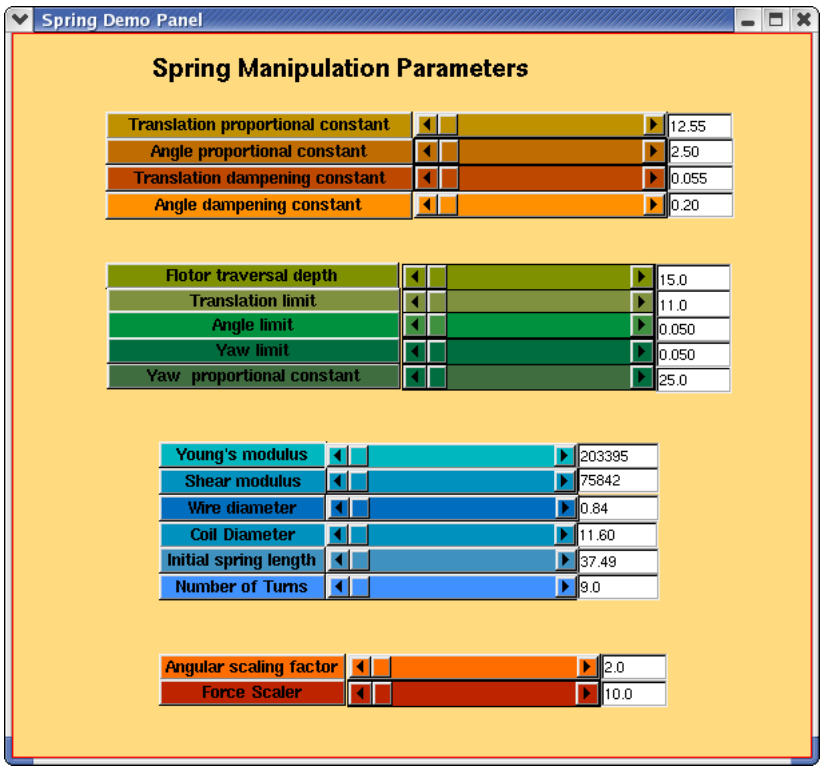

Figure 10: Panel for modifying physical parameters.

However, all the expressions are in terms of the other set of constants - the rigidity constants. This allows manipulation of the spring's characteristics through the rigidity constants as shown in Fig. 11, in addition to manipulation through the physical parameters. This is possible since all three rigidity constants can be calculated from the physical parameters. In the current setup, changes in the physical parameters using the panel in Fig. 10 cause changes in the rigidity constants and are reflected in the panel in Fig. 11. But due to the underconstrained nature of the relationship between the rigidity constants and the physical parameters the reverse does not take place, i.e. changes in rigidity constants using the panel in Fig. 11 do not result in changes in the panel in Fig. 10.

Manipulation through the rigidity constants was predominantly used in our simulation because our first study was on how subjects perceived change in stiffness of the spring, which related directly to the rigidity constants. In particular we wanted to analyse the perception of the spring constant, k, which is related to γ by (11). The default values of the rigidity constants were $\alpha = 179.15Nmm^2$, $\beta = 45.52N$, and $\gamma = 16.97$. The first step in the simulation was to transform the data into a form that is usable by the simulation. The downward vertical motion range of the handle, V, is approximately 15mm. In the simulation, this depth corresponds to the free length of any

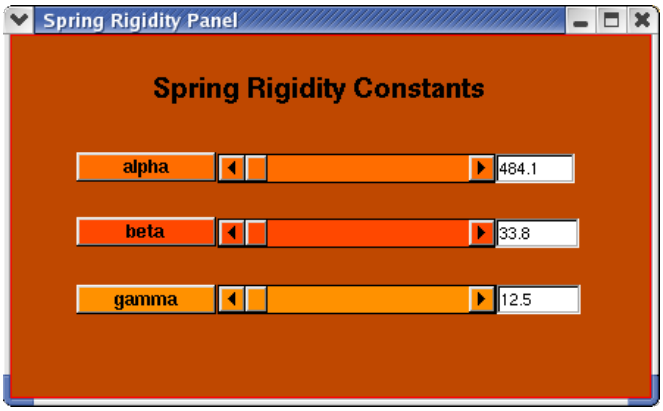

Figure 11: Panel for modifying rigidity constants.

spring, L_0. Therefore, the vertical and horizontal translations obtained from the position sensors, $S_{X,Y,Z}$, are first transformed from real world coordinates to simulation coordinates, $T_{X,Y,Z}$, by

$$T_{X,Y,Z} = (S_{X,Y,Z}) \left(\frac{L_0}{V} \right). \tag{77}$$

T_Y is negative during compression and positive during elongation. This enables the user to move the upper end plate through L_0 to reach the lower end plate in the simulation and at the same time reach the bottom of the motion range of the handle in the real world. The coordinate system used in this document refers to the Y axis as the vertical axis, while the X and Z axes define the horizontal plane. Furthermore, the simulation coordinates are vertically offset by L_0 such that the simulation coordinates of the lower end plate are transformed from $\{0, -L_0, 0\}$ to $\{0, 0, 0\}$. The compressed length of the spring is then calculated as the Euclidean distance between the upper end point of the spring centerline, $\{T_X, L_0 + T_Y, T_Z\}$, and the lower end point of the spring centerline, $\{0, 0, 0\}$,

$$L = \sqrt{T_X^2 + (L_0 + T_Y)^2 + T_Z^2}. \tag{77}$$

The pitch and roll angles of the flotor handle are computed from the sensor data. They are the angles of rotation of the upper end plate of the spring about the Z and X axes respectively (ψ_Z, ψ_X). The pitch and roll angles of the lower end plate are zero since it is held fixed. The angles of the spring centerline ends (ψ_{Ux}, ψ_{Lx}, ψ_{Uz} and ψ_{Lz}) are the angles made by the normals to the end points of the spring centerline with the respective end plates. The first subscript refers to the end plate and the second subscript refers to the rotation axis. Angles of rotations about the Z axis are used to compute horizontal forces along X-axis, and vice versa. These angles are computed as

$$\psi_{Lz,x} = \tan^{-1} \left(\frac{T_{X,Z}}{T_Y} \right), \text{ and} \tag{77}$$

$$\psi_{Uz,x} = \psi_{Lz,x} - \psi_{Z,X}, \tag{77}$$

and are shown in Fig. 12. Using the computed value of L, the specified values of L_0 and γ, the vertical reaction force is first calculated as in (10). Based on whether $L < L_0$, $L > L_0$, or $L_0 + \eta < L < L_0 - \eta$, $H_{X,Z}$, $H'_{X,Z}$ or $H_{X,Zlimit}$ are calculated using (48), (71),

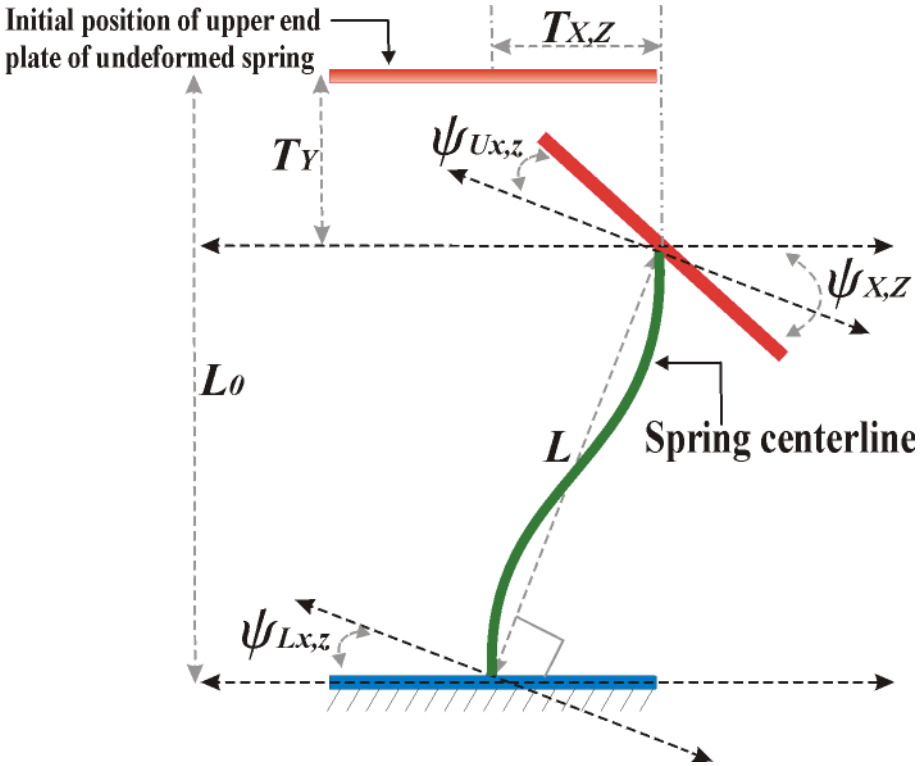

Figure 12: 2D view of spring angles and translations under deformation.

or (76). η is an empirical value that is used since the limiting value holds for very small values of P, in addition to when $L = L_0$. Using the computed horizontal reaction forces, $M_{Uz,x}$ or $M'_{Uz,x}$ are calculated, using (49) or (72). There is no special expression for the moments in the limiting case - the appropriate expression is used based on whether η is positive or negative. In all cases, (14) is used to calculate $M_{Lz,x}$.

Certain changes were made to the prescribed values of the forces and moments. The resisting moment experienced upon rotation about the spring's vertical axis (yaw) is not defined by the analytical model in [14]. In this simulation it is modeled as a resisting moment proportional to the rotation. A feedforward force with magnitude equal to the weight of the flotor (7.7N) and in opposite direction to gravity is added to the computed value of P while assigning the outputs. A difference between [14] and our simulation is that that the lower end plate is our simulation is not capable of rotation. This issue is addressed by making the computed forces in the simulation undergo a transformation of axes through ψ_{Lz} and then through ψ_{Lx}. Commutativeness of axes transformations is assumed since the angles are small. The moments experienced at the upper end plate as a result of these transformations are the sum of $M_{Uz,x}$ and $M_{Lz,x}$.

If the calculated forces and moments with the default physical parameters and rigidity constant values are rendered using a MLHD, harsh forces are experienced upon compression to a certain length. The compressed length at which this occurred was invariant over trials. After ensuring that there were no other forces or moments acting, I hypothesised that the observed phenomenon was buckling. The forces, moments, and shape of the spring coil after buckling was not predicted by [14] and hence the harsh

forces. The first step was to predict when buckling occurred. Using (19) and (20), the value of P at which buckling occurs is

$$P_{buckling} = \frac{\beta}{2}\left(-1+\sqrt{1+\frac{16\pi^2\alpha}{\beta L_0^2}}\right) \tag{77}$$

Using (10), the compressed length at which buckling occurs can be expressed as

$$L_{buckling} = L_0\left[1-\frac{\beta}{2\gamma}\left(-1+\sqrt{1+\frac{16\pi^2\alpha}{\beta L_0^2}}\right)\right]. \tag{77}$$

The buckling point depends on the specific settings of the spring parameters. For example, a spring of lower stiffness has its buckling point higher than a very stiff spring, relative to the lower end plate. The behavior of the spring after buckling is not defined in [14]. A temporary solution was to set the horizontal forces and moments to zero and to scale down the vertical force. This renders an illusion of the spring bottoming out after buckling. Unseating of the spring, or slipping of the lower end plate, does not occur in this simulation because the lower end plate is held fixed. In some springs, the buckling point is so low relative to the lower end plate that it occurs at a compressed length lower than that at which the spring coils touch each other completely. In this case, the touching of the spring coils prevents further compression and hence such a spring never buckles. The case in which the spring coils touch each other completely is handled by rendering a high vertical reaction force when $L = nd$.

A spherical boundary was implemented to keep the flotor within its sensor range. A heuristic for the radius of the spherical boundary was the maximum vertical compression possible, $L_0 - L$, after which the spring coils touch each other completely. The boundary was modeled as a proportional-derivative controlled repulsive boundary. The derivative gain was set as a variable depending on the stiffness of the spring. The angle of rotation was limited to 2.86°. This limitation was necessary to keep the flotor within the sensor range in the cases where the resisting moments are very high.

Spring weight, seat friction, end-coil effects and the dynamics of a spring are not included in the quasi-static model. However, due to the 580g mass of the flotor, the simulation brings out dynamic characteristics of a spring such as oscillations with damping effect. For example, if the spring is lightly compressed and released, the frequency and amplitude of the resulting dampened oscillation motion is less than that when the spring is compressed with a larger force.

3.3 Visual Rendering

The spring coil was built in three stages: (1) definition of the spring centerline based on Lowery's model, (2) definition of a helical curve based on the spring centerline, and (3) definition of a helix with a circular cross-section based on the helical curve, as shown in Fig. 13. These definitions are used to update the graphics at a rate of 30 Hz. The graphics of the 3D spring was rendered using Open Inventor [8]. The transformation of the translation data captured by the sensors, and the calculation of the vertical force, horizontal forces and resisting moments are identical to those used for haptic rendering. These data are then used to calculate the 3D coordinates of each point in the spring centerline as given by (38), and (37) or (61). These values also undergo a transformation of axes through ψ_{Lz} and then through ψ_{Lx}.

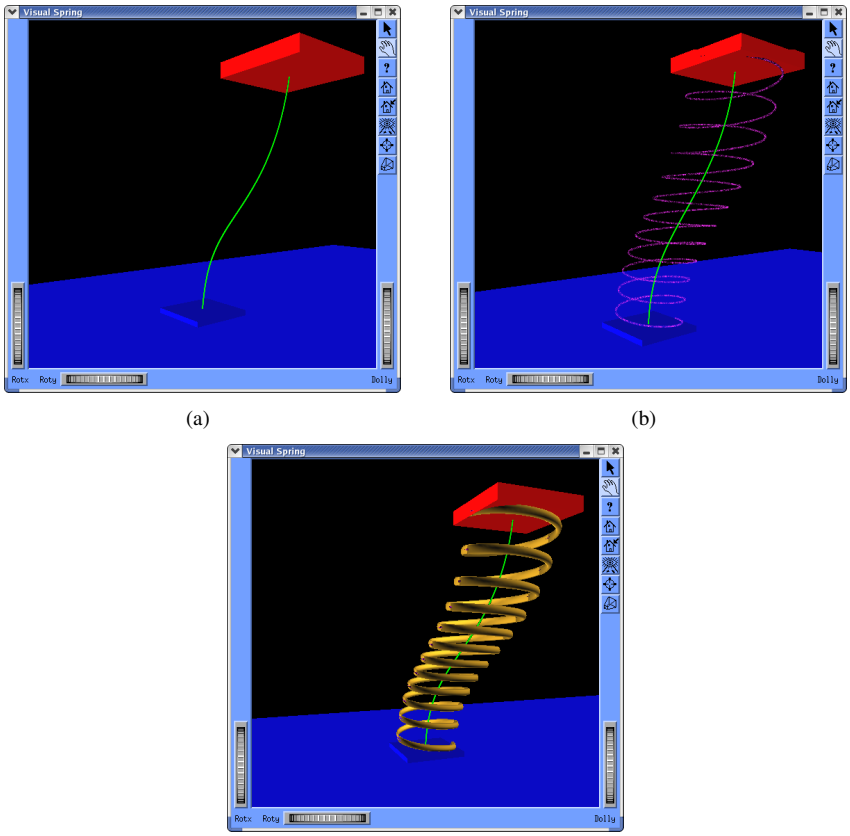

Figure 13: Building a visual spring: (a) Spring centerline (b) Helical curve (c) Helix with a circular cross-section.

A static helix of one coil is defined as $\{\cos(i),\ i,\ \sin(i)\}$, where $i : 0 \rightarrow 2\pi$. A helix of diameter D and with each point in its centerline having coordinates $\{C_X,\ C_Y,\ C_Z\}$ is defined as

$$\left\{\frac{D\cos(i)}{2} + C_X,\ C_Y,\ \frac{D\sin(i)}{2} + C_Z\right\},\quad i : 0 \rightarrow 2n\pi. \tag{78}$$

In Open Inventor, the helix is rendered using a NURBS curve, with as many points as there are in the spring centerline, and with a matrix of the coordinates of each point in the helix. For a helix with a circular cross-section, circles defined by eight equiangular points were drawn about each point in the helical curve. The corresponding points on each of the circles are connected using Open Inventor's QuadMesh class, with as many points as there are in the spring centerline, and with a matrix of the eight equiangular points of each circle. The number of points in the spring centerline are large enough to render a seamless helix and small enough to render a real-time simulation on an AMD XP 2000+ computer with a nVIDIA GeForce4 TI 4600 graphics card. In our simulation there were as many points such that the distance between each of the points is 0.1 units vertically.

3.4 Calibration along the Vertical Axis

Accurate and realistic haptic rendering of a deformable object required calibration of the MLHD along the vertical axis. The flotor of a MLHD has 6 coils as shown in Fig. 14. Each coil lies between a fixed magnet assembly as shown in Fig. 15. When the current passes through a coil, a Lorentz force is applied to the flotor. In other words, Lorentz forces are generated where the current loops of the six actuator coils intersect with the magnetic flux loops. Hence, the magnetic field and thereby the Lorentz forces obtained from the coil currents are dependent on the position of the coil in the air gap. Due to the large air gap in each magnet assembly these fields are not uniform in the given workspace, which leads to a nonlinear force vs. displacement curve [4].

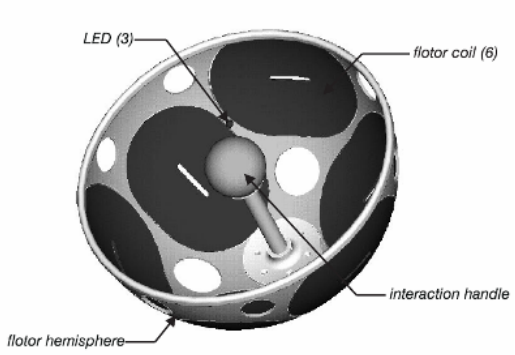

Figure 14: Arrangement of coils in a flotor of a MLHD.

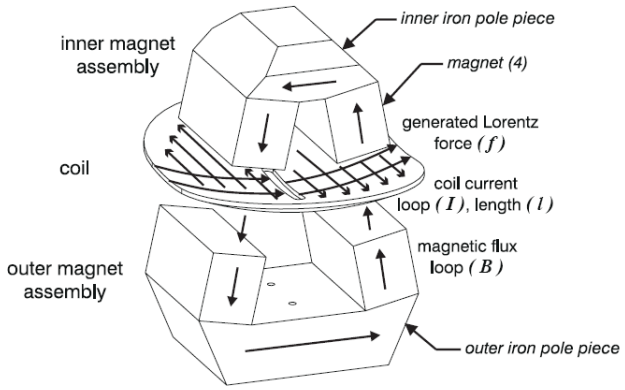

Figure 15: Single magnet assembly within a MLHD.

To address this issue the device was calibrated and the first step was to plot the nonlinearity by commanding a constant desired feedforward force, using a force sensor to measure the actual rendered force for every 1 mm of displacement, and repeating this process for different values of feedforward force. First, a physical setup was built to hold the force sensor.A plate with screw holes was inserted through the hole of the MLHD handle. An identical plate was placed on top of the handle. These plates were clamped together with two pairs of screws and bolts. The rest of the setup included erector sets and face plates to hold the force sensor and was designed in such a way

that it had a stable base. The setup was then placed on top of the MLHD. The weight of the flotor was measured as 6.9N and the two plates, screws and bolts added a weight of 1.68N, making the net weight of the flotor as 8.58N. Next, API functions were used to levitate the flotor in a manner suitable for calibration. A calibration mode was programmed, in which all the axes besides the vertical axis are locked using an appropriate API command, and a specified feedforward force is commanded. The feedforward force was set at a value greater than that required to counter the weight of the flotor. Lastly, the force sensor was then lowered so that it sits just on top of the plate that is attached on top of the handle. The force sensor is then lowered by a millimeter at a time. The reading on the force sensor and the values given by the position sensors (incorporated into the calibration mode) were noted. This step was repeated for 14mm for two iterations of feedforward values of 10N and 12N. This gave four force versus displacement curves. Had the magnetic fields of the MLHD been uniform in the given workspace, the force readings would have been a constant value of the difference between the commanded feedforward force and the weight of the flotor. However, the error caused by the nonlinearity was found to be approximately 10.87%.

Each of the four curves were first denormalised by subtracting each curve by its mean. Next, the nonlinear error curve was parameterized by fitting a second-degree polynomial function to it. This normalised polynomial was found using MATLAB [17] as shown in Fig. 16 and is expressed as

$$p(x) = -0.00451x^2 - 0.02583x + 0.19157, \quad \text{where} \tag{78}$$

x is the normalised displacement.

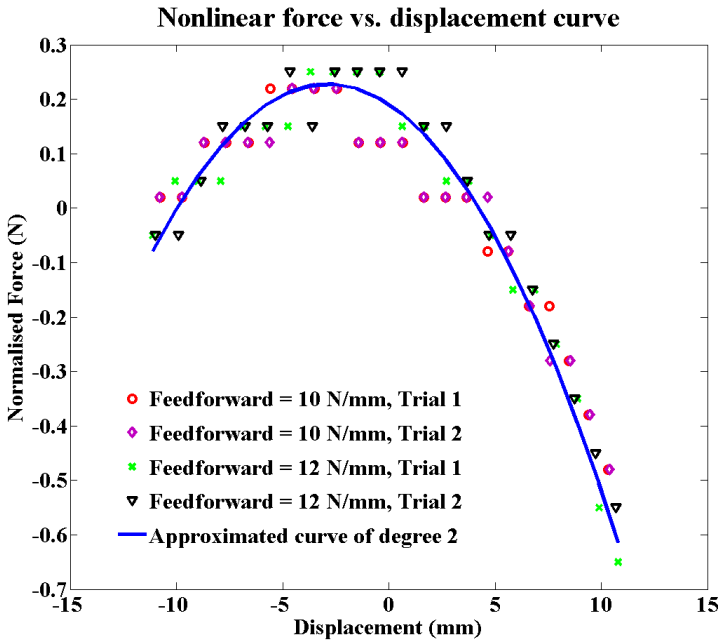

Figure 16: Nonlinear Error Curve.

If $p(x)$ is inverted and applied to each of the normalised force readings, a straight line at $\sim$1 is obtained. In order to be applied to the run-time unnormalised force values,

$p(x)$ was denormalised by adding the mean value of each curve to itself. This mean value is unknown during run-time. Hence, it was approximated as the commanded feedforward force. If the denormalised polynomial curve is inverted and multiplied to the calculated vertical force values, a straight line at $\sim$1 is obtained. Hence, it is then multiplied by the mean force value or the commanded feedforward force. The final equation for calibration is

$$P_{calibrated} = \frac{P \times FF}{p(x) + FF}, \text{ where} \tag{78}$$

FF is the commanded feedforward force.

4 PSYCHOPHYSICAL EXPERIMENTS

Implementation of an analytical model of the spring allowed for easy modification of the spring parameters. Such a spring can be defined either in terms of its physical properties (L_0, n, d, D, E, and G) or in terms of its rigidity constants (α, β, γ). It can be seen from (4), (5), and (6) that α and β vary only with γ if D, G and E were assumed to be constant. Using this property, two psychophysical experiments, Spring Stiffness Magnitude Estimation and JND of Spring Stiffness, were implemented using springs of varying γ. Two modalities were presented in each experiment: vision and haptics (VH), where the subject could see a graphic representation of the haptic spring, and haptics-alone (H), where the window displaying the visual spring was hidden. The visual spring did not change in appearance (e.g., coil thickness or length) with change in γ to prevent the use of visual size and shape cues. However, the visual spring does reflect the motion and deformable characteristics of the haptic spring. The subjects were allowed to modify their view of the visual spring by zooming in or out, rotating the spring about its vertical axis, and changing the angle of inclination of the plane upon which the spring rested. The subjects wore headphones playing white noise during the experiments to minimize auditory influences. A warning in the form of a beep (audible over the noise) and printed message was given if the subject applied a force against the virtual boundary sufficient to push the flotor out of the device's sensor range.

4.1 Spring Stiffness Magnitude Estimation

Sixteen students (6 females and 10 males) from Carnegie Mellon University served as subjects. Two were left-handed and the rest were right-handed by self-report. Eight of the subjects started in the VH modality and the rest started in the H modality, constituting an order variable.

Twelve springs with γ ranging uniformly from 12.0 N to 48.0 N (corresponding to k ranging uniformly from 0.48 N/mm to 1.92 N/mm for L_0 = 25.0 mm) were presented to each subject in random order. A subject started in a particular modality and went through three replications of the 12 randomly ordered springs and was then presented with three replications in the other modality. Each set of replications was preceded by a demo of 5 springs in the same modality, sampled from the range of γs experienced and included the maximum and minimum values of γ. The subject was asked to rate the springs using any number, with the rule that a higher number meant that the spring felt stiffer. The range was self-selected by the subject; analysis of the data included normalization to account for inter-subject variability in the range. With 3 replications of 12

γ values in two modalities, there were 72 trials total per subject, lasting approximately 20 minutes.

4.2 Just Noticeable Difference of Spring Stiffness

Sixteen students (7 females and 9 males) from Carnegie Mellon University served as subjects. Two were left-handed and the rest were right-handed by self-report. Eight of the subjects started in VH modality and the other eight started in H modality, constituting an order variable.

A version of Kaernbach's unforced weighted up-down adaptive threshold estimation was used to rapidly determine the JND [12]. According to this technique, subjects are asked to compare between a base γ_B and a comparison γ_C. A correct decision reduces the difference δ between γ_B and γ_C by stepsize $D1$. An incorrect decision increases δ by a stepsize of $D2$, and an indeterminate answer increases δ by a stepsize of $D3 < D2$. These values were proportional to $D1$ specified by the algorithm with a goal of converging to an accuracy of 75%. The initial value of $D1$ was chosen as 25% of the initial δ, which was chosen as 40% of γ_B; these values were pre-tested to confirm convergence in a reasonable time. Under the algorithm, as the experiment progresses γ_C moves towards γ_B and reaches an equilibrium after a certain number of reversals in the direction of δ. The stepsizes are halved at the 2^{nd} and 4^{th} reversals. Equilibrium is assumed after occurence of the 8^{th} reversal and the JND is calculated as the mean of all δs between the 4^{th} and the 8^{th} reversal.

The experiment consisted of 3 γ_Bs of 17.0 N, 26.0 N and 35.0 N (corresponding to ks of 0.68 N/mm, 1.04 N/mm and 1.4 N/mm for L_0 = 25.0 mm). Four replications of alternating modalities were conducted per subject. The base values followed a non-repeating Latin-square order between replications, and the order within a replication was randomly chosen from the 6 possible permutations. The experiment was preceded by a demo of 4 springs in each modality in mixed order, which together sampled the range of γ and the types of comparisons (one spring obviously stiffer than the other, one spring close but distinguishable from the other, and two very similar springs) the subject would experience. The experiment took approximately 1 hour per subject, during which the subject felt about 200 pairs of springs.

4.3 Results

4.3.1 Spring Stiffness Magnitude Estimation

For the magnitude estimation task, the normalized magnitude rating were analyzed with an ANOVA on modality (2, VH and H), order (2: VH first; H first), and γ (12). The effect of modality did not approach significance [F(1, 14) $<$ 0.02, p = 0.88] nor was the effect involving order significant [F(1, 14) $<$ 0.823, p = 0.198]. The sole effect was that of γ [F(11, 154) = 13.57, $p < 0.001$], as shown in Fig. 17. This reflected an increase in judged magnitude with rendered γ that was essentially linear (R^2 for linear trend = 0.983).

The positions of the handle and forces applied by the subjects during the experiment were recorded at 1000 Hz. From these data, the variables of mean velocity, acceleration and force along the vertical axis were analyzed with an ANOVA on γ (12), and modality (2, VH and H). The effect of γ was clearly seen in the variables of mean velocity [F(11,77) = 9.25, $p < 0.001$], acceleration [F(11, 77) = 3.73, $p < 0.001$], and force

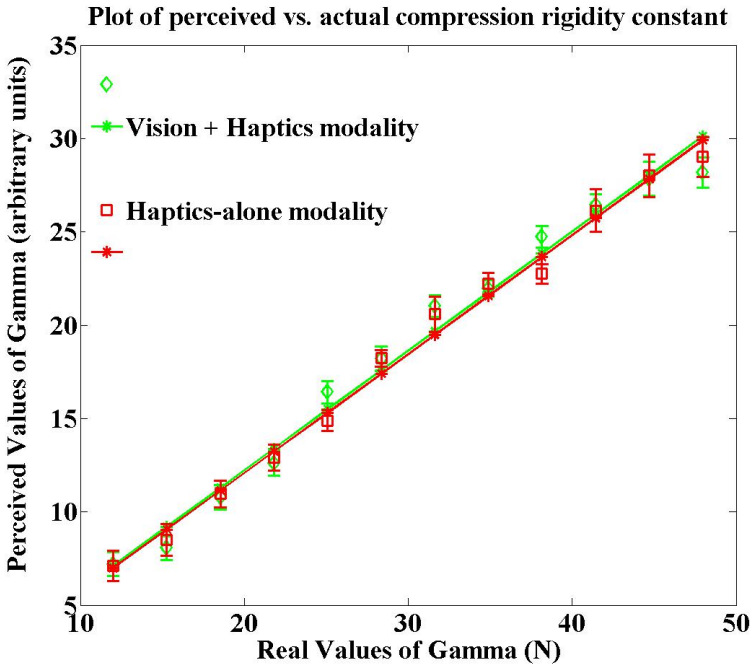

Figure 17: Plot of perceived γ vs. real γ in Vision+Haptics and Haptics-alone modalities.

[F(11,77) = 11.92, $p < 0.001$] along the vertical axis. However, these three variables were not affected by modality.

4.3.2 Just Noticeable Difference of Spring Stiffness

For the JND task, the threshold values were analyzed with an ANOVA on modality (2, VH and H), γ_B (3), and order (2: VH first; H first). This analysis showed effects of modality [F(1, 14 = 11.94, p = 0.004] and γ_B [F(2, 28) = 23.01, $p < 0.001$], but no significant effect involving order, as shown in Fig. 18.

The hypothesis that the JND is a constant fraction of γ_B for each modality would predict a modality by γ_B interaction, which approached significance [F(2, 28) = 3.10, p = 0.061]. In a subsidiary analysis testing this hypothesis, it was found that JND expressed as a proportion of γ_B was statistically invariant across γ_B values for both the VH [F(2,30) = 1.46, p = 0.25] and H conditions [F(2,30) = 0.80, p = 0.46]. The average JND as a proportion of γ_B, i.e., the "Weber fraction," was 14.2% for VH versus 17.2% for H, and these differed reliably [t(15) = 3.30, p = 0.005 (two-tail)].

The variables of mean velocity, acceleration and force along the vertical axis were analyzed with an ANOVA on γ_B (3), and modality (2, VH and H). The findings were that velocity shows only an effect of γ_B [F(2,12) = 5.13, p = 0.025], acceleration shows no effect of γ_B or modality, and force shows an effect only of γ_B [F(2,12) = 13.18, p = 0.001].

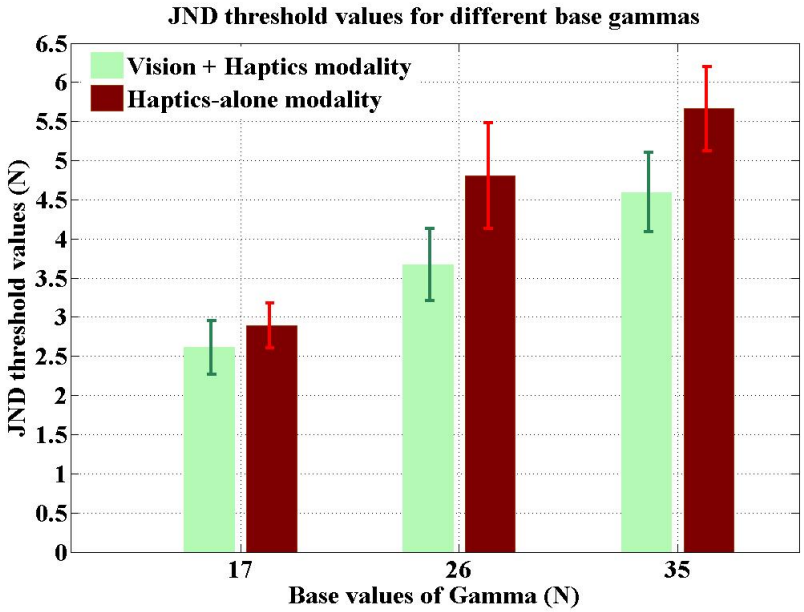

Figure 18: Bar graph of threshold values for different γ_Bs in Vision+Haptics and Haptics-alone modalities.

5 Conclusions

From the stated results, it was found that perceived stiffness increased linearly with rendered stiffness across the full range studied here. Clearly, participants in the experiment were able to discriminate and evaluate the rendered stiffness well. The importance of kinesthesis is demonstrated by the finding that velocity and acceleration are directly proportional and force is inversely proportional to the rendered γ. This means that the level of active control of the spring varied with the rendered γ and implies that it helped in perception of stiffness. Moreover, visual information did not modulate the judged stiffness value, indicating that it relied completely on the haptic rendering.

In contrast, although vision did not affect the sense of stiffness, visual cues did improve people's ability to discriminate between two stiffness values at the difference threshold. The differential stiffness required to discriminate between two springs increased by over 20% if vision was eliminated. The findings that kinesthetic responses in the JND experiment depended only on the rendered γ and not on modality further imply that visual cues were used to improve the performance in the task of discriminating between springs. A significant finding was the value of Weber fraction for spring stiffness as 14.2% or 1/7 with both visual and haptic sensory information, and 17.2% or 1/5.8 with haptic sensory information alone.

During debriefing of subjects, many of them stated that they explored the spring using vertical oscillatory up and down motions, which agrees with our findings and is predicted in [16]. Some commented that they started making their decisions using a certain strategy (compression in most cases) and if the comparisons became difficult, they tried an additional strategy such as elongation or making note of when buckling occurred. Almost all subjects commented that the haptic and visual simulations were

very realistic. These results demonstrate the effectiveness of a MLHD in rendering a deformable spring.

6 Key Contributions

In summary, the key contributions of this thesis are:

- Developed a haptic and visual rendering of a 3D coil spring using a Magnetic Levitation Haptic Device
- Derived equations for describing the behavior of a spring during elongation
- Determined that perception of spring stiffness magnitude follows a linear trend
- Showed that presence of vision enables better discrimination of spring stiffness
- Demonstrated effectiveness of a Magnetic Levitation Haptic Device in rendering a deformable object

7 Future Work

An improved haptic and visual model for post-buckling spring behavior would increase the effectiveness and realism of the simulation. Also, an increase in the limit on the angle of rotation would allow greater exploration of the spring's characteristics. A variation of the current implementation of the spring model with a lower end plate that is not held fixed would allow for unseating. Such a model, along with the current implementation, would allow for an analysis of how users perceive buckling and unseating and the role of each in maintaining stability of the spring. It would be interesting to investigate how people exploit the free exploration provided by the device in solving the stiffness question. This can be followed by a study to understand how people perceive the change in a spring's characteristics, such as stiffness, with respect to physical parameters (L_0, n, d, D, E, and G) and the two other rigidity constants of the spring (α, β). Comparison of the results of the psychophysical experiments with virtual springs to parallel experiments with real springs is an important further step. An immediate application of the simulation of the 3D spring is for educational purposes to develop a "feel" for the role of different materials and structural parameters in determining the stiffness and behavior of a helical spring.

The Weber fraction for weight discrimination is 1/53 [1]. It would be interesting to investigate why this value is a fraction of the Weber fraction for spring stiffness discrimination. Such research might show the effect of applying force versus experiencing force on discrimination, depending on the method used to determine the JND of weight.

In conclusion, this work is a concrete example of the ability of a MLHD to render a deformable object based on an analytical model and to quantify human perception of its stiffness.

References

[1] *A Dictionary of Psychology 2001*. Oxford University Press, 2001.

[2] J. Barbic and D. L. James. Real-Time Subspace Integration of St.Venant-Kirchhoff Deformable Models. In *ACM Transactions on Graphics (ACM SIGGRAPH 2005)*, 2005.

[3] K. Bathe. *Finite Element Procedures*. Prentice Hall, New Jersey, 1996.

[4] P. J. Berkelman and R. L. Hollis. Lorentz Magnetic Levitation for Haptic Interaction: Device Design, Performance, and Integration with Physical Simulations. *International Journal of Robotics Research*, Vol. 19, No. 7:644–667, July 2000.

[5] G. P. Bingham, R. C. Schmidt, and L. D. Rosenblum. Dynamics and the Orientation of Kinematic Forms in Visual Event Recognition. *Journal of Experimental Psychology*, Vol. 21, No. 6:1473–1493, 1995.

[6] M. de Pascale, G. de Pascale, and D. Prattichizzo. Haptic and Graphic Rendering of Deformable Objects based on GPUs. In *IEEE 6th Workshop on Multimedia Signal Processing*, 2004.

[7] A. Frisoli, L. F. Borelli, C. Stasi, M. Bellini, C. Bianchi, E. Ruffaldi, G. Di Pietro, and M. Bergamasco. Simulation of real-time deformable soft tissues for computer assisted surgery. *The International Journal of Medical Robotics & Computer Assisted Surgery*, 1, Issue 1:107–113, 2004.

[8] Silicon Graphics, Inc., 1140 E. Arques Ave., Sunnyvale, CA, and USA. Open inventor, http://oss.sgi.com/projects/inventor/.

[9] J. A. Haringx. On Highly Compressible Helical Springs and Rubber Rods, and their Application for Vibration-free Mountings, I. *Philips Research Reports*, Vol. 3, NR 6:401–449, 1948.

[10] D. L. James and D. K. Pai. Accurate Real Time Deformable Objects. In *SIGGRAPH 99 Conference Proceedings*, pages 65–72, 1999.

[11] D. L. James and D. K. Pai. A Unified Treatment of Elastostatic and Rigid Contact Simulation for Real Time Haptics. *Haptics-e, the Electronic Journal of Haptics Research*, 2, No. 1, 2001.

[12] C. Kaernbach. Adaptive threshold estimation with unforced-choice tasks. *Perception and Psychophysics*, Vol. 63:1377–1388, 2001.

[13] R. H. LaMotte. Softness Discrimination With a Tool. *Journal of Neurophysiology*, Vol. 83:1777–1786, 2000.

[14] T. M. Lowery. *How to predict buckling and unseating of coil springs*, pages 56–60. McGraw-Hill Book Company, 1961.

[15] M. Mahvash and V. Hayward. Haptic Simulation of a Tool in Contact with a Nonlinear Deformable Body. In *IS4TM: Int'l Symp. Surgery Simulation and Soft Tissue Modelling*, pages 311–320. Springer Verlag, 2003.

[16] M. A. Srinivasan and R. H. LaMotte. Tactual Discrimination of Softness. *Journal of Neurophysiology*, Vol. 73, No. 1:88–101, 1995.

[17] The Mathworks, Inc. MATLAB. 1994-2007.

[18] P. Volino, P. Davy, U. Bonanni, C. Luible, N. Magnenat-Thalmann, M. Mkinen, and H. Meinander. From measured physical parameters to the haptic feeling of fabric. *The Visual Computer: International Journal of Computer Graphics*, Vol. 23, Issue 2:133–142, 2007.

[19] W. Wu, C. Basdogan, and M. A. Srinivasan. Visual, haptic, and bimodal perception of size and stiffness in virtual environments. *ASME Dynamic Systems and Control*, DSC-Vol. 67:19–26, 1999.

A Source Code for Haptic Rendering of Spring

The code presented here is function springControl() in the file vvaradha@dulcian:teleop_src_wdir/maglev/icntrl.c.

```
/* ml_icntrl.c */
/* simple pid rpy control module with setpoint to coriolis pos */
/* *** more consistency on pass vs. global vars */

#include <math.h>
#include <stdio.h>
#include "ml.h"
#include "icntrl.h"
#include "ml_boxinbox.h"
#include "setControls.h"

/****************************** Externed Globals ****************/

double rotlimkp = 100.0;//working value was 100.0;
double rotlimkv = 0.7;//working value was 0.7;
double transpring =.08;// working value was 0.08;

float wireDiameter = 0.63;    // Wire diameter of spring 1
//float wireDiameter = 0.84;     // Wire diameter of spring 2
//float wireDiameter = 0.51;     // Wire diameter of spring 3

float coilDiameter = 6.07;     // Mean coil diameter of spring 1
//float coilDiameter = 11.6;     // Mean Coil diameter of spring 2
//float coilDiameter = 4.81;     // Mean Coil diameter of spring 3

float shearModulus = 75842;   // Shear modulus of spring 1
float youngModulus = 203395;    // Young's modulus of spring 1

float L0 = 25;                  // Free initial length of spring 1
//float L0 = 37.49;               // Free initial length of spring 2
//float L0 = 27.00;               // Free initial length of spring 3

float flotor_depth = 15;      // Maximum flotor depth in real world coordinates

float n = 11.8;               // Number of turns of spring 1
//float n = 9;                  // Number of turns of spring 2
//float n = 15;                   // Number of turns of spring 3

float alpha = 179.1;          // Rigidity constant wrt moment of spring 1
float beta = 45.5;            // Rigidity constant wrt shear of spring 1

float gamma = 16.97;         // Rigidity constant wrt compression of spring 1
//float gamma = 12.53;      // Rigidity constant wrt compression of spring 2
//float gamma = 10.32;      // Rigidity constant wrt compression of spring 3

float scaler = 10;
int change = 0;       // 0: change in gamma, 1: change in alpha, 2: change in beta,
// 3: change in physical params

//FILE* spring_datafile;

/****************************************************************/

/****************************** Local globals *******************/

double prev_des_pos[6] = {0.0,0.0,0.0,0.0,0.0,0.0};

const double local_windup[2] = {50.0,1.0};
```

```
double pid_ramp=PID_DFLT_RAMP; // speed of ramp up to windup_clamp;
double pid_wndupClamp[2]={PID_PDFLT_WNDUP,PID_ODFLT_WNDUP}; // windup clamps

/*********************************************************************/

int springControl(double int_fwpos[6],shstruct *s)
{
  // Physical parameters set at previous points in the code are now
  // assigned to symbols for easy use in the code that follows
  float d = wireDiameter;
  float D = coilDiameter;
  float G = shearModulus;
  float E = youngModulus;

  float PI = 3.14159265;

  //Declaration of variables
  float P;           // Vertical reaction force
  float q;           // Buckling factor
  float Hz;          // Horizontal reaction force along the Z axis
  float Hx;          // Horizontal reaction force along the X axis
  float M1z;         // Resisting moment of the upper end plate about the Z axis
  float M1x;         // Resisting moment of the upper end plate about the X axis
  float M2z;         // Resisting moment of the lower end plate about the Z axis
  float M2x;         // Resisting moment of the lower end plate about the X axis

  float siZu;        // Angle made by upperbar about Z-axis,
    // measured from X axis - Roll
  float siXu;        // Angle made by upperbar about X-axis,
    // measured from Z axis - Pitch
  float siYu;        // Angle made by upperbar about Y-axis - Yaw

  float si1X;        // Angle by normal to upper end of spring centerline
    // with upper end plate about X axis
  float si1Z;        // Angle by normal to upper end of spring centerline
    // with upper end plate about Z axis
  float si2X;        // Angle by normal to lower end of spring centerline
    // with upper end plate about X axis
  float si2Z;        // Angle by normal to lower end of spring centerline
    // with upper end plate about Z axis

  float L;           // Compressed length of spring
  float compressY;   // Compression along Y-axis
  float shearX;      // Shear along X-axis
  float shearZ;      // Shear along Z-axis

  float Lvertical;   // Compressed length along the Y-axis

  float forceY;     // Reactive force along the Y-axis
  float forceX;     // Reactive force along the X-axis
  float forceZ;     // Reactive force along the Z-axis
  float inter_forceY; // Intermediate force value used during transformation of axes

  float polyval;    // Value of polynomial calculated for calibration

  float angle_limit;  // Limiting value of roll and pitch angle so that flotor
     // does not go out of sensor range
  float yaw_limit;    // Limiting value of yaw angle so that flotor
     // does not go out of sensor range

  float buckling_P;  // Value of compression force at which buckling occurs
```

```
float buckling_length; // Value of compressed length at which buckling occurs

//*************************************************************************
// Calculation of run-time variables
compressY = s->fwpos[2]*L0/flotor_depth;   // Simulation coordinates of translation
     // along Y axis; -ve value if compression
     // and +ve if elongation
shearX = s->fwpos[0]*L0/flotor_depth; // Along X axis
shearZ = s->fwpos[1]*L0/flotor_depth; // Along Z axis

siXu = s->fwpos[3];     // Pitch
siZu = -s->fwpos[4];    // Roll
siYu = s->fwpos[5];     // Yaw

Lvertical = L0 + compressY; // Free length of spring + translation along Y axis
// Euclidean distance between upper end of spring centerline
// and lower end of spring centerline
L = sqrt(pow((L0+compressY),2)+pow(shearX,2)+pow(shearZ,2));

si2Z = atan(shearX/Lvertical);
si1Z = si2Z - siZu;

si2X = atan(shearZ/Lvertical);
si1X = si2X - siXu;

switch(change)
{
case 0: // Change in gamma
  // beta and alpha are directly proportional to gamma,
  // assuming D, E and G are constant
  beta = gamma*E/G;
  alpha = gamma*pow(D,2)/(4*G/E+2);
  break;
case 1: // Change in alpha
  beta = alpha*(4+2*E/G)/pow(D,2);
  gamma = alpha*(4*G/E+2)/pow(D,2);
  break;
case 2: // Change in beta
  alpha = beta*pow(D,2)/(4+2*E/G);
  gamma = beta*G/E;
  break;
case 3: // Change in physical parameters
  alpha = L0*pow(d,4)*E/(32*n*D*(1+E/(2*G)));
  beta = L0*pow(d,4)*E/(8*n*pow(D,3));
  gamma = L0*pow(d,4)*G/(8*n*pow(D,3));
 break;
}

buckling_P = (-beta+sqrt(pow(beta,2)+16*pow(PI,2)*alpha*beta/pow(L0,2)))/2;
buckling_length = L0-buckling_P*L0/gamma;

// Radius of the spherical boundary. It is set to the value at which the
// spring coils would touch each other during compression
s->spring_limit[0] = (L0-n*d)*flotor_depth/L0;

//*************************************************************************
// Calculation of force and moment values

if(L <= buckling_length) // If compression is beyond buckling point
  {
    // This is a rudimentary implementation of handling buckling
    forceY = gamma*(L0-L)/(10*L0); // Force along Y axis is reduced to 10 times
```

```
        // less than that computed at that point
      forceX = 0; // No horizontal forces
      forceZ = 0;
      M1x = 0; // No moments
      M2x = 0;
      M1z = 0;
      M2z = 0;

      P = forceY;
      angle_limit = 0.01; // Angle value is limited to 0.01 radians -
         // effectively, no rotation is allowed
      yaw_limit = 0.01;
    }
  else if(L > L0 & L > 0) // Elongation
    {
      angle_limit = s->spring_limit[2];
      yaw_limit = s->spring_limit[3];

      P = gamma*(L0-L)/L0;
      q = sqrt(-(P/alpha)*(1+P/beta));
      Hx = P*(si1Z+si2Z)/((-L*P/(tanh(q*L/2)*alpha*q))-2);
      Hz = P*(si1X+si2X)/((-L*P/(alpha*q*tanh(q*L/2)))-2);

      if(L < L0+1e-4) // Limit of H as L->L0 and P->0
{
  Hx = -beta*(si1Z+si2Z)/(2*(1+beta*pow(L,2)/(12*alpha)));
  Hz = -beta*(si1X+si2X)/(2*(1+beta*pow(L,2)/(12*alpha)));
}

      // Transformation of axes from Lowery's model since we assume
// that lower end plate is held fixed
      forceX = Hx*cos(si2Z)+P*sin(si2Z);
      inter_forceY = P*cos(si2Z)-Hx*sin(si2Z);
      forceZ = Hz*cos(si2X)+inter_forceY*sin(si2X);
      forceY = inter_forceY*cos(si2X)-Hz*sin(si2X);

      M1z = ((Hx/P+si1Z)*alpha*q+Hx*L/sinh(q*L))/tanh(q*L/2);
      M1x = ((Hz/P+si1X)*alpha*q+Hz*L/sinh(q*L))/tanh(q*L/2);
      M2z = M1z + Hx*L;
      M2x = M1x + Hz*L;
    }
  else if(L <= L0 & L > 0) // Compression
    {
      angle_limit = s->spring_limit[2];
      yaw_limit = s->spring_limit[3];

      P = gamma*(L0-L)/L0;
      q = sqrt((P/alpha)*(1+P/beta));

      Hx = P*(si1Z+si2Z)/(L*P/(tan(q*L/2)*alpha*q)-2);
      Hz = P*(si1X+si2X)/(L*P/(tan(q*L/2)*alpha*q)-2);

      if(L > L0-1e-4) // Limit of H as L->L0 and P->0
{
  Hx = -beta*(si1Z+si2Z)/(2*(1+beta*pow(L,2)/(12*alpha)));
  Hz = -beta*(si1X+si2X)/(2*(1+beta*pow(L,2)/(12*alpha)));
}

      // Transformation of axes from Lowery's model since
      // we assume that lower end plate is held fixed
      forceX = Hx*cos(si2Z)+P*sin(si2Z);
      inter_forceY = P*cos(si2Z)-Hx*sin(si2Z);
```

```
    forceZ = Hz*cos(si2X)+inter_forceY*sin(si2X);
    forceY = inter_forceY*cos(si2X)-Hz*sin(si2X);

    M1z = (alpha*q*(Hx/P+si1Z)-Hx*L/sin(q*L))/tan(q*L/2);
    M1x = (alpha*q*(Hz/P+si1X)-Hz*L/sin(q*L))/tan(q*L/2);
    M2z = M1z + Hx*L;
    M2x = M1x + Hz*L;
  }

// Calibration along vertical axis
polyval = -0.00451*pow(s->fwpos[2],2)-0.02583*s->fwpos[2]+0.1915;
forceY *= s->feedfor/(s->feedfor+polyval);

// Assign force values
// Add feedforward force to counter weight of flotor
s->force[2] = forceY + s->feedfor;
s->force[0] = forceX;
s->force[1] = forceZ;
s->force[3] = M1x+M2x; // Moment felt at upper end plate is sum of
  // calculated M1 and M2 since lower end plate
  // is held fixed
s->force[4] = -M1z-M2z; // Sign change
s->force[5] = 0; // Assigned as 0 and calculated later

// Slightly dampen the moments to avoid oscillations caused by small disturbances
s->force[3] = - s->Kv[1]*(s->fwpos[3]-s->prev_fwpos[3])*s->rate + s->force[3];
s->force[4] = - s->Kv[1]*(s->fwpos[4]-s->prev_fwpos[4])*s->rate + s->force[4];

// Yaw force is modeled as proportional to the yaw angle.
s->force[5] = - s->Kv[1]*(s->fwpos[5]-s->prev_fwpos[5])*s->rate
 -s->spring_limit[1]*(s->fwpos[5]);

// Distance of the handle from the center of the flotor's hemisphere
// in terms of real world coordinates
float real_dist = sqrt(pow(s->fwpos[0],2)+pow(s->fwpos[1],2)+pow(s->fwpos[2],2));

// If the handle is beyond the boundary, then simulate a hard wall
if(real_dist > s->spring_limit[0])
  {
    // Kp[0]: Proportional constant for translation,
    // Kv[0]: Dampening constant for translation
    s->force[0] = -s->Kp[0]*(real_dist-s->spring_limit[0])*s->fwpos[0]/real_dist
     -s->Kv[0]*(s->fwpos[0]-s->prev_fwpos[0])*s->rate+s->force[0];

    s->force[1] = -s->Kp[0]*(real_dist-s->spring_limit[0])*s->fwpos[1]/real_dist
       -s->Kv[0]*(s->fwpos[1]-s->prev_fwpos[1])*s->rate+s->force[1];

    s->force[2] = -s->Kp[0]*(real_dist-s->spring_limit[0])*s->fwpos[2]/real_dist
     -s->Kv[0]*(s->fwpos[2]-s->prev_fwpos[2])*s->rate+s->force[2];

  }

  // The proportional constant to restrict angular motion depends on the gamma
  // of the spring. It is empirically set as 2.5 times gamma. A value of 30 has
  // been identified as the minimum for this constant.

float propConstant = s->Kp[1]*gamma;

if(propConstant < 30)
  {
    propConstant = 30;
  }
```

```
  if(fabs(s->fwpos[3])>angle_limit)
    {
      if(s->fwpos[3]>0.0)
{
  s->force[3] =  -(s->fwpos[3]-angle_limit)*propConstant
    -(s->fwpos[3]-s->prev_fwpos[3])*s->Kv[1]*s->rate+s->force[3];
  }
      else
{
  s->force[3] =  -(s->fwpos[3]+angle_limit)*propConstant
    -(s->fwpos[3]-s->prev_fwpos[3])*s->Kv[1]*s->rate+s->force[3];
}
    }

  if(fabs(s->fwpos[4])>angle_limit)
    {
      if(s->fwpos[4]>0.0)
{
  s->force[4] =  -(s->fwpos[4]-angle_limit)*propConstant
                 -(s->fwpos[4]-s->prev_fwpos[4])*s->Kv[1]*s->rate+s->force[4];
}
      else
{
  s->force[4] =  -(s->fwpos[4]+angle_limit)*propConstant
    -(s->fwpos[4]-s->prev_fwpos[4])*s->Kv[1]*s->rate+s->force[4];
}
    }

  if(fabs(s->fwpos[5])>yaw_limit)
    {
      if(s->fwpos[5]>0.0)
{
  s->force[5] =  -(s->fwpos[5]-yaw_limit)*propConstant
    -(s->fwpos[5]-s->prev_fwpos[5])*s->Kv[1]*s->rate+s->force[5];
}
      else
{
  s->force[5] =  -(s->fwpos[5]+yaw_limit)*propConstant
    -(s->fwpos[5]-s->prev_fwpos[5])*s->Kv[1]*s->rate+s->force[5];
}
    }

  // If force values are greater than these specified values, emit a beep
  if(fabs(s->force[0]) > 10 || fabs(s->force[1]) > 10 || fabs(s->force[2]) > 50)
    {
      printf("%c\n",7);
    }
}
```

B Source Code for Visual Rendering of Spring

The code presented here is the file vvaradha@dijeridu:src/teleop_wdir/springGraph.C.

```
/* solid cube inside wireframe cube */
/*
> CC -n32 -c cubes.C
> CC -n32 -o cubes cubes.o -L/usr/lib32/ -lInventor -lInventorXt -lm
*/

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/Xt/SoXtComponent.h>
#include <Inventor/engines/SoElapsedTime.h>
#include <Inventor/manips/SoTransformManip.h>
#include <Xm/Form.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoCylinder.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoDrawStyle.h>
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/sensors/SoIdleSensor.h>
#include <Inventor/nodes/SoScale.h>
#include <Inventor/nodes/SoTranslation.h>
#include <Inventor/nodes/SoMaterialBinding.h>
#include <Inventor/nodes/SoNormal.h>
#include <Inventor/nodes/SoNormalBinding.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoLineSet.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoTexture2.h>
#include <Inventor/nodes/SoCallback.h>
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/nodes/SoPointSet.h>
#include <Inventor/nodes/SoShapeHints.h>
#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoRotation.h>
#include <Inventor/events/SoEvent.h>
#include <Inventor/events/SoKeyboardEvent.h>
#include <Inventor/events/SoMouseButtonEvent.h>
#include <Inventor/events/SoMotion3Event.h>
#include <Inventor/sensors/SoTimerSensor.h>
#include <Inventor/events/SoLocation2Event.h>
#include <Inventor/manips/SoHandleBoxManip.h>
#include <Inventor/manips/SoTransformBoxManip.h>
#include <Inventor/manips/SoCenterballManip.h>
#include <Inventor/actions/SoBoxHighlightRenderAction.h>
#include <Inventor/draggers/SoDragPointDragger.h>
#include <Inventor/draggers/SoTranslate2Dragger.h>
#include <Inventor/nodes/SoNurbsCurve.h>
#include <Inventor/nodes/SoNurbsSurface.h>
#include <Inventor/nodes/SoComplexity.h>
```

```
#include <Inventor/SbColor.h>
#include <Inventor/SoPath.h>
#include <Inventor/SbLinear.h>
#include <Inventor/SbString.h>
#include <Inventor/SoPickedPoint.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/engines/SoCompose.h>
#include <Inventor/nodes/SoProfileCoordinate3.h>
#include <Inventor/nodes/SoNurbsProfile.h>
#include <Inventor/nodes/SoLinearProfile.h>
#include <Inventor/nodes/SoFont.h>
#include <Inventor/nodes/SoProfileCoordinate2.h>
#include <Inventor/nodes/SoText3.h>
#include <Inventor/nodes/SoText2.h>
#include <Inventor/nodes/SoQuadMesh.h>
#include <Inventor/nodes/SoSpotLight.h>
#include <FL/Fl.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Window.H>

/* added for socket ********/
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <stdio.h>
#include <stdlib.h>
#include "cubes.h"
#include "springGraph.h"
#include "springJNDExp_cb.h"
#include "texture_callbacks.h"
#include "callbacks.h"
#include "comm_callbacks.h"
#include "teleop_callbacks.h"
#include "puma_callbacks.h"
#include "monitor_callbacks.h"
#include "expDOF_callbacks.h"
#include "exp_callbacks.h"

/*******************************************/
// Externed globals

SoTransform *springTrans; // spring motion transform.
                          // This stores current location
                          // of the flotor.

#define PI 3.14159265

int springNumber;
int springPointSetNumber;
int coilNumber;
int coilNurbsCurveNumber;
int cylNumber;
int meshNumber;

float pitch;

/*****************************************/
```

```
/*
Function: update_spring
Output: void
Input: SbVec3f* translation of haptic flotor and
       SbVec3f* rotation of haptic flotor
Called from: cubes.C in graphics_callback()
Description: Uses the current location of the haptic
flotor to update the shape of the spring.
*/
void update_spring(SbVec3f translation, SbVec3f rotation, SbVec3f force, SbVec3f torque)
{
  float P;           // Vertical reaction force
  float q;           // Buckling factor
  float Hz;          // Horizontal reaction force along X axis
  float Hx;          // Horizontal reaction force along Z axis
  float M1z;         // Resisting moments
  float M1x;
  float M2z;
  float M2x;

  float siZu;        // Angle made by upperbar ABOUT Z-axis, measured
     // from X axis - Roll
  float siXu;        // Angle made by upperbar ABOUT  X-axis, measured
     // from Z axis - Pitch
  float siYu;        // Angle made by upperbar ABOUT Y-axis - Yaw

  float si1X;        // Upper spring centerline ends
  float si1Z;
  float si2X;        // Lower spring centerline ends
  float si2Z;

  float L;           // Compressed length of spring
  float compressY;   // Compression along Y-axis
  float shearX;      // Shear along X-axis
  float shearZ;      // Shear along Z-axis

  int coilCount;     // Counter for number of divisions in spring coil

  float x = 0;       // Temporary coordinate values for each division
  float y = 0;      // in the spring coil
  float z = 0;

  // Single character names are defined here for ease of use in formulaes
  float d = wireDiameter;
  float D = coilDiameter;
  float G = shearModulus;
  float E = youngModulus;

  float Lvertical; // Compressed length along the vertical axis

  float newx;    // Coordinates after transformation of axes
  float newy;
  float newz;
  float intery;

  float barY;   // Coordinates of upper end plate
  float barX;
  float barZ;

  float forceX; // Force values as calculated by the control
// loop used to generate warnings when the values are
// too high
```

```
float forceZ;
float forceY;

float gamma;  // Rigidity constant wrt compression
float alpha;  // Rigidity constant wrt moment
float beta;   // Rigidity constant wrt shear

// Calculate rigidity constants based on changes in physical
// parameters or rigidity constants
switch(springChange)
  {
  case 0: // Change in gamma
   // beta and alpha are directly proportional to gamma,
   // assuming D, E and G are constant
    gamma = spring_gamma;
    beta = gamma*E/G;
    alpha = gamma*pow(D,2)/(4*G/E+2);
    set_alpha(alpha);
    set_beta(beta);
    break;
  case 1: // Change in alpha
    alpha = spring_alpha;
    beta = alpha*(4+2*E/G)/pow(D,2);
    gamma = alpha*(4*G/E+2)/pow(D,2);
    set_beta(beta);
    set_gamma(gamma);
    break;
  case 2: // Change in beta
    beta = spring_beta;
    alpha = beta*pow(D,2)/(4+2*E/G);
    gamma = beta*G/E;
    set_alpha(alpha);
    set_gamma(gamma);
    break;
  case 3: // Change in physical parameters
    alpha = L0*pow(d,4)*E/(32*n*D*(1+E/(2*G)));
    beta = L0*pow(d,4)*E/(8*n*pow(D,3));
    gamma = L0*pow(d,4)*G/(8*n*pow(D,3));
    set_alpha(alpha);
    set_beta(beta);
    set_gamma(gamma);
    break;
  }

if(reset_spring_flag)
  {
    reset_spring_scene();
    reset_spring_flag = false;
  }

// Identify how the upper end plate is moving and thereby determine
// the resulting shape of the spring centerline, coil and
// cylindrical helix
SoSeparator *upperBar = (SoSeparator *)SoNode::getByName("UpperBar");
SoTransform *upperBarTransform = (SoTransform *)SoNode::getByName("UpperBarTransform");
SoRotationXYZ *upperRotationX = (SoRotationXYZ *)SoNode::getByName("UpperRotationX");
SoRotationXYZ *upperRotationY = (SoRotationXYZ *)SoNode::getByName("UpperRotationY");
SoRotationXYZ *upperRotationZ = (SoRotationXYZ *)SoNode::getByName("UpperRotationZ");

// Define new coordinates of the spring centerline
SoCoordinate3 *newSpringCoords = new SoCoordinate3;
SoPointSet *newSpringPointSet = new SoPointSet;
```

```
  // Define new coordinates of the spring coil
  SoCoordinate3 *newCoilCoords = new SoCoordinate3;
  SoNurbsCurve *newCoilCurve = new SoNurbsCurve;

  SoSeparator *springCenterline = (SoSeparator *)SoNode::getByName("SpringCenterline");
  SoPath *springPath = (SoPath *)SoNode::getByName("NewSpringCoords");

  if(springPath == NULL)
    {
      SoCoordinate3 *springCoords = (SoCoordinate3 *)SoNode::getByName("SpringCoords");
      springNumber = springCenterline->findChild(springCoords);

      SoPointSet *springPointSet = (SoPointSet *)SoNode::getByName("SpringPointSet");
      springPointSetNumber = springCenterline->findChild(springPointSet);
    }

  SoSeparator *springCoil = (SoSeparator *)SoNode::getByName("SpringCoil");
  SoPath *coilPath = (SoPath *)SoNode::getByName("NewCoilCoords");

  if(coilPath == NULL)
    {
      SoCoordinate3 *coilCoords = (SoCoordinate3 *)SoNode::getByName("CoilCoords");
      coilNumber = springCoil->findChild(coilCoords);

      SoNurbsCurve *coilCurve = (SoNurbsCurve *)SoNode::getByName("CoilCurve");
      coilNurbsCurveNumber = springCoil->findChild(coilCurve);
    }

  SoMaterial *coilMaterial = (SoMaterial *)SoNode::getByName("CoilMaterial");

  // Define new coordinates of the cylindrical helix
  SoCoordinate3 *newOuterLayer = new SoCoordinate3;
  SoQuadMesh *newQuadMesh = new SoQuadMesh;

  SoSeparator *cylLayer = (SoSeparator *)SoNode::getByName("CylLayer");
  SoPath *cylPath = (SoPath *)SoNode::getByName("NewOuterLayer");

  if(cylPath == NULL)
    {
      SoCoordinate3 *outerLayer = (SoCoordinate3 *)SoNode::getByName("OuterLayer");
      cylNumber = cylLayer->findChild(outerLayer);

      SoQuadMesh *myQuadMesh = (SoQuadMesh *)SoNode::getByName("MyQuadMesh");
      meshNumber =  cylLayer->findChild(myQuadMesh);
    }

/*
Get the translation (x,y,z) and rotation (roll, pitch, yaw) along the
graphical ZXY axes. The +Y axis is vertical, the +Z axis pointing AWAY
from you and +X axis pointing to your right as you look at this
screen. When the upper spring bar rotates such that its right end goes
down while its left goes up and vice versa the angle defined is roll -
rotation about Z axis. The angle it defines while rotating such that
it goes into the screen at an angle and out of the screen at an angle
it is pitch - rotation about X axis. Yaw is defineed by twist -
rotation about Y axis. Translation along the +Y axis is elongation and
along -Y axis is compression. Translation along X axis is shear and
translation along Z axis is shear in and out of the graphical 2D plane.
*/

/*
```

```
The above axes definitions are for the graphical world. In the haptic
world, the +Z axis is vertical, the +Y axis pointing AWAY
from you and +X axis pointing to your right as you look at this
screen. Hence the third value of SbVec3f translation of the form
(x,y,z) is compression, and the other two are shear. Roll, pitch and
yaw have to be suitably modified as well.
*/

/*
SbVec3f translation gives the translation of the flotor along XYZ
axes. If the flotor goes down, the third element gets a negative value
of its absolute vertical translation. When this is added to the
starting distance between the bars the current vertical distance is
obtained.
*/

/* rotation[i] are angles as defined by haptics world. siXu, siZu,
   siYu are angles defined by graphics world. The latter are opposite in
   sign to the theoretical theta.

*/

  compressY = translation[2]*L0/flotor_depth;   //-ve value if compression and
//+ve if elongation
  shearX = translation[0]*L0/flotor_depth;
  shearZ = translation[1]*L0/flotor_depth;

  siXu = rotation[0];     // Pitch
  siZu = -rotation[1];    // Roll
  siYu = rotation[2];     // Yaw

  forceX = force[0];
  forceZ = force[1];
  forceY = force[2];

// Euclidean distance between spring cenerline ends
  L = sqrt(pow((L0+compressY),2)+pow(shearX,2)+pow(shearZ,2));
  Lvertical = L0 + compressY;

  si2Z = atan(shearX/Lvertical);
  si1Z = si2Z - siZu;

  si2X = atan(shearZ/Lvertical);
  si1X = si2X - siXu;

  // If forces are too high, display a warning
  if(fabs(forceX) > 10 || fabs(forceZ) > 10 || fabs(forceY) > 50)
    {
      ui->excessForceWarn->show();
    }
  else
    {
      ui->excessForceWarn->hide();
    }

  // Compression
  if(L <= L0 & L > 0)
    {
      P = gamma*(L0-L)/L;
      q = sqrt((P/alpha)*(1+P/beta));
      Hx = P*(si1Z+si2Z)/(L*P/(tan(q*L/2)*alpha*q)-2);
      Hz = P*(si1X+si2X)/(L*P/(alpha*q*tan(q*L/2))-2);
```

```
      // Value of H as L->L0 and P->0
      if(L >= L0-1e-4)
        {
  Hx = -beta*(si1Z+si2Z)/(2*(1+beta*pow(L,2)/(12*alpha)));
  Hz = -beta*(si1X+si2X)/(2*(1+beta*pow(L,2)/(12*alpha)));
        }

      M1z = (alpha*q*(Hx/P+si1Z)*(L0/L0)-Hx*L/sin(q*L))/tan(q*L/2);
      M1x = (alpha*q*(Hz/P+si1X)*(L0/L0)-Hz*L/sin(q*L))/tan(q*L/2);

      M2z = M1z + Hx*L;

      M2x = M1x + Hz*L;

      coilCount = 0;

      newCoilCurve->knotVector.set1Value(1,0);

      pitch=L0/n;

      // Calculate the (x,y,z) values for each point on the spring coil
      for(float i = 0; i <= n*2*PI; i+=0.1)
{

  y = i*L/(n*2*PI);

  x = (M1z*(tan(q*L/2)*sin(q*y) + cos(q*y) - 1) +
   Hx*L*(-sin(q*y)/tan(q*L) + cos(q*y) + y/L - 1))/P;

  z = (M1x*(tan(q*L/2)*sin(q*y) + cos(q*y) - 1) +
   Hz*L*(-sin(q*y)/tan(q*L) + cos(q*y) + y/L - 1))/P;

  // Transform the axes by si2 since lower end plate is held fixed
  newx = x*cos(si2Z)+y*sin(si2Z);

  intery = y*cos(si2Z)-x*sin(si2Z);

  newz =-(z*cos(si2X)+intery*sin(si2X));

  newy = intery*cos(si2X)-z*sin(si2X);

  newCoilCoords->point.set1Value(coilCount, D*cos(i)/2 + newx, newy, newz + D*sin(i)/2);

  coilCount++;

  newCoilCurve->knotVector.set1Value(coilCount+1, coilCount);
  newSpringCoords->point.set1Value(coilCount, newx, newy, newz);

}

      newSpringPointSet->numPoints.setValue(coilCount);
      newCoilCurve->numControlPoints = coilCount;
      newCoilCurve->knotVector.set1Value(coilCount+2, coilCount+1);
    }

  // Elongation
  else if(L > L0 & L > 0)
    {
      P = gamma*(L0-L)/L;
      q = sqrt(-((P/alpha)*(1+P/beta)));
      Hx = P*(si1Z+si2Z)/(-L*P/(tanh(q*L/2)*alpha*q)-2);
```

```
      Hz = P*(si1X+si2X)/(-L*P/(alpha*q*tanh(q*L/2))-2);

      // Value of H as L->L0 and P->0
      if(L <= L0+1e-4)
{
   Hx = -beta*(si1Z+si2Z)/(2*(1+beta*pow(L,2)/(12*alpha)));
   Hz = -beta*(si1X+si2X)/(2*(1+beta*pow(L,2)/(12*alpha)));
}

      M1z = ((Hx/P+si1Z)*alpha*q*(L0/L0)+Hx*L/sinh(q*L))/tanh(q*L/2);
      M1x = ((Hz/P+si1X)*alpha*q*(L0/L0)+Hz*L/sinh(q*L))/tanh(q*L/2);

      M2z = M1z + Hx*L;
      M2x = M1x + Hz*L;

      coilCount = 0;

      newCoilCurve->knotVector.set1Value(1,0);

      // Calculate the (x,y,z) values for each point on the spring coil
      for(float i = 0; i <= n*2*PI; i+=0.1)
{
   y = i*L/(n*2*PI);

   x = (M1z*(-tanh(q*L/2)*sinh(q*y) + cosh(q*y) - 1) +
       Hx*L*(-sinh(q*y)/tanh(q*L) + cosh(q*y) + y/L - 1))/P;

   z = (M1x*(-tanh(q*L/2)*sinh(q*y) + cosh(q*y) - 1) +
       Hz*L*(-sinh(q*y)/tanh(q*L) + cosh(q*y) + y/L - 1))/P;


  // Transform the axes by si2 since lower end plate is held fixed
  newx = x*cos(si2Z)+y*sin(si2Z);

  intery = y*cos(si2Z)-x*sin(si2Z);

  newz =-(z*cos(si2X)+intery*sin(si2X));

  newy = intery*cos(si2X)-z*sin(si2X);

  newCoilCoords->point.set1Value(coilCount, D*cos(i)/2 + newx , newy, newz + D*sin(i)/2);
  coilCount++;

  newCoilCurve->knotVector.set1Value(coilCount+1, coilCount);
  newSpringCoords->point.set1Value(coilCount, newx, newy, newz);
}

      newSpringPointSet->numPoints.setValue(coilCount);
      newCoilCurve->numControlPoints = coilCount;
      newCoilCurve->knotVector.set1Value(coilCount+2, coilCount+1);
    }

  // Move the upper end plate to the new position
  barY = newy;
  barX = newx;
  barZ = newz;

  upperRotationX->angle = angle_scaler*siXu;
  upperRotationY->angle = angle_scaler*siYu;
  upperRotationZ->angle = angle_scaler*siZu;
  upperBarTransform->translation.setValue(barX, barY-L0/2, barZ);
```

```
int cylCount = 0;

SbVec3f* layer = new SbVec3f(0,0,-wireDiameter/2);

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,wireDiameter/(2*sqrt(2)),-wireDiameter/(2*sqrt(2)));

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,wireDiameter/2,0);

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,wireDiameter/(2*sqrt(2)),wireDiameter/(2*sqrt(2)));

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,0,wireDiameter/2);

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,-wireDiameter/(2*sqrt(2)),wireDiameter/(2*sqrt(2)));

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,-wireDiameter/2,0);

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
    newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,-wireDiameter/(2*sqrt(2)),-wireDiameter/(2*sqrt(2)));

for(int i = 0; i < newCoilCoords->point.getNum(); i++)
  {
```

```
      newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
      cylCount++;
    }

  layer = new SbVec3f(0,0,-wireDiameter/2);

  for(int i = 0; i < newCoilCoords->point.getNum(); i++)
    {
      newOuterLayer->point.set1Value(cylCount, newCoilCoords->point[i]+ *layer);
      cylCount++;
    }

  // Update the coordinates of the scene
  newQuadMesh->setName("NewQuadMesh");
  newQuadMesh->verticesPerRow = newCoilCoords->point.getNum();

  newQuadMesh->verticesPerColumn = 9;
  cylLayer->replaceChild(meshNumber, newQuadMesh);

  newSpringCoords->setName("NewSpringCoords");
  springCenterline->replaceChild(springNumber, newSpringCoords);
  newSpringPointSet->setName("NewSpringPointSet");
  springCenterline->replaceChild(springPointSetNumber, newSpringPointSet);

  newCoilCoords->setName("NewCoilCoords");
  springCoil->replaceChild(coilNumber, newCoilCoords);

  newCoilCurve->setName("NewCoilCurve");
  springCoil->replaceChild(coilNurbsCurveNumber, newCoilCurve);

  newOuterLayer->setName("NewOuterLayer");
  cylLayer->replaceChild(cylNumber, newOuterLayer);
}

// This function is called whenever the coil diameter or wireDiameter of
// the spring changes
void reset_spring_scene()
{
  float l = coilDiameter/2;
  float w = coilDiameter/2;
  float h = 0.5;

  float vertices[24][3] =
    {
      {-l,-h,-w}, {l,-h,-w}, {l,h,-w}, {-l,h,-w},
      {l,-h,-w}, {l,-h,w}, {l,h,w}, {l,h,-w},
      {l,-h,w}, {-l,-h,w}, {-l,h,w}, {l,h,w},
      {-l,-h,w}, {-l,h,w}, {-l,h,-w}, {-l,-h,-w},
      {-l,-h,-w}, {l,-h,-w}, {l,-h,w}, {-l,-h,w},
      {-l,h,-w}, {l,h,-w}, {l,h,w}, {-l,h,w}
    };

  SoCoordinate3 *newBarCoords = (SoCoordinate3*)SoNode::getByName("BarCoords");
  newBarCoords->point.setValues(0, 24, vertices);

  SoDrawStyle *newCoilDrawStyle = (SoDrawStyle*)SoNode::getByName("CoilDrawStyle");
  newCoilDrawStyle->lineWidth = wireDiameter;

  SoTransform *newLowerBarTransform = (SoTransform*)SoNode::getByName("LowerBarTransform");
  newLowerBarTransform->translation.setValue(0, -(L0/2), 0);

  SoTransform *newPlaneTransform = (SoTransform*)SoNode::getByName("PlaneTransform");
```

```
  newPlaneTransform->translation.setValue(0,-(L0/2)-h,0);

  SoTransform *newCoilTransform = (SoTransform*)SoNode::getByName("CoilTransform");
  newCoilTransform->translation.setValue(0,-(L0/2)+h,0);
}


// This function is called in cubes.C to setup the spring scene
SoSeparator* init_springGraphics()
{
  float numberOfTurns = n;

  springTrans = new SoTransform;
  rootSpring = new SoSeparator;
  rootSpring->ref();

  // Set up camera position
  SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
  SbVec3f cameraViewpoint;
  cameraViewpoint.setValue(0.0, 0.0,90);
  myCamera->position = cameraViewpoint;
  SbVec3f origin;
  origin.setValue(0.0, 0.0, 0.0);
  myCamera->pointAt(origin);
  rootSpring->addChild(myCamera);

  // Set up lighting
  SoLightModel *sceneLightModel = new SoLightModel;
  sceneLightModel->model = SoLightModel::PHONG;
  SoDirectionalLight *myFrontLight = new SoDirectionalLight;
  SoDirectionalLight *myRearLight = new SoDirectionalLight;
  SoSpotLight *mySpotLight = new SoSpotLight;
  myFrontLight->direction.setValue(1, 1, 1);
  myRearLight->direction.setValue(-1,-1,-1);
  mySpotLight->direction.setValue(1, 0, 0);
  mySpotLight->cutOffAngle.setValue(0.5);
  mySpotLight->dropOffRate.setValue(0.7);
  mySpotLight->on.setValue(TRUE);

  rootSpring->addChild(sceneLightModel);
  rootSpring->addChild(myFrontLight);
  rootSpring->addChild(myRearLight);

  // The objects in the scene are two cuboids for the spring
  // ends, a spring centerline, a spring coil, and a cylindircal helix.
  //  A general cuboid is first defined
  float l = coilDiameter/2; // Length of cuboid
  float w = coilDiameter/2; // Width
  float h = 0.5;            // Height
  float transDist = L0/2;

  // Define a plane that acts as ground
  SoSeparator *plane = new SoSeparator;
  SoMaterial *planeMaterial = new SoMaterial;
  planeMaterial->diffuseColor.setValue(0, 0, 0.95);
  SoCube *ground = new SoCube();
  ground->width.setValue(120);
  ground->height.setValue(50);
  ground->depth.setValue(1);

  SoTransform *planeTransform = new SoTransform;
  planeTransform->setName("PlaneTransform");
```

```
planeTransform->translation.setValue(0,-transDist-h,0);

SoRotationXYZ *planeRotation = new SoRotationXYZ;
planeRotation->axis = SoRotationXYZ::X;
planeRotation->angle = 1.57;

plane->addChild(planeMaterial);
plane->addChild(planeTransform);
plane->addChild(planeRotation);
plane->addChild(ground);
rootSpring->addChild(plane);

// Specify the vertices of the cuboid
static float vertices[24][3] =
  {
    {-l,-h,-w}, {l,-h,-w}, {l,h,-w}, {-l,h,-w},
    {l,-h,-w}, {l,-h,w}, {l,h,w}, {l,h,-w},
    {l,-h,w}, {-l,-h,w}, {-l,h,w}, {l,h,w},
    {-l,-h,w}, {-l,h,w}, {-l,h,-w}, {-l,-h,-w},
    {-l,-h,-w}, {l,-h,-w}, {l,-h,w}, {-l,-h,w},
    {-l,h,-w}, {l,h,-w}, {l,h,w}, {-l,h,w}
  };

static int numvertices[6] = {4, 4, 4, 4, 4, 4};

SoCoordinate3 *barCoords = new SoCoordinate3;
barCoords->setName("BarCoords");
barCoords->point.setValues(0, 24, vertices);

SoFaceSet *barFaceSet = new SoFaceSet;
barFaceSet->setName("BarFaceSet");
barFaceSet->numVertices.setValues(0,6, numvertices);

// Color for upper end plate
SoMaterial *upperMaterial = new SoMaterial;
upperMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);

// Color for lower end plate
SoMaterial *lowerBarMaterial = new SoMaterial;
lowerBarMaterial->diffuseColor.setValue(0.0, 0.0, 1.0);

// Translate and rotate the end plates so that they sit at the ends
// of the spring
SoTransform *upperBarTransform = new SoTransform;
upperBarTransform->setName("UpperBarTransform");
upperBarTransform->translation.setValue(0,transDist,0);

SoRotationXYZ *upperRotationX = new SoRotationXYZ;
upperRotationX->setName("UpperRotationX");
upperRotationX->axis = SoRotationXYZ::X;
upperRotationX->angle = 0.0;

SoRotationXYZ *upperRotationY = new SoRotationXYZ;
upperRotationY->setName("UpperRotationY");
upperRotationY->axis = SoRotationXYZ::Y;
upperRotationY->angle = 0.0;

SoRotationXYZ *upperRotationZ = new SoRotationXYZ;
upperRotationZ->setName("UpperRotationZ");
upperRotationZ->axis = SoRotationXYZ::Z;
upperRotationZ->angle = 0.0;
```

```
SoTransform *lowerBarTransform = new SoTransform;
lowerBarTransform->setName("LowerBarTransform");
lowerBarTransform->translation.setValue(0,-transDist,0);

SoSeparator *upperBar = new SoSeparator;
upperBar->setName("UpperBar");
upperBar->addChild(upperMaterial);
upperBar->addChild(upperBarTransform);
upperBar->addChild(upperRotationX);
upperBar->addChild(upperRotationY);
upperBar->addChild(upperRotationZ);
upperBar->addChild(barCoords);
upperBar->addChild(barFaceSet);

SoSeparator *lowerBar = new SoSeparator;
lowerBar->setName("LowerBar");
lowerBar->addChild(lowerBarMaterial);
lowerBar->addChild(lowerBarTransform);
lowerBar->addChild(barCoords);
lowerBar->addChild(barFaceSet);

// Build the spring centerline
SoMaterial *pointMaterial = new SoMaterial;
pointMaterial->diffuseColor.setValue(0.0, 1.0, 0.0);

SoDrawStyle *pointDrawStyle = new SoDrawStyle;
pointDrawStyle->style = SoDrawStyle::POINTS;
pointDrawStyle->pointSize = 2.0;

SoTransform *pointsTransform = new SoTransform;
pointsTransform->setName("PointsTransform");
pointsTransform->translation.setValue(0,-transDist+h,0);

// Build the spring coil
SoMaterial *coilMaterial = new SoMaterial;
coilMaterial->setName("CoilMaterial");
coilMaterial->diffuseColor.setValue(1, 0.2, 1);

SoDrawStyle *coilDrawStyle = new SoDrawStyle;
coilDrawStyle->setName("CoilDrawStyle");
coilDrawStyle->style = SoDrawStyle::FILLED;
coilDrawStyle->lineWidth = 3*wireDiameter;

SoTransform *coilTransform = new SoTransform;
coilTransform->setName("CoilTransform");
coilTransform->translation.setValue(0,-transDist+h,0);

// Specify the values of the coordinates on the spring
// centerline.
SoCoordinate3 *springCoords = new SoCoordinate3;
springCoords->setName("SpringCoords");

// Specify the values of the coordiantes on the spring coil.
SoCoordinate3 *coilCoords = new SoCoordinate3;
coilCoords->setName("CoilCoords");

// Define the NURBS curve including the control points
// and a complexity.
SoComplexity  *complexity = new SoComplexity;
complexity->value = 0.8;

SoNurbsCurve *coilCurve = new SoNurbsCurve;
```

```
coilCurve->setName("CoilCurve");
coilCurve->knotVector.set1Value(1,0);

SoNurbsSurface *coilSurface = new SoNurbsSurface;
coilSurface->setName("CoilSurface");
coilSurface->uKnotVector.set1Value(1,0);
coilSurface->vKnotVector.set1Value(1,0);

int coilCount = 0;

pitch=L0/(wireDiameter*numberOfTurns);

// Calculate the values of the spring centerline coordinates and use
// that to calculate the values of the spring coil
for(float i = 0; i <= wireDiameter*numberOfTurns*2*PI; i+=0.1)
  {
    springCoords->point.set1Value(coilCount, 0.0, pitch*i/(2*PI), 0.0);
    coilCoords->point.set1Value(coilCount, coilDiameter*cos(i)/2,
                                pitch*i/(2*PI), coilDiameter*sin(i)/2);

    coilCount++;
    coilCurve->knotVector.set1Value(coilCount+1, coilCount);
}

coilCurve->knotVector.set1Value(coilCount+2, coilCount+1);
coilCurve->knotVector.set1Value(coilCount+3, coilCount+2);
coilCurve->numControlPoints = coilCount;

// Define the cylindrical helix
SoCoordinate3* outerLayer = new SoCoordinate3;
outerLayer->setName("OuterLayer");

int cylCount = 0;

// The cylindrical helix is made up of connected circles defined by eight
// equiangular points. This can be thought of as eight helical
// curves. First, specify the eight curves
SbVec3f* layer = new SbVec3f(0,0,-wireDiameter/2);

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,wireDiameter/(2*sqrt(2)),-wireDiameter/(2*sqrt(2)));

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,wireDiameter/2,0);

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,wireDiameter/(2*sqrt(2)),wireDiameter/(2*sqrt(2)));
```

```
for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,0,wireDiameter/2);

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,-wireDiameter/(2*sqrt(2)),wireDiameter/(2*sqrt(2)));

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,-wireDiameter/2,0);

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

layer = new SbVec3f(0,-wireDiameter/(2*sqrt(2)),-wireDiameter/(2*sqrt(2)));

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

 layer = new SbVec3f(0,0,-wireDiameter/2);

for(int i = 0; i < coilCoords->point.getNum(); i++)
  {
    outerLayer->point.set1Value(cylCount, coilCoords->point[i]+ *layer);
    cylCount++;
  }

float curveKnots[7] = {0,1,2,3,4,5,6};

SoPointSet *springPointSet = new SoPointSet;
springPointSet->setName("SpringPointSet");
springPointSet->numPoints.setValue(coilCount);

SoSeparator *springCenterline = new SoSeparator;
springCenterline->setName("SpringCenterline");
springCenterline->addChild(pointMaterial);
springCenterline->addChild(pointDrawStyle);
springCenterline->addChild(pointsTransform);
springCenterline->addChild(springCoords);
springCenterline->addChild(springPointSet);

SoSeparator *springCoil = new SoSeparator;
```

```
   springCoil->setName("SpringCoil");
   springCoil->addChild(coilMaterial);
   springCoil->addChild(coilDrawStyle);
   springCoil->addChild(coilTransform);
   springCoil->addChild(complexity);
   springCoil->addChild(coilCoords);
   springCoil->addChild(coilCurve);

   SoSeparator *springSurface = new SoSeparator;
   springSurface->setName("SpringSurface");
   springSurface->addChild(coilMaterial);
   springSurface->addChild(coilTransform);
   springSurface->addChild(complexity);
   springSurface->addChild(coilCoords);
   springSurface->addChild(coilSurface);

   // Build picture by adding children
   //   rootSpring->addChild(springCenterline);
   //    rootSpring->addChild(springCoil);
   rootSpring->addChild(lowerBar);
   rootSpring->addChild(upperBar);

    // After building the spring centerline and coil, build the
    // cylindrical helix
    SoSeparator *cylLayer = new SoSeparator;
    cylLayer->setName("CylLayer");

    // Define the material
    SoMaterial *cylMaterial = new SoMaterial;
    cylMaterial->diffuseColor.setValue(.78, .57, .11);
    cylMaterial->shininess.setValue(100);
    cylLayer->addChild(cylMaterial);
    cylLayer->addChild(coilTransform);
    cylLayer->addChild(outerLayer);

    // Define the QuadMesh.
    SoQuadMesh *myQuadMesh = new SoQuadMesh;
    myQuadMesh->setName("MyQuadMesh");
    myQuadMesh->verticesPerRow = coilCoords->point.getNum();
    myQuadMesh->verticesPerColumn = 9;
    cylLayer->addChild(myQuadMesh);

    rootSpring->addChild(cylLayer);

    return rootSpring;

}
```