ArtiSynth Modeling Guide

John Lloyd and Antonio Sánchez
Last update: March, 2021

Contents

Preface

This guide describes how to create mechanical and biomechanical models in ArtiSynth using its Java API. Detailed information on how to use the ArtiSynth GUI for model visualization, navigation and simulation control is given in the ArtiSynth User Interface Guide. It is also possible to interface ArtiSynth with, or run it under, MATLAB. For information on this, see the guide Interfacing ArtiSynth to MATLAB.

Information on how to install and configure ArtiSynth is given in the installation guides for Windows, MacOS, and Linux.

It is assumed that the reader is familiar with basic Java programming, including variable assignment, control flow, exceptions, functions and methods, object construction, inheritance, and method overloading. Some familiarity with the basic I/O classes defined in `java.io.*`, including input and output streams and the specification of file paths using `File`, as well as the collection classes `ArrayList` and `LinkedList` defined in `java.util.*`, is also assumed.

How to read this guide

Section 1 offers a general overview of ArtiSynth's software design, and briefly describes the algorithms used for physical simulation (Section 1.2). The latter section may be skipped on first reading. A more comprehensive overview paper is available online.

The remainder of the manual gives details instructions on how to build various types of mechanical and biomechanical models. Sections 3 and 4 give detailed information about building general mechanical models, involving particles, springs, rigid bodies, joints, constraints, and contact. Section 5 describes how to add control panels, controllers, and input and output data streams to a simulation. Section 6 describes how to incorporate finite element models. The required mathematics is reviewed in Section A.

If time permits, the reader will profit from a top-to-bottom read. However, this may not always be necessary. Many of the sections contain detailed examples, all of which are available in the package `artisynth.demos.tutorial` and which may be run from ArtiSynth using Models > All demos > tutorials. More experienced readers may wish to find an appropriate example and then work backwards into the text and preceding sections for any needed explanatory detail.

Chapter 1

ArtiSynth Overview

ArtiSynth is an open-source, Java-based system for creating and simulating mechanical and biomechanical models, with specific capabilities for the combined simulation of rigid and deformable bodies, together with contact and constraints. It is presently directed at application domains in biomechanics, medicine, physiology, and dentistry, but it can also be applied to other areas such as traditional mechanical simulation, ergonomic design, and graphical and visual effects.

1.1 System structure

An ArtiSynth model is composed of a hierarchy of models and model components which are implemented by various Java classes. These may include sub-models (including finite element models), particles, rigid bodies, springs, connectors, and constraints. The component hierarchy may be in turn connected to various *agent* components, such as control panels, controllers and monitors, and input and output data streams (i.e., *probes*), which have the ability to control and record the simulation as it advances in time. Agents are presented in more detail in Section 5.

The models and agents are collected together within a top-level component known as a *root model*. Simulation proceeds under the control of a *scheduler*, which advances the models through time using a physics simulator. A rich graphical user interface (GUI) allows users to view and edit the model hierarchy, modify component properties, and edit and temporally arrange the input and output probes using a *timeline* display.

1.1.1 Model components

Every ArtiSynth component is an instance of ModelComponent. When connected to the hierarchy, it is assigned a unique number relative to its parent; the parent and number can be obtained using the methods getParent() and getNumber(), respectively. Components may also be assigned a name (using setName()) which is then returned using getName().

A sub-interface of `ModelComponent` includes CompositeComponent, which contains child components. A ComponentList is a `CompositeComponent` which simply contains a list of other components (such as particles, rigid bodies, sub-models, etc.).

Components which contain state information (such as position and velocity) should extend HasState, which provides the methods getState() and setState() for saving and restoring state.

A Model is a sub-interface of `CompositeComponent` and `HasState` that contains the notion of advancing through time and which implements this with the methods `initialize(t0)` and `advance(t0, t1, flags)`, as discussed further in Section 1.1.4. The most common instance of `Model` used in ArtiSynth is `MechModel` (Section 1.1.5), which is the top-level container for a mechanical or biomechanical model.

1.1.2 The RootModel

The top-level component in the hierarchy is the *root model*, which is a subclass of RootModel and which contains a list of models along with lists of agents used to control and interact with these models. The component lists in `RootModel` include:

`models`	top-level models of the component hierarchy
`inputProbes`	input data streams for controlling the simulation
`controllers`	functions for controlling the simulation
`monitors`	functions for observing the simulation
`outputProbes`	output data streams for observing the simulation

Each agent may be associated with a specific top-level model.

1.1.3 Component path names

The names and/or numbers of a component and its ancestors can be used to form a component path name. This path has a construction analogous to Unix file path names, with the '/' character acting as a separator. Absolute paths start with '/', which indicates the root model. Relative paths omit the leading '/' and can begin lower down in the hierarchy. A typical path name might be

```
/models/JawHyoidModel/axialSprings/lad
```

For nameless components in the path, their numbers can be used instead. Numbers can also be used for components that have names. Hence the path above could also be represented using only numbers, as in

```
/0/0/1/5
```

although this would most likely appear only in machine-generated output.

1.1.4 Model advancement

ArtiSynth simulation proceeds by advancing all of the root model's top-level models through a sequence of time steps. Every time step is achieved by calling each model's advance() method:

```
public StepAdjustment advance (double t0, double t1) {
   ... perform simulation ...
}
```

This method advances the model from time `t0` to time `t1`, performing whatever physical simulation is required (see Section 1.2). The method may optionally return a `StepAdjustment` indicating that the step size (`t1 - t0`) was too large and that the advance should be redone with a smaller step size.

The root model has it's own advance(), which in turn calls the advance method for all of the top-level models, in sequence. The advance of each model is surrounded by the application of whatever agents are associated with that model. This is done by calling the agent's `apply()` method:

```
model.preadvance (t0, t1);
for (each input probe p) {
   p.apply (t1);
}
for (each controller c) {
   c.apply (t0, t1);
}
model.advance (t0, t1);
for (each monitor m) {
   m.apply (t0, t1);
}
for (each output probe p) {
   p.apply (t1);
}
```

Agents not associated with a specific model are applied before (or after) the advance of all other models.

More precise details about model advancement are given in the ArtiSynth Reference Manual.

1.1.5 MechModel

Most ArtiSynth applications contain a single top-level model which is an instance of MechModel. This is a CompositeComponent that may (recursively) contain an arbitrary number of mechanical components, including finite element models, other `MechModel`s, particles, rigid bodies, constraints, attachments, and various force effectors. The `MechModel advance()` method invokes a physics simulator that advances these components forward in time (Section 1.2).

For convenience each `MechModel` contains a number of predefined containers for different component types, including:

`particles`	3 DOF particles
`points`	other 3 DOF points
`rigidBodies`	6 DOF rigid bodies
`frames`	other 6 DOF frames
`axialSprings`	point-to-point springs
`connectors`	joint-type connectors between bodies
`constrainers`	general constraints
`forceEffectors`	general force-effectors
`attachments`	attachments between dynamic components
`renderables`	renderable components (for visualization only)

Each of these is a child component of `MechModel` and is implemented as a ComponentList. Special methods are provided for adding and removing items from them. However, applications are not required to use these containers, and may instead create any component containment structure that is appropriate. If not used, the containers will simply remain empty.

1.2 Physics simulation

Only a brief summary of ArtiSynth physics simulation is described here. Full details are given in [8] and in the related overview paper.

For purposes of physics simulation, the components of a `MechModel` are grouped as follows:

Dynamic components
: Components, such as a particles and rigid bodies, that contain position and velocity state, as well as mass. All dynamic components are instances of the Java interface DynamicComponent.

Force effectors
: Components, such as springs or finite elements, that exert forces between dynamic components. All force effectors are instances of the Java interface ForceEffector.

Constrainers
: Components that enforce constraints between dynamic components. All constrainers are instances of the Java interface Constrainer.

Attachments
: Attachments between dynamic components. While technically these are constraints, they are implemented using a different approach. All attachment components are instances of DynamicAttachment.

The positions, velocities, and forces associated with all the dynamic components are denoted by the composite vectors $\mathbf{q}$, $\mathbf{u}$, and $\mathbf{f}$. In addition, the composite mass matrix is given by $\mathbf{M}$. Newton's second law then gives

$$\mathbf{f} = \frac{d\mathbf{M}\mathbf{u}}{dt} = \mathbf{M}\dot{\mathbf{u}} + \dot{\mathbf{M}}\mathbf{u}, \tag{1.1}$$

where the $\dot{\mathbf{M}}\mathbf{u}$ accounts for various "fictitious" forces.

Each integration step involves solving for the velocities $\mathbf{u}^{k+1}$ at time step $k+1$ given the velocities and forces at step k. One way to do this is to solve the expression

$$\mathbf{M}\mathbf{u}^{k+1} = \mathbf{M}\mathbf{u}^{k} + h\bar{\mathbf{f}} \tag{1.2}$$

for $\mathbf{u}^{k+1}$, where h is the step size and $\bar{\mathbf{f}} \equiv \mathbf{f} - \dot{\mathbf{M}}\mathbf{u}$. Given the updated velocities $\mathbf{u}^{k+1}$, one can determine $\dot{\mathbf{q}}^{k+1}$ from

$$\dot{\mathbf{q}}^{k+1} = \mathbf{Q}\mathbf{u}^{k+1}, \tag{1.3}$$

where $\mathbf{Q}$ accounts for situations (like rigid bodies) where $\dot{\mathbf{q}} \neq \mathbf{u}$, and then solve for the updated positions using

$$\mathbf{q}^{k+1} = \mathbf{q}^k + h\dot{\mathbf{q}}^{k+1}. \tag{1.4}$$

(1.2) and (1.4) together comprise a simple *symplectic Euler* integrator.

In addition to forces, bilateral and unilateral constraints give rise to locally linear constraints on $\mathbf{u}$ of the form

$$\mathbf{G}(\mathbf{q})\mathbf{u} = 0, \qquad \mathbf{N}(\mathbf{q})\mathbf{u} \geq 0. \tag{1.5}$$

Bilateral constraints may include rigid body joints, FEM incompressibility, and point-surface constraints, while unilateral constraints include contact and joint limits. Constraints give rise to constraint forces (in the directions $\mathbf{G}(\mathbf{q})^T$ and $\mathbf{N}(\mathbf{q})^T$) which supplement the forces of (1.1) in order to enforce the constraint conditions. In addition, for unilateral constraints, we have a complementarity condition in which $\mathbf{N}\mathbf{u} > 0$ implies no constraint force, and a constraint force implies $\mathbf{N}\mathbf{u} = 0$. Any given constraint usually involves only a few dynamic components and so $\mathbf{G}$ and $\mathbf{N}$ are generally sparse.

Adding constraints to the velocity solve (1.2) leads to a mixed linear complementarity problem (MLCP) of the form

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^T & -\mathbf{N}^T \\ \mathbf{G} & 0 & 0 \\ \mathbf{N} & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}^{k+1} \\ \tilde{\lambda} \\ \tilde{\theta} \end{pmatrix} + \begin{pmatrix} -\mathbf{M}\mathbf{u}^k - h\hat{\mathbf{f}}^k \\ -\mathbf{g} \\ -\mathbf{n} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \mathbf{w} \end{pmatrix},$$
$$0 \leq \theta \perp \mathbf{w} \geq 0, \tag{1.6}$$

where $\mathbf{w}$ is a slack variable, $\tilde{\lambda}$ and $\tilde{\theta}$ give the force constraint impulses over the time step, and $\mathbf{g}$ and $\mathbf{n}$ are derivative terms defined by

$$\mathbf{g} \equiv -h\dot{\mathbf{G}}\mathbf{u}^k, \quad \mathbf{n} \equiv -h\dot{\mathbf{N}}\mathbf{u}^k, \tag{1.7}$$

to account for time variations in $\mathbf{G}$ and $\mathbf{N}$. In addition, $\hat{\mathbf{M}}$ and $\hat{\mathbf{f}}$ are $\mathbf{M}$ and $\bar{\mathbf{f}}$ augmented with stiffness and damping terms terms to accommodate implicit integration, which is often required for problems involving deformable bodies. The actual constraint forces λ and θ can be determined by dividing the impulses by the time step h:

$$\lambda = \tilde{\lambda}/h, \quad \theta = \tilde{\theta}/h. \tag{1.8}$$

We note here that ArtiSynth uses a *full coordinate* formulation, in which the position of each dynamic body is solved using full, or unconstrained, coordinates, with constraint relationships acting to restrict these coordinates. In contrast, some other simulation systems, including OpenSim [4], use *reduced* coordinates, in which the system dynamics are formulated using a smaller set of coordinates (such as joint angles) that implicitly take the system's constraints into account. Each methodology has its own advantages. Reduced formulations yield systems with fewer degrees of freedom and no constraint errors. On the other hand, full coordinates make it easier to combine and connect a wide range of components, including rigid bodies and FEM models.

Attachments between components can be implemented by constraining the velocities of the attached components using special constraints of the form

$$\mathbf{u}_j = -\mathbf{G}_{j\alpha}\mathbf{u}_\alpha \tag{1.9}$$

where $\mathbf{u}_j$ and $\mathbf{u}_\alpha$ denote the velocities of the attached and non-attached components. The constraint matrix $\mathbf{G}_{j\alpha}$ is sparse, with a non-zero block entry for each *master* component to which the attached component is connected. The simplest case involves attaching a point j to another point k, with the simple velocity relationship

$$\mathbf{u}_j = \mathbf{u}_k \tag{1.10}$$

That means that $\mathbf{G}_{j\alpha}$ has a single entry of $-\mathbf{I}$ (where $\mathbf{I}$ is the 3×3 identity matrix) in the k-th block column. Another common case involves connecting a point j to a rigid frame k. The velocity relationship for this is

$$\mathbf{u}_j = \mathbf{u}_k - \mathbf{l}_j \times \omega_k \tag{1.11}$$

where $\mathbf{u}_k$ and ω_k are the translational and rotational velocity of the frame and l_j is the location of the point relative to the frame's origin (as seen in world coordinates). The corresponding $\mathbf{G}_{j\alpha}$ contains a single 3×6 block entry of the form

$$\begin{pmatrix} \mathbf{I} & [l_j] \end{pmatrix} \tag{1.12}$$

in the $k-th$ block column, where

$$[l] \equiv \begin{pmatrix} 0 & -l_z & l_y \\ l_z & 0 & -l_x \\ -l_y & l_x & 0 \end{pmatrix} \tag{1.13}$$

is a skew-symmetric *cross product matrix*. The attachment constraints $\mathbf{G}_{j\alpha}$ could be added directly to (1.6), but their special form allows us to explicitly solve for $\mathbf{u}_j$, and hence reduce the size of (1.6), by factoring out the attached velocities before solution.

The MLCP (1.6) corresponds to a single step integrator. However, higher order integrators, such as Newmark methods, usually give rise to MLCPs with an equivalent form. Most ArtiSynth integrators use some variation of (1.6) to determine the system velocity at each time step.

To set up (1.6), the `MechModel` component hierarchy is traversed and the methods of the different component types are queried for the required values. Dynamic components (type `DynamicComponent`) provide $\mathbf{q}$, $\mathbf{u}$, and $\mathbf{M}$; force effectors (`ForceEffector`) determine $\hat{\mathbf{f}}$ and the stiffness/damping augmentation used to produce $\hat{\mathbf{M}}$; constrainers (`Constrainer`) supply $\mathbf{G}$, $\mathbf{N}$, $\mathbf{g}$ and $\mathbf{n}$, and attachments (`DynamicAttachment`) provide the information needed to factor out attached velocities.

1.3 Basic packages

The core code of the ArtiSynth project is divided into three main packages, each with a number of sub-packages.

1.3.1 maspack

The packages under `maspack` contain general computational utilities that are independent of ArtiSynth and could be used in a variety of other contexts. The main packages are:

```
maspack.util               // general utilities
maspack.matrix             // matrix and linear algebra
maspack.graph              // graph algorithms
maspack.fileutil           // remote file access
maspack.properties         // property implementation
maspack.spatialmotion      // 3D spatial motion and dynamics
maspack.solvers            // LCP solvers and linear solver interfaces
maspack.render             // viewer and rendering classes
maspack.geometry           // 3D geometry and meshes
maspack.collision          // collision detection
maspack.widgets            // Java swing widgets for maspack data types
maspack.apps               // stand-alone programs based only on maspack
```

1.3.2 artisynth.core

The packages under `artisynth.core` contain the core code for ArtiSynth model components and its GUI infrastructure.

```
artisynth.core.util        // general ArtiSynth utilities
artisynth.core.modelbase   // base classes for model components
artisynth.core.materials   // materials for springs and finite elements
artisynth.core.mechmodels  // basic mechanical models
artisynth.core.femmodels   // finite element models
artisynth.core.probes      // input and output probes
artisynth.core.workspace   // RootModel and associated components
artisynth.core.driver      // start ArtiSynth and drive the simulation
artisynth.core.gui         // graphical interface
artisynth.core.inverse     // inverse tracking controller
artisynth.core.moviemaker  // used for making movies
artisynth.core.renderables // components that are strictly visual
artisynth.core.opensim     // OpenSim model parser (under development)
artisynth.core.mfreemodels // mesh free models (experimental, not supported)
```

1.3.3 artisynth.demos

These packages contain demonstration models that illustrate ArtiSynth's modeling capabilities:

```
artisynth.demos.mech       // mechanical model demos
artisynth.demos.fem        // demos involving finite elements
artisynth.demos.inverse    // demos involving inverse control
artisynth.demos.tutorial   // demos in this manual
```

1.4 Properties

ArtiSynth components expose *properties*, which provide a uniform interface for accessing their internal parameters and state. Properties vary from component to component; those for `RigidBody` include `position`, `orientation`, `mass`, and `density`, while those for `AxialSpring` include `restLength` and `material`. Properties are particularly useful for automatically creating control panels and probes, as described in Section 5. They are also used for automating component serialization.

Properties are described only briefly in this section; more detailed descriptions are available in the Maspack Reference Manual and the overview paper.

The set of properties defined for a component is fixed for that component's class; while property values may vary between component instances, their definitions are class-specific. Properties are exported by a class through code contained in the class definition, as described in Section 5.2.

1.4.1 Querying and setting property values

Each property has a unique name that can be used to access its value interactively in the GUI. This can be done either by using a custom control panel (Section 5.1) or by selecting the component and choosing Edit properties ... from the right-click context menu).

Properties can also be accessed in code using their set/get accessor methods. Unless otherwise specified, the names for these are formed by simply prepending `set` or `get` to the property's name. More specifically, a property with the name foo and a value type of `Bar` will usually have accessor signatures of

```
Bar getFoo()

void setFoo (Bar value)
```

1.4.2 Property handles and paths

A property's name can also be used to obtain a *property handle* through which its value may be queried or set generically. Property handles are implemented by the class Property and are returned by the component's getProperty() method. `getProperty()` takes a property's name and returns the corresponding handle. For example, components of type `Muscle` have a property `excitation`, for which a handle may be obtained using a code fragment such as

```
Muscle muscle;
...
Property prop = muscle.getProperty ("excitation");
```

Property handles can also be obtained for sub-components, using a *property path* that consists of a path to the sub-component followed by a colon '`:`' and the property name. For example, to obtain the `excitation` property for a sub-component located by `axialSprings/lad` relative to a `MechModel`, one could use a call of the form

```
MechModel mech;
...
Property prop = mech.getProperty ("axialSprings/lad:excitation");
```

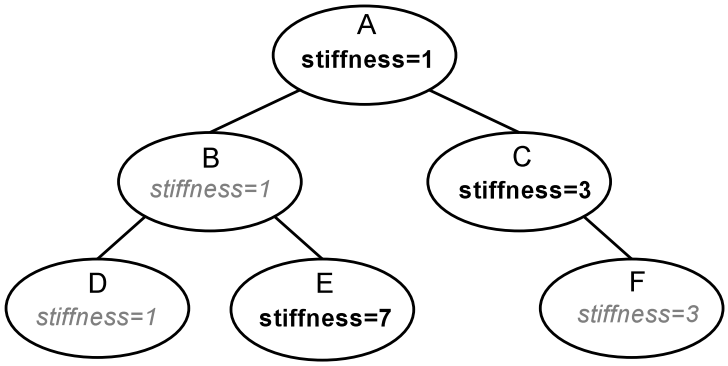


Figure 1.1: Inheritance of a property named *stiffness* among a component hierarchy. Explicit settings are in bold; inherited settings are in gray italic.

1.4.3 Composite and inheritable properties

Composite properties are possible, in which a property value is a composite object that in turn has sub-properties. A good example of this is the `RenderProps` class, which is associated with the property `renderProps` for renderable objects and which itself can have a number of sub-properties such as `visible`, `faceStyle`, `faceColor`, `lineStyle`, `lineColor`, etc.

Properties can be declared to be `inheritable`, so that their values can be inherited from the same properties hosted by ancestor components further up the component hierarchy. Inheritable properties require a more elaborate declaration and are associated with a *mode* which may be either `Explicit` or `Inherited`. If a property's mode is inherited, then its value is obtained from the closest ancestor exposing the same property whose mode is explicit. In Figure (1.1), the property *stiffness* is explicitly set in components A, C, and E, and inherited in B and D (which inherit from A) and F (which inherits from C).

1.5 Creating an application model

ArtiSynth applications are created by writing and compiling an *application model* that is a subclass of `RootModel`. This application-specific root model is then loaded and run by the ArtiSynth program.

The code for the application model should:

- Declare a no-args constructor
- Override the `RootModel` build() method to construct the application.

ArtiSynth can load a model either using the build method or by reading it from a file:

Build method

ArtiSynth creates an instance of the model using the no-args constructor, assigns it a name (which is either user-specified or the simple name of the class), and then calls the `build()` method to perform the actual construction.

Reading from a file

ArtiSynth creates an instance of the model using the no-args constructor, and then the model is named and constructed by reading the file.

The no-args constructor should perform whatever initialization is required in both cases, while the `build()` method takes the place of the file specification. Unless a model is originally created using a file specification (which is very tedious), the first time creation of a model will almost always entail using the `build()` method.

The general template for application model code looks like this:

```
package artisynth.models.experimental; // package where the model resides
import artisynth.core.workspace.RootModel;
... other imports ...

public class MyModel extends RootModel {

   // no-args constructor
   public MyModel() {
      ... basic initialization ...
   }

   // build method to do model construction
   public void build (String[] args) {
      ... code to build the model ....
   }
}
```

Here, the model itself is called `MyModel`, and is defined in the (hypothetical) package `artisynth.models.experimental` (placing models in the super package `artisynth.models` is common practice but not necessary).

> Note: The `build()` method was only introduced in ArtiSynth 3.1. Prior to that, application models were constructed using a constructor taking a `String` argument supplying the name of the model. This method of model construction still works but is deprecated.

1.5.1 Implementing the build() method

As mentioned above, the `build()` method is responsible for actual model construction. Many applications are built using a single top-level `MechModel`. Build methods for these may look like the following:

```
   public void build (String[] args) {
      MechModel mech = new MechModel("mech");
      addModel (mech);

      ... create and add components to the mech model ...
      ... create and add any needed agents to the root model ...

   }
```

First, a MechModel is created (with the name `"mech"` in this example, although any name, or no name, may be given) and added to the list of models in the root model using the `addModel()` method. Subsequent code then creates and adds the components required by the `MechModel`, as described in Sections 3, 4 and 6. The `build()` method also creates and adds to the root model any agents required by the application (controllers, probes, etc.), as described in Section 5.

When constructing a model, there is no fixed order in which components need to be added. For instance, in the above example, `addModel(mech)` could be called near the end of the `build()` method rather than at the beginning. The only restriction is that when a component is added to the hierarchy, all other components that it refers to should already have been added to the hierarchy. For instance, an axial spring (Section 3.1) refers to two points. When it is added to the hierarchy, those two points should already be present in the hierarchy.

The `build()` method supplies a `String` array as an argument, which can be used to transmit application arguments in a manner analogous to the `args` argument passed to static `main()` methods. Build arguments can be specified when a model is loaded directly from a class using Models > Load from class ..., or when the *startup model* is set to automatically load a model when ArtiSynth is first started (Settings > Startup model). Details are given in the "Loading, Simulating and Saving Models" section of the User Interface Guide.

Build arguments can also be listed directly on the ArtiSynth command line when specifying a model to load using the `-model <classname>` option. This is done by enclosing the desired arguments within square brackets `[ ]` immediately following the `-model` option. So, for example,

```
> artisynth -model projects.MyModel [ -size 50 ]
```

would cause the strings `"-size"` and `"50"` to be passed to the `build()` method of `MyModel`.

1.5.2 Making models visible to ArtiSynth

In order to load an application model into ArtiSynth, the classes associated with its implementation must be made visible to ArtiSynth. This usually involves adding the top-level class folder associated with the application code to the classpath used by ArtiSynth.

> The demonstration models referred to in this guide belong to the package `artisynth.demos.tutorial` and are already visible to ArtiSynth.

In most current ArtiSynth projects, classes are stored in a folder tree separate from the source code, with the top-level class folder named `classes`, located one level below the project root folder. A typical top-level class folder might be stored in a location like this:

```
/home/joeuser/artisynthProjects/classes
```

In the example shown in Section 1.5, the model was created in the package `artisynth.models.experimental`. Since Java classes are arranged in a folder structure that mirrors package names, with respect to the sample project folder shown above, the model class would be located in

```
/home/joeuser/artisynthProjects/classes/artisynth/models/experimental
```

At present there are three ways to make top-level class folders known to ArtiSynth:

Add projects to your Eclipse launch configuration

If you are using the Eclipse IDE, then you can add the project in which are developing your model code to the launch configuration that you use to run ArtiSynth. Other IDEs will presumably provide similar functionality.

Add the folders to the external classpath

You can explicitly add the class folders to ArtiSynth's external classpath. The easiest way to do this is to select "Settings > External classpath ..." from the Settings menu, which will open an external classpath editor which lists all the classpath entries in a large panel on the left. (When ArtiSynth is first installed, the external classpath has no entries, and so this panel will be blank.) Class folders can then by added via the "Add class folder" button, and the classpath is saved using the Save button.

Add the folders to your CLASSPATH environment variable

If you are running ArtiSynth from the command line, using the `artisynth` command (or `artisynth.bat` on Windows), then you can define a CLASSPATH environment variable in your environment and add the needed folders to this.

All of these methods are described in more detail in the "Installing External Models and Packages" section of the ArtiSynth Installation Guide (available for Linux, Windows, and MacOS).

1.5.3 Loading and running a model

If a model's classes are visible to ArtiSynth, then it may be loaded into ArtiSynth in several ways:

Loading from the Model menu

If the root model is contained in a package located under `artisynth.demos` or `artisynth.models`, then it will appear in the default model menu (Models in the main menu bar) under the submenu All demos or All models.

Loading by class path

A model may also be loaded by choosing "Load from class ..." from the Models menu and specifying its package name and then choosing its root model class. It is also possible to use the `-model <classname>` command line argument to have a model loaded directly into ArtiSynth when it starts up.

Loading from a file

If a model has been saved to a `.art` file, it may be loaded from that file by choosing File > Load model ....

These methods are described in detail in the section “Loading and Simulating Models” of the ArtiSynth User Interface Guide.

> The demonstration models referred to in this guide should already be present in the model menu and may be loaded from the submenu Models > All demos > tutorial.

Once a model is loaded, it can be simulated, or *run*. Simulation of the model can then be started, paused, single-stepped, or reset using the play controls (Figure 1.2) located at the upper right of the ArtiSynth window frame. Starting and stopping a simulation is done by clicking play/pause, while reset resets the simulation to time 0. The single-step button advances the simulation by one time step. The stop-all button will also stop the simulation, along with any Jython commands or scripts that are running.

Figure 1.2: The ArtiSynth play controls. From left to right: step size control, current simulation time, and the reset, skip-back, play/pause, single-step, skip-forward and stop-all buttons.

Comprehensive information on exploring and interacting with models is given in the ArtiSynth User Interface Guide.

Chapter 2

Supporting classes

ArtiSynth uses a large number of supporting classes, mostly defined in the super package `maspack`, for handling mathematical and geometric quantities. Those that are referred to in this manual are summarized in this section.

2.1 Vectors and matrices

Among the most basic classes are those used to implement vectors and matrices, defined in `maspack.matrix`. All vector classes implement the interface Vector and all matrix classes implement Matrix, which provide a number of standard methods for setting and accessing values and reading and writing from I/O streams.

General sized vectors and matrices are implemented by VectorNd and MatrixNd. These provide all the usual methods for linear algebra operations such as addition, scaling, and multiplication:

```
VectorNd v1 = new VectorNd (5);         // create a 5 element vector
VectorNd v2 = new VectorNd (5);
VectorNd vr = new VectorNd (5);
MatrixNd M = new MatrixNd (5, 5);       // create a 5 x 5 matrix

M.setIdentity();                        // M = I
M.scale (4);                            // M = 4*M

v1.set (new double[] {1, 2, 3, 4, 5}); // set values
v2.set (new double[] {0, 1, 0, 2, 0});
v1.add (v2);                            // v1 += v2
M.mul (vr, v1);                         // vr = M*v1

System.out.println ("result=" + vr.toString ("%8.3f"));
```

As illustrated in the above example, vectors and matrices both provide a `toString()` method that allows their elements to be formatted using a C-printf style format string. This is useful for providing concise and uniformly formatted output, particularly for diagnostics. The output from the above example is

```
result=   4.000   12.000   12.000   24.000   20.000
```

Detailed specifications for the format string are provided in the documentation for NumberFormat.set(String). If either no format string, or the string `"%g"`, is specified, `toString()` formats all numbers using the full-precision output provided by `Double.toString(value)`.

For computational efficiency, a number of fixed-size vectors and matrices are also provided. The most commonly used are those defined for three dimensions, including Vector3d and Matrix3d:

```
Vector3d v1 = new Vector3d (1, 2, 3);
Vector3d v2 = new Vector3d (3, 4, 5);
Vector3d vr = new Vector3d ();
Matrix3d M = new Matrix3d();
```

```
M.set (1, 2, 3,  4, 5, 6,  7, 8, 9);

M.mul (vr, v1);         // vr = M * v1
vr.scaledAdd (2, v2);   // vr += 2*v2;
vr.normalize();         // normalize vr
System.out.println ("result=" + vr.toString ("%8.3f"));
```

2.2 Rotations and transformations

`maspack.matrix` contains a number classes that implement rotation matrices, rigid transforms, and affine transforms.

Rotations (Section A.1) are commonly described using a RotationMatrix3d, which implements a rotation matrix and contains numerous methods for setting rotation values and transforming other quantities. Some of the more commonly used methods are:

```
RotationMatrix3d();          // create and set to the identity
RotationMatrix3d(u, angle); // create and set using an axis-angle

setAxisAngle (u, ang);       // set using an axis-angle
setRpy (roll, pitch, yaw);  // set using roll-pitch-yaw angles
setEuler (phi, theta, psi); // set using Euler angles
invert ();                   // invert this rotation
mul (R)                      // post multiply this rotation by R
mul (R1, R2);                // set this rotation to R1*R2
mul (vr, v1);                // vr = R*v1, where R is this rotation
```

Rotations can also be described by AxisAngle, which characterizes a rotation as a single rotation about a specific axis.

Rigid transforms (Section A.2) are used by ArtiSynth to describe a rigid body's pose, as well as its relative position and orientation with respect to other bodies and coordinate frames. They are implemented by RigidTransform3d, which exposes its rotational and translational components directly through the fields `R` (a `RotationMatrix3d`) and `p` (a `Vector3d`). Rotational and translational values can be set and accessed directly through these fields. In addition, `RigidTransform3d` provides numerous methods, some of the more commonly used of which include:

```
RigidTransform3d();          // create and set to the identity
RigidTransform3d(x, y, z);  // create and set translation to x, y, z

// create and set translation to x, y, z and rotation to roll-pitch-yaw
RigidTransform3d(x, y, z, roll, pitch, yaw);

invert ();                   // invert this transform
mul (T)                      // post multiply this transform by T
mul (T1, T2);                // set this transform to T1*T2
mulLeftInverse (T1, T2);     // set this transform to inv(T1)*T2
```

Affine transforms (Section A.3) are used by ArtiSynth to effect scaling and shearing transformations on components. They are implemented by AffineTransform3d.

Rigid transformations are actually a specialized form of affine transformation in which the basic transform matrix equals a rotation. `RigidTransform3d` and `AffineTransform3d` hence both derive from the same base class AffineTransform3dBase.

2.3 Points and Vectors

The rotations and transforms described above can be used to transform both vectors and points in space.

Vectors are most commonly implemented using Vector3d, while points can be implemented using the subclass Point3d. The only difference between `Vector3d` and `Point3d` is that the former ignores the translational component of rigid and

affine transforms; i.e., as described in Sections A.2 and A.3, a vector `v` has an implied homogeneous representation of

$$\mathbf{v}^* \equiv \begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix}, \tag{2.1}$$

while the representation for a point `p` is

$$\mathbf{p}^* \equiv \begin{pmatrix} \mathbf{p} \\ 1 \end{pmatrix}. \tag{2.2}$$

Both classes provide a number of methods for applying rotational and affine transforms. Those used for rotations are

```
void transform (R);             // this = R * this
void transform (R, v1);         // this = R * v1
void inverseTransform (R);      // this = inverse(R) * this
void inverseTransform (R, v1);  // this = inverse(R) * v1
```

where `R` is a rotation matrix and `v1` is a vector (or a point in the case of `Point3d`).

The methods for applying rigid or affine transforms include:

```
void transform (X);             // transforms this by X
void transform (X, v1);         // sets this to v1 transformed by X
void inverseTransform (X);      // transforms this by the inverse of X
void inverseTransform (X, v1);  // sets this to v1 transformed by inverse of X
```

where `X` is a rigid or affine transform. As described above, in the case of `Vector3d`, these methods ignore the translational part of the transform and apply only the matrix component (`R` for a `RigidTransform3d` and `A` for an `AffineTransform3d`). In particular, that means that for a `RigidTransform3d` given by `X` and a `Vector3d` given by `v`, the method calls

```
v.transform (X.R)
v.transform (X)
```

produce the same result.

2.4 Spatial vectors and inertias

The velocities, forces and inertias associated with 3D coordinate frames and rigid bodies are represented using the 6 DOF spatial quantities described in Sections A.5 and A.6. These are implemented by classes in the package `maspack.spatialmotion`.

Spatial velocities (or twists) are implemented by Twist, which exposes its translational and angular velocity components through the publicly accessible fields `v` and `w`, while spatial forces (or wrenches) are implemented by Wrench, which exposes its translational force and moment components through the publicly accessible fields `f` and `m`.

Both `Twist` and `Wrench` contain methods for algebraic operations such as addition and scaling. They also contain `transform()` methods for applying rotational and rigid transforms. The rotation methods simply transform each component by the supplied rotation matrix. The rigid transform methods, on the other hand, assume that the supplied argument represents a transform between two frames fixed within a rigid body, and transform the twist or wrench accordingly, using either (A.27) or (A.29).

The spatial inertia for a rigid body is implemented by SpatialInertia, which contains a number of methods for setting its value given various mass, center of mass, and inertia values, and querying the values of its components. It also contains methods for scaling and adding, transforming between coordinate systems, inversion, and multiplying by spatial vectors.

2.5 Meshes

ArtiSynth makes extensive use of 3D meshes, which are defined in `maspack.geometry`. They are used for a variety of purposes, including visualization, collision detection, and computing physical properties (such as inertia or stiffness variation within a finite element model).

A mesh is essentially a collection of vertices (i.e., points) that are topologically connected in some way. All meshes extend the abstract base class MeshBase, which supports the vertex definitions, while subclasses provide the topology.

Through `MeshBase`, all meshes provide methods for adding and accessing vertices. Some of these include:

```
int numVertices();                  // return the number of vertices
Vertex3d getVertex (int idx);       // return the idx-th vertex
void addVertex (Vertex3d vtx);      // add vertex vtx to the mesh
Vertex3d addVertex (Point3d p);     // create and return a vertex at position p
void removeVertex (Vertex3d vtx);   // remove vertex vtx for the mesh
ArrayList<Vertex3d> getVertices(); // return the list of vertices
```

Vertices are implemented by Vertex3d, which defines the position of the vertex (returned by the method getPosition()), and also contains support for topological connections. In addition, each vertex maintains an index, obtainable via getIndex(), that equals the index of its location within the mesh's vertex list. This makes it easy to set up parallel array structures for augmenting mesh vertex properties.

Mesh subclasses currently include:

PolygonalMesh

Implements a 2D surface mesh containing faces implemented using half-edges.

PolylineMesh

Implements a mesh consisting of connected line-segments (polylines).

PointMesh

Implements a point cloud with no topological connectivity.

PolygonalMesh is used quite extensively and provides a number of methods for implementing faces, including:

```
int numFaces();                 // return the number of faces
Face getFace (int idx);         // return the idx-th face
Face addFace (int[] vidxs);     // create and add a face using vertex indices
void removeFace (Face f);       // remove the face f
ArrayList<Face> getFaces();     // return the list of faces
```

The class Face implements a face as a counter-clockwise arrangement of vertices linked together by half-edges (class HalfEdge). `Face` also supplies a face's (outward facing) normal via getNormal().

Some mesh uses within ArtiSynth, such as collision detection, require a *triangular* mesh; i.e., one where all faces have three vertices. The method isTriangular() can be used to check for this. Meshes that are not triangular can be made triangular using triangulate().

2.5.1 Mesh creation

Meshes are most commonly created using either one of the factory methods supplied by MeshFactory, or by reading a definition from a file (Section 2.5.5). However, it is possible to create a mesh by direct construction. For example, the following code fragment creates a simple closed tetrahedral surface:

```
// a simple four-faced tetrahedral mesh
PolygonalMesh mesh = new PolygonalMesh();
mesh.addVertex (0, 0, 0);
mesh.addVertex (1, 0, 0);
mesh.addVertex (0, 1, 0);
mesh.addVertex (0, 0, 1);
mesh.addFace (new int[] { 0, 2, 1 });
mesh.addFace (new int[] { 0, 3, 2 });
mesh.addFace (new int[] { 0, 1, 3 });
mesh.addFace (new int[] { 1, 2, 3 });
```

Some of the more commonly used factory methods for creating polyhedral meshes include:

```
MeshFactory.createSphere (radius, nslices, nlevels);
MeshFactory.createIcosahedralSphere (radius, divisons);
MeshFactory.createBox (widthx, widthy, widthz);
MeshFactory.createCylinder (radius, height, nslices);
MeshFactory.createPrism (double[] xycoords, height);
MeshFactory.createTorus (rmajor, rminor, nmajor, nminor);
```

Each factory method creates a mesh in some standard coordinate frame. After creation, the mesh can be transformed using the transform(X) method, where `X` is either a rigid transform (RigidTransform3d) or a more general affine transform (AffineTransform3d). For example, to create a rotated box centered on $(5,6,7)$, one could do:

```
// create a box centered at the origin with widths 10, 20, 30:
PolygonalMesh box = MeshFactory.createBox (10, 20, 20);

// move the origin to 5, 6, 7 and rotate using roll-pitch-yaw
// angles 0, 0, 45 degrees:
box.transform (
   new RigidTransform3d (5, 6, 7,  0, 0, Math.toRadians(45)));
```

One can also scale a mesh using scale(s), where `s` is a single scale factor, or scale(sx,sy,sz), where `sx`, `sy`, and `sz` are separate scale factors for the x, y and z axes. This provides a useful way to create an ellipsoid:

```
 // start with a unit sphere with 12 slices and 6 levels ...
PolygonalMesh ellipsoid = MeshFactory.createSphere (1.0, 12, 6);

// and then turn it into an ellipsoid by scaling about the axes:
ellipsoid.scale (1.0, 2.0, 3.0);
```

MeshFactory can also be used to create new meshes by performing Boolean operations on existing ones:

```
MeshFactory.getIntersection (mesh1, mesh2);
MeshFactory.getUnion (mesh1, mesh2);
MeshFactory.getSubtraction (mesh1, mesh2);
```

2.5.2 Setting normals, colors, and textures

Meshes provide support for adding normal, color, and texture information, with the exact interpretation of these quantities depending upon the particular mesh subclass. Most commonly this information is used simply for rendering, but in some cases normal information might also be used for physical simulation.

> For polygonal meshes, the normal information described here is used only for smooth shading. When flat shading is requested, normals are determined directly from the faces themselves.

Normal information can be set and queried using the following methods:

```
setNormals (
   List<Vector3d> nrmls, int[] indices);  // set all normals and indices

ArrayList<Vector3d> getNormals();         // get all normals
int[] getNormalIndices();                 // get all normal indices
int numNormals();                         // return the number of normals
Vector3d getNormal (int idx);             // get the normal at index idx

setNormal (int idx, Vector3d nrml);       // set the normal at index idx
clearNormals();                           // clear all normals and indices
```

The method setNormals() takes two arguments: a set of normal vectors (`nrmls`), along with a set of index values (`indices`) that map these normals onto the vertices of each of the mesh's geometric features. Often, there will be one unique normal per vertex, in which case `nrmls` will have a size equal to the number of vertices, but this is not always the case, as described below. Features for the different mesh subclasses are: faces for `PolygonalMesh`, polylines for

`PolylineMesh`, and vertices for `PointMesh`. If `indices` is specified as `null`, then `normals` is assumed to have a size equal to the number of vertices, and an appropriate index set is created automatically using createVertexIndices() (described below). Otherwise, `indices` should have a size of equal to the number of features times the number of vertices per feature. For example, consider a `PolygonalMesh` consisting of two triangles formed from vertex indices (0, 1, 2) and (2, 1, 3), respectively. If normals are specified and there is one unique normal per vertex, then the normal indices are likely to be

```
[ 0 1 2  2 1 3 ]
```

As mentioned above, sometimes there may be *more* than one normal per vertex. This happens in cases when the same vertex uses different normals for different faces. In such situations, the size of the `nrmls` argument will exceed the number of vertices.

The method `setNormals()` makes internal copies of the specified normal and index information, and this information can be later read back using getNormals() and getNormalIndices(). The number of normals can be queried using numNormals(), and individual normals can be queried or set using getNormal(idx) and setNormal(idx,nrml). All normals and indices can be explicitly cleared using clearNormals().

Color and texture information can be set using analogous methods. For colors, we have

```
setColors (
   List<float[]> colors, int[] indices);  // set all colors and indices

ArrayList<float[]> getColors();           // get all colors
int[] getColorIndices();                  // get all color indices
int numColors();                          // return the number of colors
float[] getColor (int idx);               // get the color at index idx

setColor (int idx, float[] color);        // set the color at index idx
setColor (int idx, Color color);          // set the color at index idx
setColor (
   int idx, float r, float g, float b, float a); // set the color at index idx
clearColors();                            // clear all colors and indices
```

When specified as `float[]`, colors are given as RGB or RGBA values, in the range $[0, 1]$, with array lengths of 3 and 4, respectively. The colors returned by getColors() are always RGBA values.

With colors, there may often be *fewer* colors than the number of vertices. For instance, we may have only two colors, indexed by 0 and 1, and want to use these to alternately color the mesh faces. Using the two-triangle example above, the color indices might then look like this:

```
[ 0 0 0 1 1 1 ]
```

Finally, for texture coordinates, we have

```
setTextureCoords (
   List<Vector3d> coords, int[] indices); // set all texture coords and indices

ArrayList<Vector3d> getTextureCoords();   // get all texture coords
int[] getTextureIndices();                // get all texture indices
int numTextureCoords();                   // return the number of texture coords
Vector3d getTextureCoords (int idx);      // get texture coords at index idx

setTextureCoords (int idx, Vector3d coords);// set texture coords at index idx
clearTextureCoords();                     // clear all texture coords and indices
```

When specifying indices using setNormals, setColors, or setTextureCoords, it is common to use the same index set as that which associates vertices with features. For convenience, this index set can be created automatically using

```
int[] createVertexIndices();
```

Alternatively, we may sometimes want to create a index set that assigns the same attribute to each feature vertex. If there is one attribute per feature, the resulting index set is called a *feature index* set, and can be created using

```
int[] createFeatureIndices();
```

If we have a mesh with three triangles and one color per triangle, the resulting feature index set would be

```
[ 0 0 0 1 1 1 2 2 2 ]
```

> Note: when a mesh is modified by the *addition* of new features (such as faces for PolygonalMesh), all normal, color and texture information is cleared by default (with normal information being automatically recomputed on demand if automatic normal creation is enabled; see Section 2.5.3). When a mesh is modified by the *removal* of features, the index sets for normals, colors and textures are adjusted to account for the removal.
>
> For colors, it is possible to request that a mesh explicitly maintain colors for either its vertices or features (Section 2.5.4). When this is done, colors will persist when vertices or features are added or removed, with default colors being automatically created as necessary.

Once normals, colors, or textures have been set, one may want to know which of these attributes are associated with the vertices of a specific feature. To know this, it is necessary to find that feature's offset into the attribute's index set. This offset information can be found using the array returned by

```
int[] getFeatureIndexOffsets()
```

For example, the three normals associated with a triangle at index `ti` can be obtained using

```
int[] indexOffs = mesh.getFeatureIndexOffsets();
ArrayList<Vector3d> nrmls = mesh.getNormals();
// get the three normals associated with the triangle at index ti:
Vector3d n0 = nrmls.get (indexOffs[ti]);
Vector3d n1 = nrmls.get (indexOffs[ti]+1);
Vector3d n2 = nrmls.get (indexOffs[ti]+2);
```

Alternatively, one may use the convenience methods

```
Vector3d getFeatureNormal (int fidx, int k);
float[] getFeatureColor (int fidx, int k);
Vector3d getFeatureTextureCoords (int fidx, int k);
```

which return the attribute values for the *k*-th vertex of the feature indexed by `fidx`.

In general, the various `get` methods return references to internal storage information and so should **not** be modified. However, specific values within the lists returned by getNormals(), getColors(), or getTextureCoords() may be modified by the application. This may be necessary when attribute information changes as the simulation proceeds. Alternatively, one may use methods such as setNormal(idx,nrml) setColor(idx,color), or setTextureCoords(idx,coords).

Also, in some situations, particularly with colors and textures, it may be desirable to *not* have color or texture information defined for certain features. In such cases, the corresponding index information can be specified as -1, and the `getNormal()`, `getColor()` and `getTexture()` methods will return `null` for the features in question.

2.5.3 Automatic creation of normals and hard edges

For some mesh subclasses, if normals are not explicitly set, they are computed automatically whenever `getNormals()` or `getNormalIndices()` is called. Whether or not this is true for a particular mesh can be queried by the method

```
boolean hasAutoNormalCreation();
```

Setting normals explicitly, using a call to `setNormals(nrmls,indices)`, will overwrite any existing normal information, automatically computed or otherwise. The method

```
boolean hasExplicitNormals();
```

will return `true` if normals have been explicitly set, and `false` if they have been automatically computed or if there is currently no normal information. To explicitly remove normals from a mesh which has automatic normal generation, one may call `setNormals()` with the `nrmls` argument set to `null`.

More detailed control over how normals are automatically created may be available for specific mesh subclasses. For example, `PolygonalMesh` allows normals to be created with multiple normals per vertex, for vertices that are associated with either open or hard edges. This ability can be controlled using the methods

```
boolean getMultipleAutoNormals();
setMultipleAutoNormals (boolean enable);
```

Having multiple normals means that even with smooth shading, open or hard edges will still appear sharp. To make an edge hard within a `PolygonalMesh`, one may use the methods

```
boolean setHardEdge (Vertex3d v0, Vertex3d v1);
boolean setHardEdge (int vidx0, int vidx1);
boolean hasHardEdge (Vertex3d v0, Vertex3d v1);
boolean hasHardEdge (int vidx0, int vidx1);
int numHardEdges();
int clearHardEdges();
```

which control the hardness of edges between individual vertices, specified either directly or using their indices.

2.5.4 Vertex and feature coloring

The method setColors() makes it possible to assign any desired coloring scheme to a mesh. However, it does require that the user explicitly reset the color information whenever new features are added.

For convenience, an application can also request that a mesh explicitly maintain colors for either its vertices or features. These colors will then be maintained when vertices or features are added or removed, with default colors being automatically created as necessary.

Vertex-based coloring can be requested with the method

```
setVertexColoringEnabled();
```

This will create a separate (default) color for each of the mesh's vertices, and set the color indices to be equal to the vertex indices, which is equivalent to the call

```
setColors (colors, createVertexIndices());
```

where `colors` contains a default color for each vertex. However, once vertex coloring is enabled, the color and index sets will be updated whenever vertices or features are added or removed. Meanwhile, applications can query or set the colors for any vertex using `getColor(idx)`, or any of the various `setColor` methods. Whether or not vertex coloring is enabled can be queried using

```
getVertexColoringEnabled();
```

Once vertex coloring is established, the application will typically want to set the colors for all vertices, perhaps using a code fragment like this:

```
mesh.setVertexColoringEnabled();
for (int i=0; i<mesh.numVertices(); i++) {
   ... compute color for the vertex ...
   mesh.setColor (i, color);
}
```

Similarly, feature-based coloring can be requested using the method

```
setFeatureColoringEnabled();
```

This will create a separate (default) color for each of the mesh's features (faces for PolygonalMesh, polylines for PolylineMesh, etc.), and set the color indices to equal the feature index set, which is equivalent to the call

```
setColors (colors, createFeatureIndices());
```

where `colors` contains a default color for each feature. Applications can query or set the colors for any vertex using `getColor(idx)`, or any of the various `setColor` methods. Whether or not feature coloring is enabled can be queried using

```
getFeatureColoringEnabled();
```

2.5.5 Reading and writing mesh files

`PolygonalMesh`, `PolylineMesh`, and `PointMesh` all provide constructors that allow them to be created from a definition file, with the file format being inferred from the file name suffix:

```
PolygonalMesh (String fileName) throws IOException
PolygonalMesh (File file) throws IOException

PolylineMesh (String fileName) throws IOException
PolylineMesh (File file) throws IOException

PointMesh (String fileName) throws IOException
PointMesh (File file) throws IOException
```

Suffix	Format	`PolygonalMesh`	`PolylineMesh`	`PointMesh`
.obj	Alias Wavefront	X	X	X
.ply	Polygon file format	X		X
.stl	STereoLithography	X		
.gts	GNU triangulated surface	X		
.off	Object file format	X		
.vtk	VTK ascii format	X		
.vtp	VTK XML format	X	X	

Table 2.1: Mesh file formats which are supported for different mesh types

The currently supported file formats, and their applicability to the different mesh types, are given in Table 2.1. For example, a `PolygonalMesh` can be read from either an Alias Wavefront `.obj` file or an `.stl` file, as show in the following example:

```
PolygonalMesh mesh0 = null;
PolygonalMesh mesh1 = null;
try {
   mesh0 = new PolygonalMesh ("meshes/torus.obj");
}
catch (IOException e) {
   System.err.println ("Can't read mesh:");
   e.printStackTrace();
}
try {
   mesh1 = new PolygonalMesh ("meshes/cylinder.stl");
}
catch (IOException e) {
   System.err.println ("Can't read mesh:");
   e.printStackTrace();
}
```

The file-based mesh constructors may throw an I/O exception if an I/O error occurs or if the indicated format does not support the mesh type. This exception must either be caught, as in the example above, or thrown out of the calling routine.

In addition to file-based constructors, all mesh types implement read and write methods that allow a mesh to be read from or written to a file, with the file format again inferred from the file name suffix:

```
read (File file) throws IOException
write (File file) throws IOException

read (File file, boolean zeroIndexed) throws IOException
write (File file, String fmtStr, boolean zeroIndexed) throws IOException
```

For the latter methods, the argument `zeroIndexed` specifies zero-based vertex indexing in the case of Alias Wavefront `.obj` files, while `fmtStr` is a C-style format string specifying the precision and style with which the vertex coordinates should be written. (In the former methods, zero-based indexing is false and vertices are written using full precision.)

As an example, the following code fragment writes a mesh as an `.stl` file:

```
PolygonalMesh mesh;

... initialize ...

try {
   mesh.write (new File ("data/mymesh.obj"));
}
catch (IOException e) {
   System.err.println ("Can't write mesh:");
   e.printStackTrace();
}
```

Sometimes, more explicit control is needed when reading or writing a mesh from/to a given file format. The constructors and read/write methods described above make use of a specific set of reader and writer classes located in the package `maspack.geometry.io`. These can be used directly to provide more explicit read/write control. The readers and writers (if implemented) associated with the different formats are given in Table 2.2.

Suffix	Format	Reader class	Writer class
.obj	Alias Wavefront	`WavefrontReader`	`WavefrontWriter`
.ply	Polygon file format	`PlyReader`	`PlyWriter`
.stl	STereoLithography	`StlReader`	`StlWriter`
.gts	GNU triangulated surface	`GtsReader`	`GtsWriter`
.off	Object file format	`OffReader`	`OffWriter`
.vtk	VTK ascii format	`VtkAsciiReader`	
.vtp	VTK XML format	`VtkXmlReader`	

Table 2.2: Reader and writer classes associated with the different mesh file formats

The general usage pattern for these classes is to construct the desired reader or writer with a path to the desired file, and then call `readMesh()` or `writeMesh()` as appropriate:

```
// read a mesh from a .obj file:
WavefrontReader reader = new WavefrontReader ("meshes/torus.obj");
PolygonalMesh mesh = null;
try {
   mesh = reader.readMesh();
}
catch (IOException e) {
   System.err.println ("Can't read mesh:");
   e.printStackTrace();
}
```

Both `readMesh()` and `writeMesh()` may throw I/O exceptions, which must be either caught, as in the example above, or thrown out of the calling routine.

For convenience, one can also use the classes GenericMeshReader or GenericMeshWriter, which internally create an appropriate reader or writer based on the file extension. This enables the writing of code that does not depend on the file format:

```
String fileName;
...
PolygonalMesh mesh = null;
```

```
try {
   mesh = (PolygonalMesh)GenericMeshReader.readMesh(fileName);
}
catch (IOException e) {
   System.err.println ("Can't read mesh:");
   e.printStackTrace();
}
```

Here, `fileName` can refer to a mesh of any format supported by `GenericMeshReader`. Note that the mesh returned by `readMesh()` is explicitly cast to `PolygonalMesh`. This is because `readMesh()` returns the superclass `MeshBase`, since the default mesh created for some file formats may be different from `PolygonalMesh`.

2.5.6 Reading and writing normal and texture information

When writing a mesh out to a file, normal and texture information are also written if they have been explicitly set and the file format supports it. In addition, by default, automatically generated normal information will also be written if it relies on information (such as hard edges) that can't be reconstructed from the stored file information.

Whether or not normal information will be written is returned by the method

```
boolean getWriteNormals();
```

This will always return `true` if any of the conditions described above have been met. So for example, if a `PolygonalMesh` contains hard edges, and multiple automatic normals are enabled (i.e., `getMultipleAutoNormals()` returns `true`), then `getWriteNormals()` will return `true`.

Default normal writing behavior can be overridden within the MeshWriter classes using the following methods:

```
int getWriteNormals()
setWriteNormals (enable)
```

where `enable` should be one of the following values:

0 normals will *never* be written;

1 normals will *always* be written;

-1 normals will written according to the default behavior described above.

When reading a `PolygonalMesh` from a file, if the file contains normal information with multiple normals per vertex that suggests the existence of hard edges, then the corresponding edges are set to be hard within the mesh.

2.5.7 Constructive solid geometry

ArtiSynth contains primitives for performing constructive solid geometry (CSG) operations on volumes bounded by triangular meshes. The class that performs these operations is maspack.collision.SurfaceMeshIntersector, and it works by robustly determining the intersection contour(s) between a pair of meshes, and then using these to compute the triangles that need to be added or removed to produce the necessary CSG surface.

The CSG operations include union, intersection, and difference, and are implemented by the following methods of `SurfaceMeshIntersector`:

```
findUnion (mesh0, mesh1);          // volume0 U volume1
findIntersection (mesh0, mesh1);   // volume0 ^ volume1
findDifference01 (mesh0, mesh1);   // volume0 - volume1
findDifference10 (mesh0, mesh1);   // volume1 - volume0
```

Each takes two `PolyhedralMesh` objects, `mesh0` and `mesh1`, and creates and returns another `PolyhedralMesh` which represents the boundary surface of the requested operation. If the result of the operation is `null`, the returned mesh will be empty.

The example below uses `findUnion` to create a dumbbell shaped mesh from two balls and a cylinder:

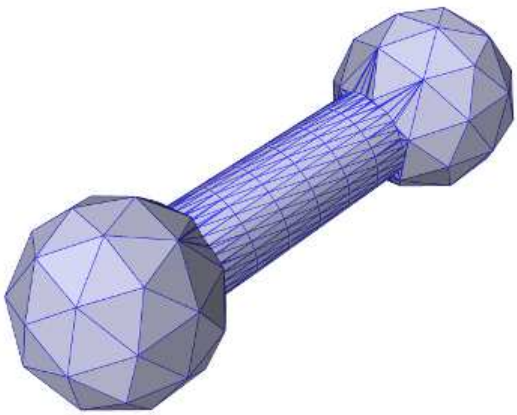


Figure 2.1: Dumbbell shaped mesh produced from the CSG union of two balls and a cylinder.

```
// first create two ball meshes and a bar mesh
double radius = 1.0;
int division = 1; // number of divisons for icosahedral sphere
PolygonalMesh ball0 = MeshFactory.createIcosahedralSphere (radius, division);
ball0.transform (new RigidTransform3d (0, -2*radius, 0));
PolygonalMesh ball1 = MeshFactory.createIcosahedralSphere (radius, division);
ball1.transform (new RigidTransform3d (0, 2*radius, 0));
PolygonalMesh bar = MeshFactory.createCylinder (
   radius/2, radius*4, /*ns=*/32, /*nr=*/1, /*nh*/10);
bar.transform (new RigidTransform3d (0, 0, 0, 0, 0, Math.PI/2));

// use a SurfaceMeshIntersector to create a CSG union of these meshes
SurfaceMeshIntersector smi = new SurfaceMeshIntersector();
PolygonalMesh balls = smi.findUnion (ball0, ball1);
PolygonalMesh mesh = smi.findUnion (balls, bar);
```

The balls and cylinder are created using the MeshFactory methods createIcosahedralSphere() and createCylinder(), where the latter takes arguments `ns`, `nr`, and `nh` giving the number of slices along the circumference, end-cap radius, and length. The final resulting mesh is shown in Figure 2.1.

2.6 Reading source relative files

ArtiSynth applications frequently need to read in various kinds of data files, including mesh files (as discussed in Section 2.5.5), FEM mesh geometry (Section 6.2.2), probe data (Section 5.4.4), and custom application data.

Often these data files do not reside in an absolute location but instead in a location relative to the application's class or source files. For example, it is common for applications to store geometric data in a subdirectory `"geometry"` located beneath the source directory. In order to access such files in a robust way, and ensure that the code does not break when the source tree is moved, it is useful to determine the application's source (or class) directory at run time. ArtiSynth supplies several ways to conveniently handle this situation. First, the `RootModel` itself supplies the following methods:

```
// find path to the root model's source directory
String findSourceDir ();

// get path to a file specified relative to the root model's source directory
String getSourceRelativePath (String relpath);
```

The first method returns the path to the source directory of the root model, while the second returns the path to a file specified relative to the root model source directory. If the root model source directory cannot be found (see discussion at the end of this section) both methods return `null`. As a specific usage example, assume that we have an application model whose `build()` method needs to load in a mesh `torus.obj` from a subdirectory `meshes` located beneath the source directory. This could be done as follows:

```
String pathToMesh = getSourceRelativePath ("meshes/torus.obj");
// read the mesh from a .obj file :
WavefrontReader reader = new WavefrontReader (pathToMesh);
PolygonalMesh mesh = null;
try {
   mesh = reader.readMesh () ;
}
catch (IOException e) {
   System.err.println ("Can't read mesh:");
   e.printStackTrace () ;
}
```

A more general path finding utility is provided by maspack.util.PathFinder, which provides several static methods for locating source and class directories:

```
// find path to the source directory associated with classObj
String findSourceDir (Object classObj);

// get path to a file specified relative to classObj source directory
String getSourceRelativePath (Object classObj, String relpath);

// find path to the class directory associated with classObj
String findClassDir (Object classObj);

// get path to a file specified relative to classObj class directory
String getClassRelativePath (Object classObj, String relpath);
```

The "find" methods return a string path to the indicated class or source directory, while the "relative path" methods locate the class or source directory and append the additional path `relpath`. For all of these, the class is determined from `classObj`, either directly (if it is an instance of `Class`), by name (if it is a `String`), or otherwise by calling `classObj.getClass()`. When identifying a package by name, the name should be either a fully qualified class name, or a simple name that can be located with respect to the packages obtained via `Package.getPackages()`. For example, if we have a class whose fully qualified name is `artisynth.models.test.Foo`, then the following calls should all return the same result:

```
Foo foo = new Foo();

PathFinder.findSourceDir (foo);
PathFinder.findSourceDir (Foo.class);
PathFinder.findSourceDir ("artisynth.models.test.Foo");
PathFinder.findSourceDir ("Foo");
```

If the source directory for `Foo` happens to be `/home/projects/src/artisynth/models/test`, then

```
PathFinder.getSourceRelativePath (foo, "geometry/mesh.obj");
```

will return `/home/projects/src/artisynth/models/test/geometry/mesh.obj`.

> When calling `PathFinder` methods from *within* the relevant class, one can specify `this` as the `classObj` argument.

With respect to the above example locating the file `"meshes/torus.obj"`, the call to the root model method `getSourceRelativePath()` could be replaced with

```
String pathToMesh = PathFinder.getSourceRelativePath (
   this, "meshes/torus.obj");
```

Since this is assumed to be called from the root model's build method, the "class" can be indicated by simply passing `this` to `getSourceRelativePath()`.

> As an alternative to placing data files in the source directory, one could place them in the class directory, and then use `findClassDir()` and `getClassRelativePath()`. If the data files were originally defined in the source directory, it will be necessary to copy them to the class directory. Some Java IDEs will perform this automatically.

The `PathFinder` methods work by climbing the class's resource hierarchy. Source directories are assumed to be located relative to the parent of the root class directory, via one of the paths specified by getSourceRootPaths(). By default, this list includes `"src"`, `"source"`, and `"bin"`. Additional paths can be added using addSourceRootPath(path), or the entire list can be set using setSourceRootPaths(paths).

At preset, source directories will not be found if the reference class is contained in a jar file.

2.7 Reading and caching remote files

ArtiSynth applications often require the use of large data files to specify items such as FEM mesh geometry, surface mesh geometry, or medical imaging data. The size of these files may make it inconvenient to store them in any version control system that is used to store the application source code. As an alternative, ArtiSynth provides a *file manager* utility that allows such files to be stored on a separate server, and then downloaded on-demand and cached locally. To use this, one starts by creating an instance of a FileManager, using the constructor

```
FileManager (String downloadPath, String remoteSourceName)
```

where `downloadPath` is a path to the local directory where the downloaded file should be placed, and `remoteSourceName` is a URI indicating the remote server location of the files. After the file manager has been created, it can be used to fetch remote files and cache them locally, using various *get* methods:

```
File get (String destName);

File get (String destName, String sourceName);
```

Both of these look for the file `destName` specified relative to the local directory, and return a `File` handle for it if it is present. Otherwise, they attempt to download the file from the remote source location, place it in the local directory, and return a `File` handle for it. The location of the remote file is given relative to the remote source URI by `destName` for the first method and `sourceName` for the second.

A simple example of using a file manager within a `RootModel build()` method is given by the following fragment:

```
// create the file manager ...
FileManager fm = new FileManager (
  getSourceRelativePath("geometry"),
  "http://myserver.org/artisynth/data/geometry");
// ... and use it to get a bone mesh file
File meshFile = fm.get ("tibia.obj");
```

Here, a file manager is created that uses a local directory `"geometry"`, located relative to the `RootModel` source directory (see Section 2.6), and looks for missing files relative to the URI

```
http://myserver.org/artisynth/data/geometry
```

The `get()` method is then used to obtain the file `"tibia.obj"` from the local directory. If it is not already present, it is downloaded from the remote location.

The FileManager contains other features and functionality, and one should consult its API documentation for more information.

Chapter 3

Mechanical Models I

This section details how to build basic multibody-type mechanical models consisting of particles, springs, rigid bodies, joints, and other constraints.

3.1 Springs and particles

The most basic type of mechanical model consists simply of particles connected together by axial springs. Particles are implemented by the class Particle, which is a dynamic component containing a three-dimensional position state, a corresponding velocity state, and a mass. It is an instance of the more general base class Point, which is used to also implement spatial points such as `markers` which do not have a mass.

3.1.1 Axial springs and materials

An axial spring is a simple spring that connects two points and is implemented by the class AxialSpring. This is a *force effector* component that exerts equal and opposite forces on the two points, along the line separating them, with a magnitude f that is a function $f(l,\dot{l})$ of the distance l between the points, and the distance derivative $\dot{l}$.

Each axial spring is associated with an *axial material*, implemented by a subclass of AxialMaterial, that specifies the function $f(l,\dot{l})$. The most basic type of axial material is a LinearAxialMaterial, which determines f according to the linear relationship

$$f(l,\dot{l}) = k(l-l_0)+d\dot{l} \tag{3.1}$$

where l_0 is the rest length and k and d are the stiffness and damping terms. Both k and d are properties of the material, while l_0 is a property of the spring.

Axial springs are assigned a linear axial material by default. More complex, nonlinear axial materials may be defined in the package `artisynth.core.materials`. Setting or querying a spring's material may be done with the methods `setMaterial()` and `getMaterial()`.

3.1.2 Example: a simple particle-spring model

An complete application model that implements a simple particle-spring model is given below.

```
1  package artisynth.demos.tutorial;
2
3  import java.awt.Color;
4  import maspack.matrix.*;
5  import maspack.render.*;
6  import artisynth.core.mechmodels.*;
7  import artisynth.core.materials.*;
8  import artisynth.core.workspace.RootModel;
9
10 /**
```

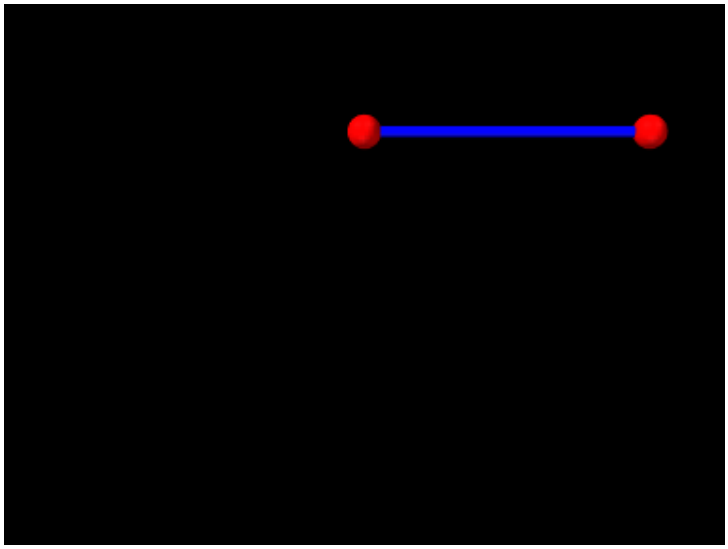


Figure 3.1: ParticleSpring model loaded into ArtiSynth.

```
11  * Demo of two particles connected by a spring
12  */
13 public class ParticleSpring extends RootModel {
14
15    public void build (String[] args) {
16
17       // create MechModel and add to RootModel
18       MechModel mech = new MechModel ("mech");
19       addModel (mech);
20
21       // create the components
22       Particle p1 = new Particle ("p1", /*mass=*/2, /*x,y,z=*/0, 0, 0);
23       Particle p2 = new Particle ("p2", /*mass=*/2, /*x,y,z=*/1, 0, 0);
24       AxialSpring spring = new AxialSpring ("spr", /*restLength=*/0);
25       spring.setPoints (p1, p2);
26       spring.setMaterial (
27          new LinearAxialMaterial (/*stiffness=*/20, /*damping=*/10));
28
29       // add components to the mech model
30       mech.addParticle (p1);
31       mech.addParticle (p2);
32       mech.addAxialSpring (spring);
33
34       p1.setDynamic (false);                // first particle set to be fixed
35
36       // increase model bounding box for the viewer
37       mech.setBounds (/*min=*/-1, 0, -1, /*max=*/1, 0, 0);
38       // set render properties for the components
39       RenderProps.setSphericalPoints (p1, 0.06, Color.RED);
40       RenderProps.setSphericalPoints (p2, 0.06, Color.RED);
41       RenderProps.setCylindricalLines (spring, 0.02, Color.BLUE);
42    }
43 }
```

Line 1 of the source defines the package in which the model class will reside, in this case `artisynth.demos.tutorial`. Lines 3-8 import definitions for other classes that will be used.

The model application class is named `ParticleSpring` and declared to extend `RootModel` (line 13), and the `build()` method definition begins at line 15. (A no-args constructor is also needed, but because no other constructors are defined, the compiler creates one automatically.)

To begin, the `build()` method creates a `MechModel` named `"mech"`, and then adds it to the `models` list of the root model

using the `addModel()` method (lines 18-19). Next, two particles, `p1` and `p2`, are created, with masses equal to 2 and initial positions at 0, 0, 0, and 1, 0, 0, respectively (lines 22-23). Then an axial spring is created, with end points set to `p1` and `p2`, and assigned a linear material with a stiffness and damping of 20 and 10 (lines 24-27). Finally, after the particles and the spring are created, they are added to the `particles` and `axialSprings` lists of the `MechModel` using the methods `addParticle()` and `addAxialSpring()` (lines 30-32).

At this point in the code, both particles are defined to be dynamically controlled, so that running the simulation would cause both to fall under the `MechModel`'s default gravity acceleration of $(0, 0, -9.8)$. However, for this example, we want the first particle to remain fixed in place, so we set it to be *non-dynamic* (line 34), meaning that the physical simulation will not update its position in response to forces (Section 3.1.3).

The remaining calls control aspects of how the model is graphically rendered. `setBounds()` (line 37) increases the model's "bounding box" so that by default it will occupy a larger part of the viewer frustum. The convenience method `RenderProps.setSphericalPoints()` is used to set points `p1` and `p2` to render as solid red spheres with a radius of 0.06, while `RenderProps.setCylindricalLines()` is used to set `spring` to render as a solid blue cylinder with a radius of 0.02. More details about setting render properties are given in Section 4.3.

To run this example in ArtiSynth, select All demos > tutorial > ParticleSpring from the Models menu. The model should load and initially appear as in Figure 3.1. Running the model (Section 1.5.3) will cause the second particle to fall and swing about under gravity.

3.1.3 Dynamic, parametric, and attached components

By default, a dynamic component is advanced through time in response to the forces applied to it. However, it is also possible to set a dynamic component's `dynamic` property to `false`, so that it does not respond to force inputs. As shown in the example above, this can be done using the method `setDynamic()`:

```
comp.setDynamic (false);
```

The method isDynamic() can be used to query the `dynamic` property.

Dynamic components can also be *attached* to other dynamic components (as mentioned in Section 1.2) so that their positions and velocities are controlled by the *master* components that they are attached to. To attach a dynamic component, one creates an `AttachmentComponent` specifying the attachment connection and adds it to the `MechModel`, as described in Section 3.6. The method isAttached() can be used to determine if a component is attached, and if it is, getAttachment() can be used to find the corresponding `AttachmentComponent`.

Overall, a dynamic component can be in one of three states:

active

Component is dynamic and unattached. The method isActive() returns `true`. The component will move in response to forces.

parametric

Component is not dynamic, and is unattached. The method isParametric() returns `true`. The component will either remain fixed, or will move around in response to external inputs specifying the component's position and/or velocity. One way to supply such inputs is to use controllers or input probes, as described in Section 5.

attached

Component is attached. The method isAttached() returns `true`. The component will move so as to follow the other master component(s) to which it is attached.

3.1.4 Custom axial materials

Application authors may create their own axial materials by subclassing AxialMaterial and overriding the functions

```
double computeF (l, ldot, l0, excitation);
double computeDFdl (l, ldot, l0, excitation);
double computeDFdldot (l, ldot, l0, excitation);
boolean isDFdldotZero ();
```

where `excitation` is an additional *excitation* signal a, which is used to implement active springs and which in particular is used to implement axial muscles (Section 4.4), for which a is usually in the range $[0,1]$.

The first three methods should return the values of

$$f(l,\dot{l},a), \quad \frac{\partial f(l,\dot{l},a)}{\partial l}, \quad \text{and} \quad \frac{\partial f(l,\dot{l},a)}{\partial \dot{l}}, \tag{3.2}$$

respectively, while the last method should return `true` if $\partial f(l,\dot{l},a)/\partial \dot{l} \equiv 0$; i.e., if it is always equals to 0.

3.1.5 Damping parameters

Mechanical models usually contain damping forces in addition to spring-type restorative forces. Damping generates forces that reduce dynamic component velocities, and is usually the major source of energy dissipation in the model. Damping forces can be generated by the spring components themselves, as described above.

A general damping can be set for all particles by setting the `MechModel`'s `pointDamping` property. This causes a force

$$\mathbf{f}_i = -d_p \mathbf{v}_i \tag{3.3}$$

to be applied to all particles, where d_p is the value of the `pointDamping` and $\mathbf{v}_i$ is the particle's velocity.

`pointDamping` can be set and queried using the `MechModel` methods

```
setPointDamping (double d);
double getPointDamping();
```

> In general, whenever a component has a property `propX`, that property can be set and queried in code using methods of the form
>
> ```
> setPropX (T d);
> T getPropX();
> ```
>
> where `T` is the type associated with the property.

`pointDamping` can also be set for particles individually. This property is *inherited* (Section 1.4.3), so that if not set explicitly, it inherits the nearest explicitly set value in an ancestor component.

3.2 Rigid bodies

Rigid bodies are implemented in ArtiSynth by the class RigidBody, which is a dynamic component containing a six-dimensional position and orientation state, a corresponding velocity state, an inertia, and an optional surface mesh.

A rigid body is associated with its own 3D spatial coordinate frame, and is a subclass of the more general Frame component. The combined position and orientation of this frame with respect to world coordinates defines the body's *pose*, and the associated 6 degrees of freedom describe its "position" state.

3.2.1 Frame markers

ArtiSynth makes extensive use of *markers*, which are (massless) points attached to dynamic components in the model. Markers are used for graphical display, implementing attachments, and transmitting forces back onto the underlying dynamic components.

A *frame marker* is a marker that can be attached to a Frame, and most commonly to a RigidBody (Figure 3.2). They are frequently used to provide the anchor points for attaching springs and, more generally, applying forces to the body.

Frame markers are implemented by the class FrameMarker, which is a subclass of Point. The methods

```
Point3d getLocation();
void setLocation (Point3d r);
```

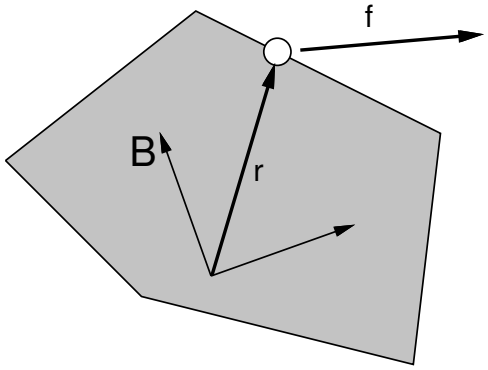


Figure 3.2: A force **f** applied to a frame marker attached to a rigid body. The marker is located at the point **r** with respect to the body coordinate frame B.

get and set the marker's location **r** with respect to the frame's coordinate system. When a 3D force **f** is applied to the marker, it generates a spatial force $\hat{\mathbf{f}}$ (Section A.5) on the frame given by

$$\hat{\mathbf{f}} = \begin{pmatrix} \mathbf{f} \\ \mathbf{r} \times \mathbf{f} \end{pmatrix}. \tag{3.4}$$

Frame markers can be created using a variety of constructors, including

```
FrameMarker ();
FrameMarker (String name);
FrameMarker (Frame frame, Point3d loc);
```

where `FrameMarker()` creates an empty marker, `FrameMarker(name)` creates an empty marker with a name, and `FrameMarker(frame,loc)` creates an unnamed marker attached to `frame` at the location `loc` with respect to the frame's coordinates. Once created, a marker's frame can be set and queried with

```
void setFrame (Frame frame);
Frame getFrame ();
```

A frame marker can be added to a MechModel with the `MechModel` methods

```
void addFrameMarker (FrameMarker mkr);
void addFrameMarker (FrameMarker mkr, Frame frame, Point3d loc);
```

where `addFrameMarker(mkr,frame,loc)` also sets the frame and the marker's location with respect to it.

`MechModel` also supplies convenience methods to create a marker, attach it to a frame, and add it to the model:

```
FrameMarker addFrameMarker (Frame frame, Point3d loc);
FrameMarker addFrameMarkerWorld (Frame frame, Point3d locw);
```

Both methods return the created marker. The first, `addFrameMarker(frame,loc)`, places it at the location `loc` with respect to the frame, while `addFrameMarkerWorld(frame,pos)` places it at `pos` with respect to *world* coordinates.

3.2.2 Example: a simple rigid body-spring model

A simple rigid body-spring model is defined in

```
artisynth.demos.tutorial.RigidBodySpring
```

This differs from ParticleSpring only in the `build()` method, which is listed below:

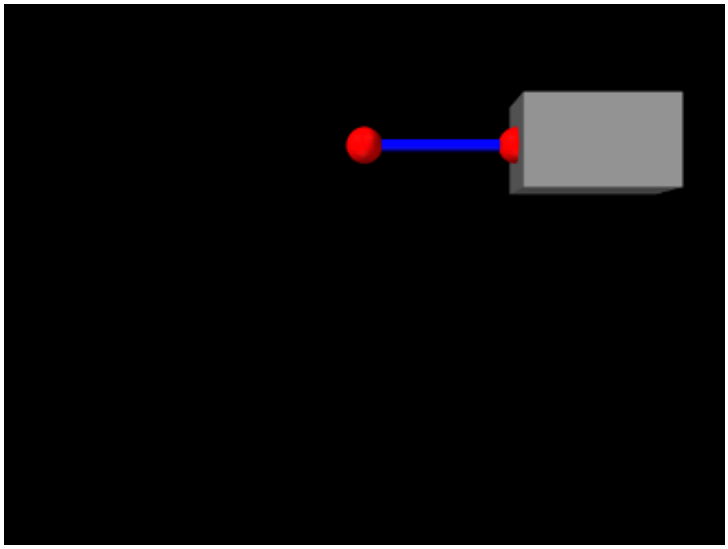


Figure 3.3: RigidBodySpring model loaded into ArtiSynth.

```
1   public void build (String[] args) {
2
3      // create MechModel and add to RootModel
4      MechModel mech = new MechModel ("mech");
5      addModel (mech);
6
7      // create the components
8      Particle p1 = new Particle ("p1", /*mass=*/2, /*x,y,z=*/0, 0, 0);
9      // create box and set it's pose (position/orientation):
10     RigidBody box =
11        RigidBody.createBox ("box", /*wx,wy,wz=*/0.5, 0.3, 0.3, /*density=*/20);
12     box.setPose (new RigidTransform3d (/*x,y,z=*/0.75, 0, 0));
13     // create marker point and connect it to the box:
14     FrameMarker mkr = new FrameMarker (/*x,y,z=*/-0.25, 0, 0);
15     mkr.setFrame (box);
16
17     AxialSpring spring = new AxialSpring ("spr", /*restLength=*/0);
18     spring.setPoints (p1, mkr);
19     spring.setMaterial (
20        new LinearAxialMaterial (/*stiffness=*/20, /*damping=*/10));
21
22     // add components to the mech model
23     mech.addParticle (p1);
24     mech.addRigidBody (box);
25     mech.addFrameMarker (mkr);
26     mech.addAxialSpring (spring);
27
28     p1.setDynamic (false);               // first particle set to be fixed
29
30     // increase model bounding box for the viewer
31     mech.setBounds (/*min=*/-1, 0, -1, /*max=*/1, 0, 0);
32     // set render properties for the components
33     RenderProps.setSphericalPoints (p1, 0.06, Color.RED);
34     RenderProps.setSphericalPoints (mkr, 0.06, Color.RED);
35     RenderProps.setCylindricalLines (mkr, 0.02, Color.BLUE);
36  }
```

The differences from `ParticleSpring` begin at line 9. Instead of creating a second particle, a rigid body is created using the factory method RigidBody.createBox(), which takes x, y, z widths and a (uniform) density and creates a box-shaped rigid body complete with surface mesh and appropriate mass and inertia. As the box is initially centered at the

origin, moving it elsewhere requires setting the body's pose, which is done using `setPose()`. The `RigidTransform3d` passed to `setPose()` is created using a three-argument constructor that generates a translation-only transform. Next, starting at line 14, a `FrameMarker` is created for a location $(-0.25, 0, 0)^T$ relative to the rigid body, and attached to the body using its `setFrame()` method.

The remainder of `build()` is the same as for `ParticleSpring`, except that the spring is attached to the frame marker instead of a second particle.

To run this example in ArtiSynth, select All demos > tutorial > RigidBodySpring from the Models menu. The model should load and initially appear as in Figure 3.3. Running the model (Section 1.5.3) will cause the rigid body to fall and swing about under gravity.

3.2.3 Creating rigid bodies

As illustrated above, rigid bodies can be created using factory methods supplied by RigidBody. Some of these include:

```
createBox (name, widthx, widthy, widthz, density);
createCylinder (name, radius, height, density, nsides);
createSphere (name, radius, density, nslices);
createEllipsoid (name, radx, rady, radz, density, nslices);
```

The bodies do not need to be named; if no name is desired, then `name` and can be specified as `null`.

In addition, there are also factory methods for creating a rigid body directly from a mesh:

```
createFromMesh (name, mesh, density, scale);
createFromMesh (name, meshFileName, density, scale);
```

These take either a polygonal mesh (Section 2.5), or a file name from which a mesh is read, and use it as the body's surface mesh and then compute the mass and inertia properties from the specified (uniform) density.

Alternatively, one can create a rigid body directly from a constructor, and then set the mesh and inertia properties explicitly:

```
PolygonalMesh femurMesh;
SpatialInertia inertia;

... initialize mesh and inertia appropriately ...

RigidBody body = new RigidBody ("femur");
body.setMesh (femurMesh);
body.setInertia (inertia);
```

3.2.4 Pose and velocity

A body's pose can be set and queried using the methods

```
setPose (RigidTransform3d T);   // sets the pose to T
getPose (RigidTransform3d T);   // gets the current pose in T
RigidTransform3d getPose();     // returns the current pose (read-only)
```

These use a RigidTransform3d (Section 2.2) to describe the pose. Body poses are described in world coordinates and specify the transform from body to world coordinates. In particular, the pose for a body A specifies the rigid transform $\mathbf{T}_{AW}$.

Rigid bodies also expose the translational and rotational components of their pose via the properties `position` and `orientation`, which can be queried and set independently using the methods

```
setPosition (Point3d p);       // sets the position to p
getPosition (Point3d p);       // gets the current position in p
Point3d getPosition();         // returns the current position (read-only)

setOrientation (AxisAngle a);  // sets the orientation to a
getOrientation (AxisAngle a);  // gets the current orientation in a
AxisAngle getOrientation();    // returns the current orientation (read-only)
```

The velocity of a rigid body is described using a Twist (Section 2.4), which contains both the translational and rotational velocities. The following methods set and query the spatial velocity as described with respect to world coordinates:

```
setVelocity (Twist v);          // sets the spatial velocity to v
getVelocity (Twist v);          // gets the current spatial velocity in v
Twist getVelocity();            // returns current spatial velocity (read-only)
```

During simulation, unless a rigid body has been set to be *parametric* (Section 3.1.3), its pose and velocity are updated in response to forces, so setting the pose or velocity generally makes sense only for setting initial conditions. On the other hand, if a rigid body is parametric, then it is possible to control its pose during the simulation, but in that case it is better to set its *target pose* and/or *target velocity*, as described in Section 5.3.1.

3.2.5 Inertia and the surface mesh

The "mass" of a rigid body is described by its spatial inertia (Section A.6), implemented by a SpatialInertia object, which specifies its mass, center of mass, and rotational inertia with respect to the center of mass.

Most rigid bodies are also associated with a polygonal surface mesh, which can be set and queried using the methods

```
setSurfaceMesh (PolygonalMesh mesh);
setSurfaceMesh (PolygonalMesh mesh, String meshFileName);
PolygonalMesh getSurfaceMesh();
```

The second method takes an optional `fileName` argument that can be set to the name of a file from which the mesh was read. Then if the model itself is saved to a file, the model file will specify the mesh using the file name instead of explicit vertex and face information, which can reduce the model file size considerably.

Rigid bodies can also have more than one mesh, as described in Section 3.2.8.

The inertia of a rigid body can be explicitly set using a variety of methods including

```
setInertia (M)                  // set using SpatialInertia M
setInertia (mass, Jxx, Jyy, Jzz); // mass and diagonal rotational inertia
setInertia (mass, J);           // mass and full rotational inertia
setInertia (mass, J, com);      // mass, rotational inertia, center-of-mass
```

and can be queried using

```
getInertia (M);                 // get SpatialInertia in M
getInertia ();                  // return read-only SpatialInertia
```

In practice, it is often more convenient to simply specify a mass or a density, and then use the geometry of the surface mesh (and possibly other meshes, Section 3.2.8) to compute the remaining inertial values. How a rigid body's inertia is computed is determined by its inertiaMethod property, which can be one

EXPLICIT

Inertia is set explicitly.

MASS

Inertia is determined implicitly from the mesh geometry and the body's mass.

DENSITY

Inertia is determined implicitly from the mesh geometry and the body's density (which is multiplied by the mesh volume(s) to determine a mass).

> When using `DENSITY` to determine the inertia, it is generally assumed that the contributing meshes are both polygonal and closed. Meshes which are either open or non-polygonal generally do not have a well-defined volume which can be multiplied by the density to determine the mass.

The inertiaMethod property can be set and queried using

```
setInertiaMethod (InertiaMethod method);
InertiaMethod getInertiaMethod();
```

and its default value is `DENSITY`. Explicitly setting the inertia using one of `setInertia()` methods described above will set `inertiaMethod` to `EXPLICIT`. The method

```
setInertiaFromDensity (density);
```

will (re)compute the inertia using the mesh geometry and a density value and set `inertiaMethod` to `DENSITY`, and the method

```
setInertiaFromMass (mass);
```

will (re)compute the inertia using the mesh geometry and a mass value and set `inertiaMethod` to `MASS`.

Finally, the (assumed uniform) density of the body can be queried using

```
getDensity();
```

> There are some subtleties involved in determining the inertia using either the `DENSITY` or `MASS` methods when the rigid body contains more than one mesh. Details are given in Section 3.2.8.

3.2.6 Damping parameters

As with particles, it is possible to set damping parameters for rigid bodies.

`MechModel` provides two properties, `frameDamping` and `rotaryDamping`, which generate a spatial force centered on each rigid body's coordinate frame

$$\hat{\mathbf{f}}_i = \begin{pmatrix} -d_f \mathbf{v}_i \\ -d_r \boldsymbol{\omega}_i \end{pmatrix}, \tag{3.5}$$

where d_f and d_r are the `frameDamping` and `rotaryDamping` values, and $\mathbf{v}_i$ and $\boldsymbol{\omega}_i$ are the translational and angular velocity of the body's coordinate frame. The damping parameters can be set and queried using the `MechModel` methods

```
setFrameDamping (double df);
setRotaryDamping (double dr);
double getFrameDamping();
double getRotaryDamping();
```

> For models involving rigid bodies, it is often necessary to set `rotaryDamping` to a non-zero value because `frameDamping` will provide no damping at all when a rigid body is simply rotating about its coordinate frame origin.

Frame and rotary damping can also be set for individual bodies using their own (inherited) `frameDamping` and `rotaryDamping` properties.

3.2.7 Rendering rigid bodies

A rigid body is rendered in ArtiSynth by drawing its mesh (or meshes, Section 3.2.8) and/or coordinate frame.

Meshes are drawn using the face rendering properties described in more detail in Section 4.3. The most commonly used of these are:

- faceColor: A value of type `java.awt.Color` giving the color of mesh faces. The default value is `GRAY`.
- shading: A value of type Renderer.Shading indicating how the mesh should be shaded, with the options being `FLAT`, `SMOOTH`, `METAL`, and `NONE`. The default value is `FLAT`.

- alpha: A double value between 0 and 1 indicating transparency, with transparency increasing as value decreases from 1. The default value is 1.
- faceStyle: A value of type Renderer.FaceStyle indicating which face sides should be drawn, with the options being `FRONT`, `BACK`, `FRONT_AND_BACK`, and `NONE`. The default value is `FRONT`.
- drawEdges: A boolean indicating whether the mesh edges should also be drawn, using either the edgeColor rendering property, or the lineColor property if edgeColor is not set. The default value is `false`.

These properties, and others, can be set either interactively in the GUI, or in code. To set the render properties in the GUI, select the rigid body or its mesh component, and then right click the mouse and choose Edit render props .... More details are given in the section "Render properties" in the ArtiSynth User Interface Guide.

Figure 3.4: Different rendering settings for a rigid body hip mesh showing the default (left), smooth rendering with a lighter color (center), and wireframe (right).

Properties can also be set in code, usually during the `build()` method. Typically this is done using a static method of the RenderProps class that has the form

```
RenderProps.setXXX (comp, value)
```

where `XXX` is the property name, `comp` is the component for which the property should be set, and `value` is the desired value. Some examples are shown in Figure 3.4 for a rigid body hip representation with a fairly coarse mesh. The left image shows the default rendering, using a gray color and flat shading. The center image shows a lighter color and smooth shading, which could be set by the following code fragment:

```
import maspack.render.*;
import maspack.render.Renderer.*;
  ...

  RigidBody hipBody;
  ...

  RenderProps.setFaceColor (hipBody, new Color (255, 255, 204));
  RenderProps.setShading (hipBody, Shading.SMOOTH);
```

Finally, the right image shows the body rendered as a wire frame, which can by done by setting faceStyle to `NONE` and drawEdges to `true`:

```
RenderProps.setFaceStyle (hip, FaceStyle.NONE);
RenderProps.setDrawEdges (hip, true);
RenderProps.setEdgeWidth (hip, 2);
RenderProps.setEdgeColor (hip, Color.CYAN);
```

Render properties can also be set in higher level model components, from which their values will be inherited by lower level components that have not explicitly set their own values. For example, setting the faceColor render property in the

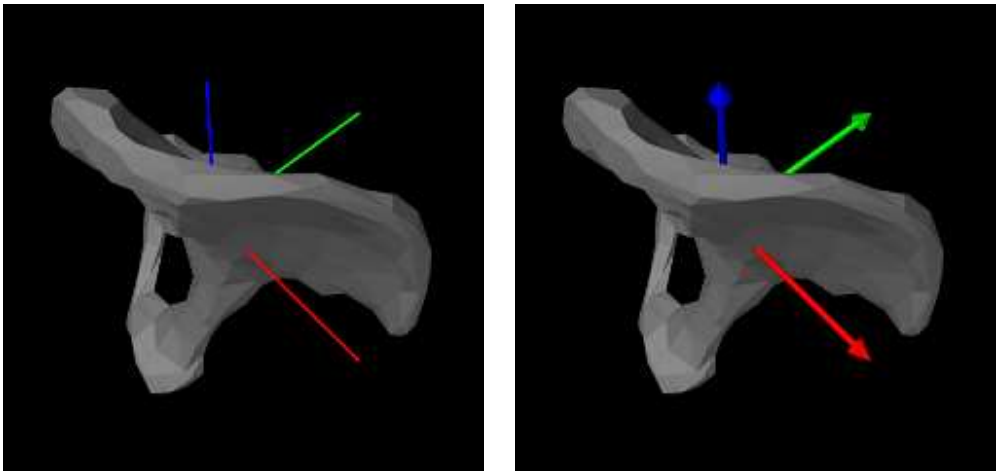


Figure 3.5: Rigid body axes rendered with axisDrawStyle set to `LINE` (left) and `ARROW` (right).

`MechModel` will automatically set the face color for all subcomponents which have not explicitly set faceColor. More details on render properties are given in Section 4.3.

In addition to mesh rendering, it is often useful to draw a rigid body's coordinate frame, which can be done using its axisLength and axisDrawStyle properties. Setting axisLength to a positive value will cause the body's three coordinate axes to be drawn, with the indicated length, with the x, y and z axes colored red, green, and blue, respectively. The `axisDrawStyle` property controls how the axes are rendered (Figure 3.5). It has the type Renderer.AxisDrawStyle, and can be set to the following values:

OFF

Axes are not rendered.

LINE

Axes are rendered as simple red-green-blue lines, with a width given by the joint's lineWidth rendering property.

ARROW

Axes are rendered as solid red-green-blue arrows.

As with the rendering proprieties, the axisLength and axisDrawStyle properties can be managed either interactively in the GUI (by selecting the body, right clicking and choosing Edit properties ...), or in code, using the following methods:

```
double getAxisLength()
void setAxisLength (double len)

AxisDrawStyle getAxisDrawStyle()
void setAxisDrawStyle (AxisDrawStyle style)
```

3.2.8 Multiple meshes

A `RigidBody` may contain multiple meshes, which can be useful for various reasons:

- It may be desirable to use different meshes for collision detection, inertia computation, and visual presentation;
- Different render properties can be set for different mesh components, allowing the body to be rendered in a more versatile way;
- Different mesh components can be selected individually.

Each rigid body mesh is encapsulated inside a RigidMeshComp component, which is in turn stored in a subcomponent list called `meshes`. Meshes do not need to be instances of PolygonalMesh; instead, they can be any instance of MeshBase, including PointMesh and PolylineMesh.

The default surface mesh, returned by `getSurfaceMesh()`, is also stored inside a `RigidMeshComp` in the `meshes` list. By default, the surface mesh is the first mesh in the list, but is otherwise defined to be the first mesh in `meshes` which is also an instance of `PolygonalMesh`. The `RigidMeshComp` containing the surface mesh can be obtained using the method `getSurfaceMeshComp()`.

A `RigidMeshComp` contains a number of properties that control how the mesh is displayed and interacts with its rigid body:

renderProps

Render properties controlling how the mesh is rendered (see Section 4.3).

hasMass

A boolean, which if `true` means that the mesh will contribute to the body's inertia when the inertiaMethod is either `MASS` or `DENSITY`. The default value is `true`.

massDistribution

An enumerated type defined by MassDistribution which specifies how the mesh's inertia contribution is determined for a given mass. `VOLUME`, `AREA`, `LENGTH`, and `POINT` indicate, respectively, that the mass is distributed evenly over the mesh's volume, area (faces), length (edges), or points. The default value is determined by the mesh type: `VOLUME` for a closed `PolygonalMesh`, `AREA` for an open `PolygonalMesh`, `LENGTH` for a `PolylineMesh`, and `POINT` for a `PointMesh`. Applications can specify an alternate value providing the mesh has the features to support it. Specifying `DEFAULT` will restore the default value.

isCollidable

A boolean, which if `true`, and if the mesh is a `PolygonalMesh`, means that the mesh will take part in collision and wrapping interactions (Sections 4.5 and 7.3). The default value is `true`, and the get/set accessors have the names `isCollidable()` and `setIsCollidable()`.

volume

A double whose value is the volume of the mesh. If the mesh is a `PolygonalMesh`, this is the value returned by its computeVolume() method. Otherwise, the volume is 0, unless setVolume(vol) is used to explicitly set a non-zero volume value.

mass

A double whose default value is the product of the density and volume properties. Otherwise, if mass has been explicitly set using setMass(mass), the value is the explicit mass.

density

A double whose default value is the rigid body's density. Otherwise, if density has been explicitly set using setDensity(density), the value is the explicit density, or if mass has been explicitly set using setMass(mass), the value is the explicit mass divided by volume.

Note that by default, the density of a `RigidMeshComp` is simply the density setting for the rigid body, and the mass is this times the volume. However, it is possible to set either an explicit mass or a density value that will override this. (Also, explicitly setting a mass will unset any explicit density, and explicitly setting the density will unset any explicit mass.)

When the inertiaMethod of the rigid body is either `MASS` or `DENSITY`, then its inertia is computed from the sum of all the inertias $\mathbf{M}_k$ of the component meshes k for which hasMass is `true`. Each $\mathbf{M}_k$ is computed by the mesh's createInertia(mass,massDistribution) method, using the mass and massDistribution properties of its `RigidMeshComp`.

When forming the body inertia from the inertia components of individual meshes, no attempt is made to account for mesh overlap. If this is important, the meshes themselves should be modified in advance so that they do not overlap, perhaps by using the CSG primitives described in Section 2.5.7.

Instances of `RigidMeshComp` can be created directly, using constructions such as

```
PolygonalMesh mesh;

... initialize mesh ...

RigidMeshComp mcomp = new RigidMeshComp (mesh);
```

or

```
RigidMeshComp mcomp = new RigidMeshComp ("meshName");
mcomp.setMesh (mesh);
```

after which they can be added or removed from the `meshes` list using the methods

```
void addMeshComp (RigidMeshComp mcomp)
void addMeshComp (RigidMeshComp mcomp, int idx)
int numMeshComps()
boolean removeMeshComp (RigidMeshComp mcomp)
boolean removeMeshComp (String name)
void clearMeshComps()
```

It is also possible to add meshes directly to the `meshes` list, using the methods

```
RigidMeshComp addMesh (MeshBase mesh)
RigidMeshComp addMesh (MeshBase mesh, boolean hasMass, boolean collidable)
```

each of which creates a `RigidMeshComp`, adds it to the mesh list, and returns it. The second method also specifies the values of the hasMass and collidable properties (both of which are `true` by default).

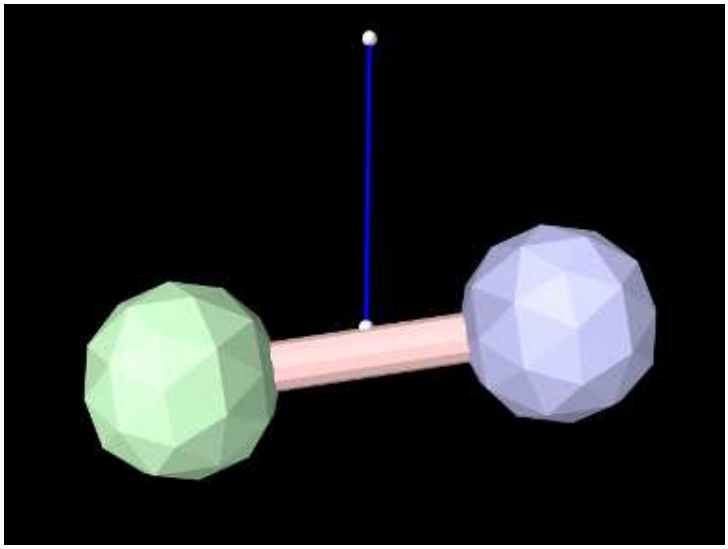


Figure 3.6: `RigidCompositeBody` loaded into ArtiSynth and run for 0.75 seconds. The ball on the right falls less because it has a lower density than the rest of the body.

3.2.9 Example: a composite rigid body

An example of constructing a rigid body from multiple meshes is defined in

```
artisynth.demos.tutorial.RigidCompositeBody
```

This uses three meshes to construct a rigid body whose shape resembles a dumbbell. The code, with the include files omitted, is listed below:

```
1  public class RigidCompositeBody extends RootModel {
2
3     public void build (String[] args) {
4
5        // create MechModel and add to RootModel
6        MechModel mech = new MechModel ("mech");
7        addModel (mech);
8
9        // create the component meshes
10       PolygonalMesh ball1 = MeshFactory.createIcosahedralSphere (0.8, 1);
11       ball1.transform (new RigidTransform3d (1.5, 0, 0));
12       PolygonalMesh ball2 = MeshFactory.createIcosahedralSphere (0.8, 1);
13       ball2.transform (new RigidTransform3d (-1.5, 0, 0));
14       PolygonalMesh axis = MeshFactory.createCylinder (0.2, 2.0, 12);
15       axis.transform (new RigidTransform3d (0, 0, 0, 0, Math.PI/2, 0));
16
17       // create the body and add the component meshes
18       RigidBody body = new RigidBody ("body");
19       body.setDensity (10);
20       body.setFrameDamping (10); // add damping to the body
21       body.addMesh (axis);
22       RigidMeshComp bcomp1 = body.addMesh (ball1);
23       RigidMeshComp bcomp2 = body.addMesh (ball2);
24       mech.addRigidBody (body);
25
26       // connect the body to a spring attached to a fixed particle
27       Particle p1 = new Particle ("p1", /*mass=*/0, /*x,y,z=*/0, 0, 2);
28       p1.setDynamic (false);
29       mech.addParticle (p1);
30       FrameMarker mkr = mech.addFrameMarkerWorld (body, new Point3d (0, 0, 0.2));
31       AxialSpring spring =
32          new AxialSpring ("spr", /*k=*/150, /*d=*/0, /*restLength=*/0);
33       spring.setPoints (p1, mkr);
34       mech.addAxialSpring (spring);
35
36       // set the density for ball1 to be less than the body density
37       bcomp1.setDensity (8);
38
39       // set render properties for the component, with the ball
40       // meshes having different colors
41       RenderProps.setFaceColor (body, new Color (250, 200, 200));
42       RenderProps.setFaceColor (bcomp1, new Color (200, 200, 250));
43       RenderProps.setFaceColor (bcomp2, new Color (200, 250, 200));
44       RenderProps.setSphericalPoints (mech, 0.06, Color.WHITE);
45       RenderProps.setCylindricalLines (spring, 0.02, Color.BLUE);
46    }
47 }
```

As in the previous examples, the `build()` method starts by creating a `MechModel` (lines 6-7). Three different meshes (two balls and an axis) are then constructed at lines 10-15, using `MeshFactory` methods (Section 2.5) and transforming each result to an appropriate position/orientation with respect to the body's coordinate frame.

The body itself is constructed at lines 18-24. Its default density is set to 10, and its frame damping (Section 3.2.6) is also set to 10 (the previous rigid body example in Section 3.2.2 relied on spring damping to dissipate energy). The meshes are added using addMesh(), which allocates and returns a RigidMeshComp. For the ball meshes, these are saved in `bcomp1` and `bcomp2` and used later to adjust density and/or render properties.

Lines 27-34 create a simple linear spring, connected to a fixed point `p0` and a marker `mkr`. The marker is created and attached to the body by the `MechModel` method addFrameMarkerWorld(), which places the marker at a known position in world coordinates. The spring is created using an AxialSpring constructor that accepts a name, along with stiffness, damping, and rest length parameters to specify a LinearAxialMaterial.

At line 37, `bcomp1` is used to set the density of `ball1` to 8. Since this is less than the default body density, the inertia component of `ball1` will be lighter than that of `ball2`. Finally, render properties are set at lines 41-45. This includes setting the default face colors for the body and for each ball.

To run this example in ArtiSynth, select All demos > tutorial > RigidCompositeBody from the Models menu. The model should load and initially appear as in Figure 3.6. Running the model (Section 1.5.3) will cause the rigid body to fall and swing about under gravity, with the right ball (`ball1`) not falling as far because it has less density.

3.3 Joints and connectors

In a typical mechanical model, many of the rigid bodies are interconnected, either using spring-type components that exert binding forces on the bodies, or through joints and connectors that enforce the connection using hard constraints. This section describes the latter. While the discussion focuses on rigid bodies, joints and connectors can be used more generally with any body that implements the ConnectableBody interface. In particular, this allows joints to also interconnect finite element models, as described in Section 6.6.2.

3.3.1 Joints and coordinate frames

Consider two rigid bodies A and B. The pose of body B with respect to body A can be described by the 6 DOF rigid transform $\mathbf{T}_{BA}$. If A and B are unconnected, $\mathbf{T}_{BA}$ may assume any possible value and has a full six degrees of freedom. A *joint* between A and B constrains the set of poses that are possible between the two bodies and reduces the degrees of freedom available to $\mathbf{T}_{BA}$. For ease of use, the constraining action of a joint is described with respect to a pair of local coordinate frames C and D that are connected to frames A and B, respectively, by auxiliary transformations. This allows joints to be placed at locations that do not correspond directly to frames A or B.

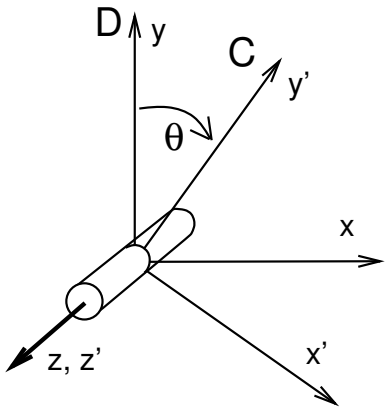


Figure 3.7: Coordinate frames D and C for a hinge joint.

The joint frames C and D move with respect to each other as the joint moves. The allowed joint motions therefore correspond to the allowed values of the *joint transform* $\mathbf{T}_{CD}$. Although both frames typically move with their attached bodies, D is considered the *base* frame and C the *motion* frame (this is because when a joint is used to connect a single body to ground, body B is set to `null` and the world frame takes its place). As an example of a joint's constraining effect, consider a hinge joint (Figure 3.7), which allows C to move with respect to D *only* by rotating about the z axis while the origins of C and D remain coincident. Other motions are prohibited. If we let θ describe the counter-clockwise rotation angle of C about the z axis, then $\mathbf{T}_{CD}$ should always have the form

$$\mathbf{T}_{CD} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.6}$$

When a joint is attached to bodies A and B, frame C is fixed to body A and frame D is fixed to body B. Except in special cases, the joint frames C and D are not coincident with the body frames A and B. Instead, they are located relative to A and B by the transforms $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$, respectively (Figure 3.8). Since $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ are both fixed, the joint constraints on $\mathbf{T}_{CD}$ constrain the relative poses of A and B, with $\mathbf{T}_{AB}$ determined from

$$\mathbf{T}_{AB} = \mathbf{T}_{DB}\,\mathbf{T}_{CD}\,\mathbf{T}_{CA}^{-1}. \tag{3.7}$$

(See Section A.2 for a discussion of determining transforms between related coordinate frames).

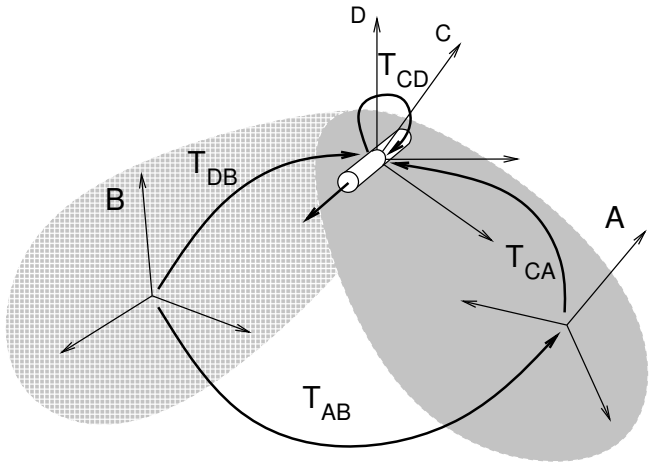


Figure 3.8: Transforms connecting joint coordinate frames C and D with rigid bodies A and B.

3.3.2 Joint coordinates, constraints, and errors

Each different joint and connector type restricts the motion between two bodies to M degrees of freedom, for some $M < 6$. Sometimes, the joint also defines a set of M coordinates that parameterize these M DOFs. For example, the hinge joint described above is parameterized by θ. Other examples are given in Section 3.4: a 2 DOF cylindrical has coordinates z and θ, a 3 DOF gimbal joint is parameterized by the roll-pitch-yaw angles θ, ϕ, and ψ, etc. When $\mathbf{T}_{CD} = \mathbf{I}$ (where $\mathbf{I}$ is the identity transform), the coordinates are usually all equal to zero, and the joint is said to be in the *zero state*.

As explained in Section 1.2, ArtiSynth uses a full coordinate formulation for dynamic simulation. That means that instead of using joint coordinates to describe system state, it uses the combined full coordinates $\mathbf{q}$ of all dynamic components. For example, a model consisting of a single rigid body connected to ground by a hinge joint will have 6 DOF (corresponding to the 6 DOF of the body), rather than the 1 DOF implied by the hinge joint. The DOF restrictions imposed by the joints are then enforced by a set of linearized constraint relationships

$$\mathbf{G}(\mathbf{q})\mathbf{u} = \mathbf{g}, \qquad \mathbf{N}(\mathbf{q})\mathbf{u} \geq \mathbf{n} \tag{3.8}$$

that restrict the body velocities $\mathbf{u}$ computed at each simulation step, usually by solving an MLCP like (1.6). As explained in Section 1.2, the right side vectors $\mathbf{g}$ and $\mathbf{n}$ in (3.8) contain time derivative terms, which for simplicity much of the following presentation will assume to be 0.

Each joint contributes its own set of constraint equations to (3.8). Typically these take the form of *bilateral*, or equality, constraints

$$\mathbf{G}_\text{J}(\mathbf{q})\mathbf{u} = 0 \tag{3.9}$$

which are added to the system's global bilateral constraint matrix $\mathbf{G}$. $\mathbf{G}_\text{J}$ contains $6-M$ rows providing $6-M$ individual constraints $\mathbf{G}_k$. During simulation, these give rise to $6-M$ constraint forces (corresponding to λ in (1.8)) which enforce the constraints.

In some cases, the joint also maintains *unilateral*, or inequality constraints, to keep $\mathbf{T}_{CD}$ out of inadmissible regions. These take the form

$$\mathbf{N}_\text{J}(\mathbf{q})\mathbf{u} \geq 0 \tag{3.10}$$

and are added to the system's global unilateral constraint matrix $\mathbf{N}$. They give rise to constraint forces corresponding to θ in (1.8). A common use of unilateral constraints is to enforce range limits of the joint coordinates, such as

$$\theta_\text{min} \leq \theta \leq \theta_\text{max}. \tag{3.11}$$

A specific unilateral constraint is added to $\mathbf{N}_\text{J}$ only when $\mathbf{T}_{CD}$ is on or within the boundary of the inadmissible region associated with that constraint. The constraint is then said to be *engaged*. The combined number of bilateral and engaged unilateral constraints for a particular joint should not exceed 6; otherwise, the joint would be overconstrained.

Joint coordinates, when supported for a particular joint, can be both read and set. Setting a coordinate causes the joint transform $\mathbf{T}_{CD}$ to change. To accommodate this, the system adjusts the poses of one or both bodies connected to the joint, along with adjacent bodies connected to them, with preference given to bodies that are not attached to "ground". However, if this is done during simulation, and particularly if one or both of the bodies connected to the joint are moving dynamically, the results will be unpredictable and will likely conflict with the simulation.

Joint coordinates are also often exported as properties. For example, the HingeJoint class (Section 3.4) exports its θ coordinate as the property theta, which can be accessed in the GUI, or via the accessor methods

```
double getTheta()          // get theta in degrees

void setTheta (double deg) // set theta in degrees
```

Since joint constraints are generally nonlinear, their linearized enforcement at the velocity level by (3.8) will usually produce small errors as the simulation proceeds. These errors are reduced using a position correction step described in Section 4.10.1 and [8]. Errors can also be caused by joint compliance (Section 3.3.7). Both effects mean that the joint transform $\mathbf{T}_{CD}$ may deviate from the allowed values dictated by the joint type. In ArtiSynth, this is accounted for by introducing an additional *constraint* frame G between D and C (Figure 3.9). G is computed to be the nearest frame to C that lies exactly in the joint constraint space. $\mathbf{T}_{GD}$ is therefore a valid joint transform, $\mathbf{T}_{GC}$ accommodates the error, and the whole joint transform is given by the composition

$$\mathbf{T}_{CD} = \mathbf{T}_{GD}\,\mathbf{T}_{CG}. \tag{3.12}$$

If there is no compliance or joint error, then frames G and C are identical, $\mathbf{T}_{CG} = \mathbf{I}$, and $\mathbf{T}_{CD} = \mathbf{T}_{GD}$. Because $\mathbf{T}_{CG}$ describes the joint error, we sometimes refer to it as $\mathbf{T}_{err} = \mathbf{T}_{CG}$.

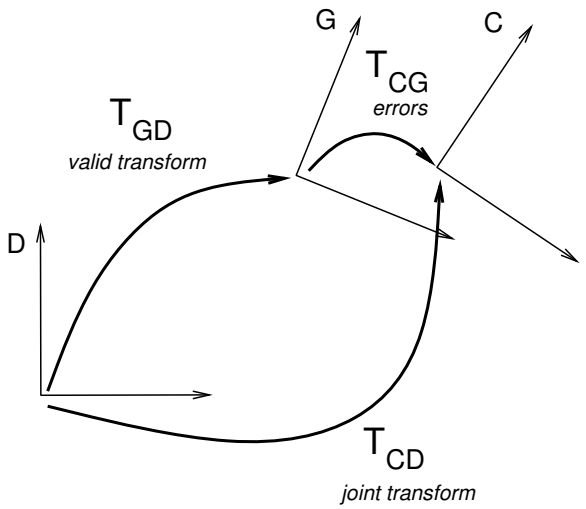


Figure 3.9: 2D schematic showing the joint frames D and C, along with the intermediate frame G that accounts for numeric error and complaint motion.

3.3.3 Creating Joints

Joint and connector components in ArtiSynth are both derived from the superclass BodyConnector, with joints being further derived from JointBase, which provides support for coordinates. Some of the commonly used joints and connectors are described in Section 3.4.

An application creates a joint by constructing it and adding it to a `MechModel`. Many joints have constructors of the form

```
XXXJoint (bodyA, bodyB, TDW)
```

which specifies the bodies A and B which the joint connects, along with the transform $\mathbf{T}_{DW}$ giving the pose of the joint base frame D in world coordinates. The constructor then assumes that the joint is in the zero state, so that C and D are the same and $\mathbf{T}_{CD} = \mathbf{I}$ and $\mathbf{T}_{CW} = \mathbf{T}_{DW}$, and then computes $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ from

$$\mathbf{T}_{CA} = \mathbf{T}_{AW}^{-1}\,\mathbf{T}_{CW} \tag{3.13}$$

$$\mathbf{T}_{DB} = \mathbf{T}_{BW}^{-1}\,\mathbf{T}_{DW}, \tag{3.14}$$

where $\mathbf{T}_{AW}$ and $\mathbf{T}_{BW}$ are the current poses of A and B.

After the joint is created, it should be added to the system's `MechModel` using addBodyConnector(), as shown in the following code fragment:

```
MechModel mech;
RigidBody bodyA, bodyB;
RigidTransform3d TDW;

... initialize mech, bodyA, bodyB, and TDW ...

HingeJoint joint = new HingeJoint (bodyA, bodyB, TDW);
mech.addBodyConnector (joint);
```

It is also possible to create a joint using its default constructor and attach it to the bodies afterward, using the method setBodies(bodyA,bodyB,TDW), as in the following:

```
HingeJoint joint = new HingeJoint ();
joint.setBodies (bodyA, bodyB, TDW);
mech.addBodyConnector (joint);
```

One reason for doing this is that it allows the joint transform $\mathbf{T}_{CD}$ to be modified (by setting coordinate values) *before* `setBodies()` is called; this is discussed further in Section 3.3.4.

Joints usually offer a number of other constructors that let its world location and body relationships to be specified in different ways. These may include:

```
XXXJoint (bodyA, TCA, bodyB, TDB)

XXXJoint (bodyA, bodyB, TCW, TDW)
```

The first, which is restricted to rigid bodies, allows the application to explicitly specify transforms $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ connecting frames C and D to the body frames A and B, and is useful when $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ are explicitly known, or the initial value of $\mathbf{T}_{CD}$ is *not* the identity. Likewise, the second constructor allows $\mathbf{T}_{CW}$ and $\mathbf{T}_{DW}$ to be explicitly specified, with $\mathbf{T}_{CD} \neq \mathbf{I}$ if $\mathbf{T}_{CW} \neq \mathbf{T}_{DW}$. For instance, suppose $\mathbf{T}_{CD}$ and $\mathbf{T}_{DW}$ are both known. Then we can use the relationship

$$\mathbf{T}_{CW} = \mathbf{T}_{DW}\,\mathbf{T}_{CD} \tag{3.15}$$

to create the joint as in the following code fragment:

```
MechModel mech;
RigidBody bodyA, bodyB;
RigidTransform3d TDW, TCD;

... initialize mech, bodyA, bodyB, TDW, and TCD ...

// compute TCW:
RigidTransform3d TCW = new RigidTransform3d();
TCW.mul (TDW, TCD);
HingeJoint joint = new HingeJoint (bodyA, bodyB, TCW, TDW);
mech.addBodyConnector (joint);
```

As an alternative to specifying $\mathbf{T}_{DW}$ or its equivalents, some joint types provide constructors that let the application locate specific joint features. These may be easier to use in some cases. For instance, HingeJoint provides a constructor

```
HingeJoint (bodyA, bodyB, originD, zaxis)
```

that specifies origin of D and its z axis (which is the rotation axis), with the remaining orientation of D aligned as closely as possible with the world. SphericalJoint provides a constructor

```
SphericalJoint (bodyA, bodyB, originD)
```

that specifies origin of D and aligns its orientation with the world. Users should consult the source code or API documentation for specific joints to see what special constructors may be available.

Finally, it is possible to use joints to connect a single body to ground (by convention, this is the A body). Most joints provide a constructor of the form

```
XXXJoint (bodyA, TDW)
```

which allows this to be done explicitly. Alternatively, most joint constructors which supply body B will allow this to be specified as `null`, so that body A will be connected to ground by default.

3.3.4 Working with coordinates

As mentioned in Section 3.3.2, some joints support coordinates that parameterize the valid motions within the joint transform $\mathbf{T}_{CD}$. All such joints are subclasses of JointBase, which provides some generic methods for querying and setting coordinate values (`JointBase` is in turn a subclass of BodyConnector).

The number of coordinates is returned by the method `numCoordinates()`; if this returns 0, then coordinates are not supported. Each coordinate has an index in the range $0 \ldots M-1$, where M is the number of coordinates. Coordinate values can be queried or set using the following methods:

```
getCoordinate (int idx)               // get coordinate value with index idx
getCoordinates (VectorNd coords)      // get all coordinates values

setCoordinate (int idx, double value) // set coordinate value with index idx
setCoordinates (VectorNd values)      // set all coordinates values
```

Similarly, there are methods for managing coordinate ranges:

```
// get/set range information for coordinate with index idx
DoubleInterval getCoordinateRange (int idx)
double getMinCoordinate (int idx)
double getMaxCoordinate (int idx)
void setCoordinateRange (idx, DoubleInterval rng)

// get/set range information in degrees for coordinate with index idx
DoubleInterval getCoordinateRangeDeg (int idx)
double getMinCoordinateDeg (int idx)
double getMaxCoordinateDeg (int idx)
void setCoordinateRangeDeg (idx, DoubleInterval rng)
```

Range checking can be disabled by setting the range to $(-\text{inf}, \text{inf})$, or by specifying `rng` as `null`, which implicitly does the same thing.

Specific joint types usually also provide names for their joint coordinates, along with integer constants describing their indices and methods for managing their ranges and values. For example, CylindricalJoint supports two coordinates, z and θ, along with the following:

```
// coordinate indices
static final int Z_IDX = 0;
static final int THETA_IDX = 1;

// set/get z value and range
double getZ()
void setZ (double z)
DoubleInterval getZRange()
void setZRange (double min, double max)

// set/get theta value and range in degrees
double getTheta()
void setTheta (double theta)
DoubleInterval getThetaRange()
void setThetaRange (double min, double max)
void setThetaRange (DoubleInterval rng)
```

The coordinate value and range information is also exported as the properties z, theta, zRange and thetaRange, allowing them to be set in the GUI. For convenience, particularly in GUI applications, the properties and methods controlling the value and range of angular coordinates generally use degrees instead of radians.

As discussed in Section 3.3.2, unlike in some multibody simulation systems (such as OpenSim), joint coordinates are *not* fundamental quantities that describe system state. As such, then, coordinates can usually only be set in specific circumstances that avoid simulation conflicts. In general, when joint coordinates are set, the system adjusts the poses of one or both bodies connected to this joint, along with adjacent bodies connected to them, with preference given to bodies that are not attached to "ground". However, if this is done during simulation, and particularly if one or both of the bodies connected to the joint are moving dynamically, the results will be unpredictable and will likely conflict with the simulation.

If a joint has been created with its default constructor and not yet attached to any bodies, then setting joint values will simply set the joint transform $\mathbf{T}_{CD}$. This can be useful in situations where one needs to initialize a joint's $\mathbf{T}_{CD}$ to a non-identity value corresponding to a particular set of joint coordinates:

```
RigidTransform3d TDW; // known location for D frame
double z, theta; // desired initial coordinate values
...
CylindricalJoint joint = new CylindricalJoint();
joint.setZ (z);
joint.setTheta (thetaDeg);
joint.setBodies (bodyA, bodyB, TDW);
```

This can also be done in vector form:

```
RigidTransform3d TDW; // known location for D frame
VectorNd coordValues; // desired initial coordinate values
...
CylindricalJoint joint = new CylindricalJoint();
joint.setCoordinates (coordValues);
joint.setBodies (bodyA, bodyB, TDW);
```

In either of these cases, `setBodies()` will not use $\mathbf{T}_{CD} = \mathbf{I}$ but instead use the value determined by the initial coordinate values.

To determine the $\mathbf{T}_{CD}$ corresponding to a particular set of coordinates, one may use the method

```
void coordinatesToTCD (RigidTransform3d TCD, VectorNd coords)
```

In some cases, within a model's `build()` method, one may wish to set initial coordinates *after* a joint has been attached to its bodies, in order to move those bodies (along with the bodies attached to them) into an initial configuration without having to explicitly calculate the poses from the joint coordinates. As mentioned above, the system will make a decision about which attached bodies are most "free" and adjust their poses accordingly. This is done in the example of the next section.

3.3.5 Example: a simple hinge joint

A simple model showing two rigid bodies connected by a joint is defined in

`artisynth.demos.tutorial.RigidBodyJoint`

The build method for this model is given below:

```
1   public void build (String[] args) {
2
3      // create MechModel and add to RootModel
4      mech = new MechModel ("mech");
5      mech.setGravity (0, 0, -98);
6      mech.setFrameDamping (1.0);
7      mech.setRotaryDamping (4.0);
8      addModel (mech);
9
10     PolygonalMesh mesh;  // bodies will be defined using a mesh
11
12     // create first body and set its pose
13     mesh = MeshFactory.createRoundedBox (lenx1, leny1, lenz1, /*nslices=*/8);
```

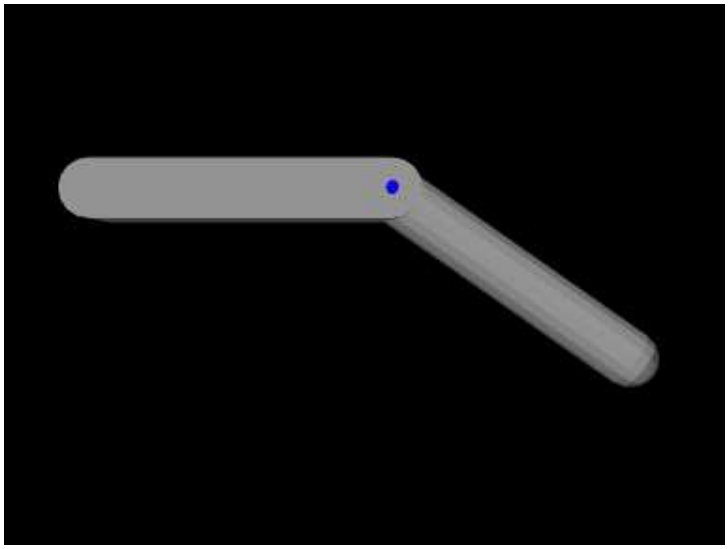


Figure 3.10: RigidBodyJoint model loaded into ArtiSynth.

```
14        RigidTransform3d TMB =
15           new RigidTransform3d (0, 0, 0, /*axisAng=*/1, 1, 1, 2*Math.PI/3);
16        mesh.transform (TMB);
17        bodyB = RigidBody.createFromMesh ("bodyB", mesh, /*density=*/0.2, 1.0);
18        bodyB.setPose (new RigidTransform3d (0, 0, 1.5*lenx1, 1, 0, 0, Math.PI/2));
19        bodyB.setDynamic (false);
20
21        // create second body and set its pose
22        mesh = MeshFactory.createRoundedCylinder (
23           leny2/2, lenx2, /*nslices=*/16, /*nsegs=*/1, /*flatBottom=*/false);
24        mesh.transform (TMB);
25        bodyA = RigidBody.createFromMesh ("bodyA", mesh, 0.2, 1.0);
26        bodyA.setPose (new RigidTransform3d (
27                          (lenx1+lenx2)/2, 0, 1.5*lenx1, 1, 0, 0, Math.PI/2));
28
29        // create the joint
30        RigidTransform3d TDW =
31           new RigidTransform3d (lenx1/2, 0, 1.5*lenx1, 1, 0, 0, Math.PI/2);
32        HingeJoint joint = new HingeJoint (bodyA, bodyB, TDW);
33
34        // add components to the mech model
35        mech.addRigidBody (bodyB);
36        mech.addRigidBody (bodyA);
37        mech.addBodyConnector (joint);
38
39        joint.setTheta (35);  // set joint position
40
41        // set render properties for components
42        RenderProps.setFaceColor (joint, Color.BLUE);
43        joint.setShaftLength (4);
44        joint.setShaftRadius (0.2);
45     }
```

A `MechModel` is created as usual at line 4. However, in this example, we also set some parameters for it: setGravity() is used to set the gravity acceleration vector to $(0,0,-98)^T$ instead of the default value of $(0,0,-9.8)^T$, and the `frameDamping` and `rotaryDamping` properties (Section 3.2.6) are set to provide appropriate damping.

Each of the two rigid bodies are created from a mesh and a density. The meshes themselves are created using the factory methods MeshFactory.createRoundedBox() and MeshFactory.createRoundedCylinder() (lines 13 and 22), and then RigidBody.createFromMesh() is used to turn these into rigid bodies with a density of 0.2 (lines 17 and 25). The pose of the two bodies is set using `RigidTransform3d` objects created with x, y, z translation and axis-angle orientation values

(lines 18 and 26).

The hinge joint is implemented using HingeJoint, which is constructed at line 32 with the joint coordinate frame D being located in world coordinates by `TDW` as described in Section 3.3.3.

Once the joint is created and added to the `MechModel`, the method setTheta() is used to explicitly set the joint parameter to 35 degrees. The joint transform $\mathbf{T}_{CD}$ is then set appropriately and `bodyA` is moved to accommodate this (`bodyA` being chosen since it is the most free to move).

Finally, joint rendering properties are set starting at line 42. We render the joint as a cylindrical shaft about the rotation axis, using its shaftLength and shaftRadius properties. Joint rendering is discussed in more detail in Section 3.3.9).

To run this example in ArtiSynth, select All demos > tutorial > RigidBodyJoint from the Models menu. The model should load and initially appear as in Figure 3.10. Running the model (Section 1.5.3) will cause `bodyA` to fall and swing under gravity.

3.3.6 Constraint forces

During each simulation solve step, the joint velocity constraints described by (3.9) and (3.10) are enforced by bilateral and unilateral constraint forces $\mathbf{f}_g$ and $\mathbf{f}_n$:

$$\mathbf{f}_g = \mathbf{G}_\mathbf{J}^T \lambda_J, \quad \mathbf{f}_n = \mathbf{N}_\mathbf{J}^T \theta_J. \tag{3.16}$$

Here, $\mathbf{f}_g$ and $\mathbf{f}_n$ are spatial forces (or *wrenches*, Section A.5) acting in the joint coordinate frame C, and λ_J and θ_J are the Lagrange multipliers computed as part of the mechanical system solve (see (1.6) and (1.8)). The sizes of λ_J and θ_J equal the number of bilateral and *engaged* unilateral constraints in the joint; these numbers can be queried for a particular joint using the methods numBilateralConstraints() and numEngagedUnilateralConstraints(). (The number of engaged unilateral constraints may be less than the total number of unilateral constraints; the latter may be queried with numUnilateralConstraints(), while the total number of constraints is returned by numConstraints().

Applications may sometimes need to query the current constraint force values, typically from within a controller or monitor (Section 5.3). The Lagrange multipliers themselves may be obtained with

```
void getBilateralForces (VectorNd lam)

void getUnilateralForces (VectorNd the)
```

which load the multipliers into `lam` or `the` and set their sizes to the number of bilateral or engaged unilateral constraints. Alternatively, one can retrieve the individual multiplier for the constraint indexed by `idx` using

```
double getConstraintForce (int idx);
```

Typically, it is more useful to find the spatial constraint forces $\mathbf{f}_g$ and $\mathbf{f}_n$, which can be obtained with respect to frame C:

```
// place the forces in the wrench f
void getBilateralForcesInC (Wrench f)
void getUnilateralForcesInC (Wrench f)

// convenience methods that allocate the wrench and return it
Wrench getBilateralForcesInC();
Wrench getUnilateralForcesInC();
```

If the attached bodies A and B are rigid bodies, it is also possible to obtain the constraint wrenches experienced by those bodies:

```
// place the forces in the wrench f
void getBilateralForcesInA (Wrench f)
void getUnilateralForcesInA (Wrench f)
void getBilateralForcesInB (Wrench f)
void getUnilateralForcesInB (Wrench f)

// convenience methods that allocate the wrench and return it
Wrench getBilateralForcesInA();
Wrench getUnilateralForcesInA();
Wrench getBilateralForcesInB();
Wrench getUnilateralForcesInB();
```

Constraint wrenches obtained for bodies A or B are given in world coordinates, which is consistent with the forces reported by rigid bodies via their `getForce()` method. To orient the forces into body coordinates, one may use the inverse of the rotation matrix $\mathbf{R}$ of the body's pose. For example:

```
RigidBody bodyA;

// ... body A initialized, etc. ...

Wrench force = joint.getBilateralForceInA();
force.inverseTransform (bodyA.getPose().R);
```

3.3.7 Compliance and regularization

By default, the constraints used to implement joints and couplings are treated as *hard*, so that the system tries to respect the constraint conditions (3.8) as exactly as possible as the simulation proceeds. Sometimes, however, it is desirable to introduce some "softness" into the constraints, whereby constraint forces are determined as a linear function of their distance from the constraint. Adding compliance also allows an application to *regularize* a system of joint constraints that would otherwise be overconstrained, as illustrated in Section 3.3.8.

To describe compliance precisely, consider the bilateral constraint portion of the MLCP in (1.6), which solves for the updated system velocities $\mathbf{u}^{k+1}$ at each time step:

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^T \\ \mathbf{G} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}^{k+1} \\ \tilde{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{M}\mathbf{u}^k - h\hat{\mathbf{f}}^k \\ 0 \end{pmatrix}. \tag{3.17}$$

Here $\mathbf{G}$ is the system's bilateral constraint matrix, $\tilde{\lambda}$ denotes the constraint impulses (from which the constraint forces λ can be determined by $\lambda = \tilde{\lambda}/h$), and for simplicity we have assumed that $\mathbf{G}$ is constant and so the $\mathbf{g}$ term on the lower right side is 0.

Solving (3.17) results in constraint forces that satisfy $\mathbf{G}\mathbf{u}^{k+1} = 0$ precisely, corresponding to hard constraints. To implement soft constraints, start by defining a function $\phi(\mathbf{q})$ that defines the *distances* from each constraint, where $\mathbf{q}$ is the vector of system positions; these distances are the local translational and rotational deviations from each constraint's correct position and are discussed in more detail in Section 4.10.1. Then assume that the constraint forces are a linear function of these distances:

$$\lambda = -\mathbf{C}^{-1}\phi(\mathbf{q}), \tag{3.18}$$

where $\mathbf{C}$ is a diagonal *compliance* matrix that is equivalent to an inverse stiffness matrix. We also note that ϕ will be time varying, and that we can approximate its change between time steps as

$$\phi^{k+1} \approx \phi^k + h\dot{\phi}^{k+1}, \quad \text{with} \quad \dot{\phi}^{k+1} = \mathbf{G}\mathbf{u}^{k+1}. \tag{3.19}$$

Next, assume that in using (3.18) to determine λ for a particular time step, we use the *average* value of ϕ over the step, represented by $\bar{\phi} = (\phi^{k+1} + \phi^k)/2$. Substituting this and (3.19) into (3.18), multiplying by $\mathbf{C}$, and rearranging yields:

$$\mathbf{G}\mathbf{u}^{k+1} + \frac{2\mathbf{C}}{h}\lambda = -\frac{2}{h}\phi^k. \tag{3.20}$$

Then noting that $\tilde{\lambda} = h\lambda$, we obtain a revised form of (3.17),

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^T \\ \mathbf{G} & 2\mathbf{C}/h^2 \end{pmatrix} \begin{pmatrix} \mathbf{u}^{k+1} \\ \tilde{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{M}\mathbf{u}^k - h\hat{\mathbf{f}}^k \\ -2\phi^k/h \end{pmatrix}, \tag{3.21}$$

in the which the zeros in the matrix and right hand side have been replaced by compliance terms. The resulting constraint behavior is different from that of (3.17) in two important ways:

1. The joint now allows 6 DOF, with motion along the constrained directions limited by restoring spring constants given by the reciprocals of the diagonal entries of $\mathbf{C}$.
2. If $\mathbf{C}$ has no zero diagonal entries, then the system (3.21) is *regularized* by the $2\mathbf{C}/h^2$ term in the lower right matrix block. This means that the matrix is always non-singular, even if $\mathbf{G}$ is rank deficient, and so compliance offers a way to handle overconstrained models, as discussed further in Section 3.3.8.

Unilateral constraints can be regularized using the same approach, with a distance function defined such that $\phi(\mathbf{q}) \leq 0$.

The reason for specifying soft constraints using compliance instead of stiffness is that by setting $\mathbf{C} = 0$ we can easily handle the case of *infinite* stiffness where the constraints are strictly enforced. The ArtiSynth compliance implementation uses a slightly more complex version of (3.21) that accounts for non-constant $\mathbf{G}$ and also allows for a damping term $-\mathbf{D}\dot{\phi}$, where $\mathbf{D}$ is again a diagonal matrix. For more details, see [6] and [16].

When using compliance, damping is often needed for stability, and, in the case of unilateral constraints, to prevent "bouncing". A good choice for damping is usually *critical damping*, which is discussed further below.

Any joint which is a subclass of BodyConnector allows individual compliance values C_i and damping values D_i to be set for each of the joint's i constraints. These values comprise the diagonal entries in the compliance and damping matrices $\mathbf{C}$ and $\mathbf{D}$, and can be queried and set using the methods

```
VectorNd getCompliance()
void setCompliance (VectorNd compliance)

VectorNd getDamping()
void setCompliance (VectorNd damping)
```

The vectors supplied to the above `set` methods contain the requested compliance or damping values. If their size n is less than `numConstraints()`, then compliance or damping will be set for the first n constraints. Damping for a specific constraint only has an effect if the compliance for that constraint is nonzero.

What compliance and damping values should be specified? Compliance is usually relatively easy to figure out. Each of the joint's individual constraints i corresponds to a row in its bilateral constraint matrix $\mathbf{G}_\mathrm{J}$ or unilateral constraint matrix $\mathbf{N}_\mathrm{J}$, and represents a specific 6 DOF direction along which the spatial velocity $\hat{\mathbf{v}}_{CD}$ (of frame C with respect to D) is restricted (more details on this are given in Section 4.10.1). Each of these constraint directions is usually predominantly linear or rotational; specific descriptions for the constraints of different joints are provided in Section 3.4. To determine compliance for a constraint i, estimate the typical force f likely to act along its direction, decide how much displacement δq (translational or rotational) along that constraint is desirable, and then set the compliance C_i to the associated inverse stiffness:

$$C_k = \delta q / f. \tag{3.22}$$

Once C_k is determined, the damping D_k can be estimated based on the desired damping ratio ζ, using the formula

$$D_k = 2\zeta\sqrt{M/C_k} \tag{3.23}$$

where M is total mass of the bodies attached to the joint. Typically, the desired damping will be close to critical damping, for which $\zeta = 1$.

Constraints associated with linear motion will typically require different compliance values from those associated with rotation. To make this process easier, joint components allow the setting of collective compliance values for their linear and rotary constraints, using the methods

```
void setLinearCompliance (double c)
double getLinearCompliance()

void setRotaryCompliance (double c)
double getRotaryCompliance()
```

The `set()` methods will set a uniform compliance for all linear or rotary constraints, except for unilateral constraints associated with coordinate limits. At the same time, they will also set an *automatically* computed critical damping value. Likewise, the `get()` methods query these linear or rotary constraints for uniform compliance values (with the corresponding critical damping), and return either that value, or -1 if it does not exist.

Most of the demonstration models for the joints described in Section 3.4 allow these linear and rotary compliance settings to be adjusted interactively using a control panel, enabling users to experimentally gain a feel for their behavior.

To determine programmatically whether a particular constraint is linear or rotary, one can use the joint method

```
VectorNi getConstraintFlags()
```

which returns a vector of information flags for all its constraints. Linear and rotary constraints are indicated by the flags `LINEAR` and `ROTARY`, defined in RigidBodyConstraint.

3.3.8 Example: an overconstrained linkage

Situations may occasionally arise in which a model is *overconstrained*, which means that the rows of the bilateral constraint matrix **G** in (3.8) are not all linearly dependent, or in other words, **G** does not have *full row rank*. At present, the ArtiSynth solver has difficultly handling overconstrained models, but these situations can often be handled by adding a small amount of compliance to the constraints. (Overconstraining is not a problem with unilateral constraints **N**, because of the way they are handled by the solver.)

One possible symptom of an overconstrained system is a error message in the application's terminal output, such as

```
Pardiso: num perturbed pivots=12
```

Overconstraining frequently occurs in closed-chain linkages, involving loops in which a jointed sequence of links is connected back on itself. Depending on how the constraints are configured and how redundant they are, the system may still be able to move. A classical example is the *four-bar linkage*, a common version of which consists of four links, or "bars", arranged as a parallelogram and connected by hinge joints at the corners. One link is usually connected to ground, and so the remaining three links together have 18 DOF, while the four hinge joints together remove 20 DOF, overconstraining the system. However, the constraints are redundant in such as way that the linkage still actually has 1 DOF.

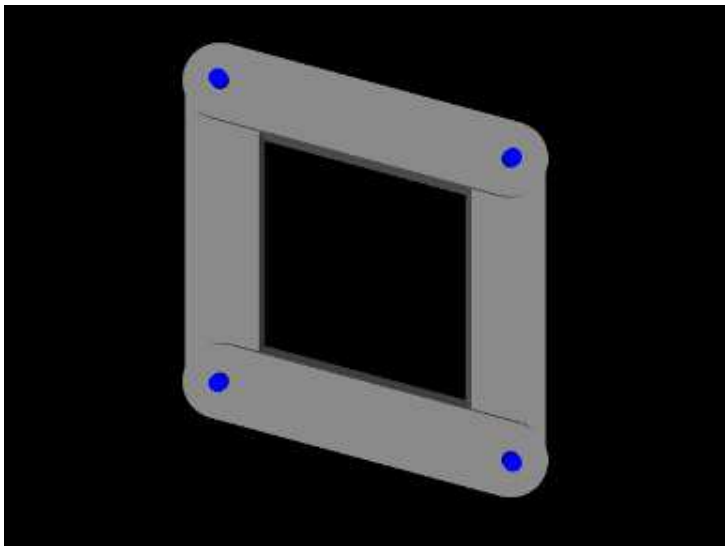


Figure 3.11: FourBarLinkage model, several steps into the simulation.

To model a four-bar in ArtiSynth presently requires adding compliance to the hinge joints. An example of this is defined by the demo program

```
artisynth.demos.tutorial.FourBarLinkage
```

shown in Figure 3.11. The code for the `build()` method and a couple of supporting methods is given below:

```
   /**
    * Create a link with a length of 1.0, width of 0.25, and specified depth
    * and add it to the mech model. The parameters x, z, and deg specify the
    * link's position and orientation (in degrees) in the x-z plane.
    */
   protected RigidBody createLink (
      MechModel mech, String name,
      double depth, double x, double z, double deg) {
      int nslices = 20; // num slices on the rounded mesh ends
      PolygonalMesh mesh =
         MeshFactory.createRoundedBox (1.0, 0.25, depth, nslices);
      RigidBody body = RigidBody.createFromMesh (
         name, mesh, /*density=*/1000.0, /*scale=*/1.0);
      body.setPose (new RigidTransform3d (x, 0, z, 0, Math.toRadians(deg), 0));
```

```
15        mech.addRigidBody (body);
16        return body;
17     }
18
19     /**
20      * Create a hinge joint connecting one end of link0 with the other end of
21      * link1, and add it to the mech model.
22      */
23     protected HingeJoint createJoint (
24        MechModel mech, String name, RigidBody link0, RigidBody link1) {
25        // easier to locate the link using TCA and TDB since we know where frames
26        // C and D are with respect the link0 and link1
27        RigidTransform3d TCA = new RigidTransform3d (0, 0,  0.5, 0, 0, Math.PI/2);
28        RigidTransform3d TDB = new RigidTransform3d (0, 0, -0.5, 0, 0, Math.PI/2);
29        HingeJoint joint = new HingeJoint (link0, TCA, link1, TDB);
30        joint.setName (name);
31        mech.addBodyConnector (joint);
32        // set joint render properties
33        joint.setAxisLength (0.4);
34        RenderProps.setLineRadius (joint, 0.03);
35        return joint;
36     }
37
38     public void build (String[] args) {
39        // create a mech model and set rigid body damping parameters
40        MechModel mech = new MechModel ("mech");
41        addModel (mech);
42        mech.setFrameDamping (1.0);
43        mech.setRotaryDamping (4.0);
44
45        // create four 'bars' from which to construct the linkage
46        RigidBody[] bars = new RigidBody[4];
47        bars[0] = createLink (mech, "link0", 0.2, -0.5,  0.0, 0);
48        bars[1] = createLink (mech, "link1", 0.3,  0.0,  0.5, 90);
49        bars[2] = createLink (mech, "link2", 0.2,  0.5,  0.0, 180);
50        bars[3] = createLink (mech, "link3", 0.3,  0.0, -0.5, 270);
51        // ground the left bar
52        bars[0].setDynamic (false);
53
54        // connect the bars using four hinge joints
55        HingeJoint[] joints = new HingeJoint[4];
56        joints[0] = createJoint (mech, "joint0", bars[0], bars[1]);
57        joints[1] = createJoint (mech, "joint1", bars[1], bars[2]);
58        joints[2] = createJoint (mech, "joint2", bars[2], bars[3]);
59        joints[3] = createJoint (mech, "joint3", bars[3], bars[0]);
60
61        // Set uniform compliance and damping for all bilateral constraints,
62        // which are the first 5 constraints of each joint
63        VectorNd compliance = new VectorNd(5);
64        VectorNd damping = new VectorNd(5);
65        for (int i=0; i<5; i++) {
66           compliance.set (i, 0.000001);
67           damping.set (i, 25000);
68        }
69        for (int i=0; i<joints.length; i++) {
70           joints[i].setCompliance (compliance);
71           joints[i].setDamping (damping);
72        }
73     }
```

Two helper methods are used to construct the model: `createLink()` (lines 6-17), and `createJoint()` (lines 23-36). `createLink()` makes the individual rigid bodies used to build the linkage: a mesh is produced defining the body's shape (a box with rounded ends), and then passed to the RigidBody `createFromMesh()` method which creates the body and sets its inertia according to a specified density. The body's pose is then set so as to center it at $(x, 0, z)$ while

rotating it about the y axis by the angle `deg` (in degrees). The completed body is then added to the `MechModel mech` and returned.

The second helper method, `createJoint()`, connects two rigid bodies (`link0` and `link1`) together using a `HingeJoint`. Because we know the location of the joint in body-relative coordinates, it is easier to create the joint using the transforms $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ instead of $\mathbf{T}_{DW}$: $\mathbf{T}_{CA}$ locates the joint at the top end of `link0`, at $(0,0,0.5)$, with the z axis parallel to the body's y axis, while $\mathbf{T}_{DB}$ similarly locates the joint at the bottom of `link1`. After the joint is created and added to the `MechModel`, its render properties are set so that its axis drawn as a blue cylinder.

The `build()` method itself begins by creating a `MechModel` and setting damping parameters for the rigid bodies (lines 40-43). Next, `createLink()` is used to create and store the four links (lines 46-50), and the left bar is attached to ground by making it non-dynamic (line 52). The links are then connected together using joints created by `createJoint()` (lines 55-59). Finally, uniform compliance and damping values are set for each of the joint's bilateral constraints, using the `setCompliance()` and `setDamping()` methods (lines 63-72). Values are set for the first five constraints, since for a `HingeJoint` these are the bilateral constraints. The compliance value of $C = 10^{-6}$ was found experimentally to be low enough so as to not cause noticeable deflections in the joints. Given C and an average mass of around $M = 150$ for each link pair, (3.23) suggests the damping factor of $D = 25000$. Note that for this example, very similar settings could be achieved by simply calling

```
for (int i=0; i<joints.length; i++) {
   joints[i].setLinearCompliance (0.000001);
   joints[i].setRotaryCompliance (0.000001);
}
```

In principle, we only need to set compliance for the constraints that are redundant, but it can sometimes be difficult to determine exactly which these are. Also, different values are often needed for linear and rotary constraints; that is not necessary here because the links have unit length and so the linear and rotary units have similar scales.

3.3.9 Rendering joints

Most joints provide a means to render themselves in order to provide a graphical representation of their position and configuration. Control over this is achieved by setting various properties in the joint component, including both specialized properties and the standard render properties (Section 4.3) used by all renderable components.

All joints which are subclasses of JointBase support rendering of both their C and D coordinate frames, through the properties drawFrameC, drawFrameD, and axisLength. The first two properties are of the type Renderer.AxisDrawStyle (described in detail in Section 3.2.7), and can be set to `LINE` or `ARROW` to enable the coordinate axes to be drawn either as lines or solid arrows. The axisLength property has type `double` and specifies the length with which the axes are drawn. As with all properties, these properties can be set either in the GUI, or in code using accessor methods supplied by the joint:

```
void setAxisLength (double l)
double getAxisLength()

void setDrawFrameC (AxisDrawStyle style)
(AxisDrawStyle getDrawFrameC()

void setDrawFrameD (AxisDrawStyle style)
(AxisDrawStyle getDrawFrameD()
```

Another pair of properties used by several joints is shaftLength and shaftRadius, which specify the length and radius used to draw shaft or axis structures associated with the joint. These are rendered as solid cylinders, using the color indicated by the faceColor rendering property. The default value of both properties is 0; if shaftLength is 0, then the structures are not drawn, while if shaftRadius is 0, a default value proportional to shaftLength is used. For example, to enable rendering of a blue shaft along the rotation axis of a hinge joint, one may use the code fragment

```
HingeJoint joint;
...

joint.setShaftLength (0.5); // set shaft dimensions
joint.setShaftRadius (0.05);
RenderProps.setFaceColor (joint, Color.BLUE); // set the color
```

As another example, to enable rendering of a green ball about the center of a spherical joint, one may use the fragment

```
SphericalJoint joint;
...

joint.setJointRadius (0.02); // set the ball size
RenderProps.setFaceColor (joint, Color.GREEN); // set the color
```

Specific joints may define additional properties to control how they are rendered.

3.4 Joint components

ArtiSynth supplies a number of basic joints and connectors in the package `artisynth.core.mechmodels`, the most common of which are described here.

Many of the descriptions are associated with a demonstration model, named `XXXJointDemo`, where `XXX` is the joint type. These demos are located in the package `artisynth.demos.mech`, and can be loaded by selecting All demos > mech > XXXJointDemo from the Models menu. When run, they can be interactively controlled, using either the pull tool (see the section "Pull Manipulation" in the ArtiSynth User Interface Guide), or the interactive control panel. The control panel allows the adjustment of coordinate values and ranges (if supported), some of the render properties, and the different compliance and damping properties (Section 3.3.7). One can inspect the source code for each demo in its `.java` file located in the folder `<ARTISYNTH_HOME>/src/artisynth/demos/mech`.

3.4.1 Hinge Joint

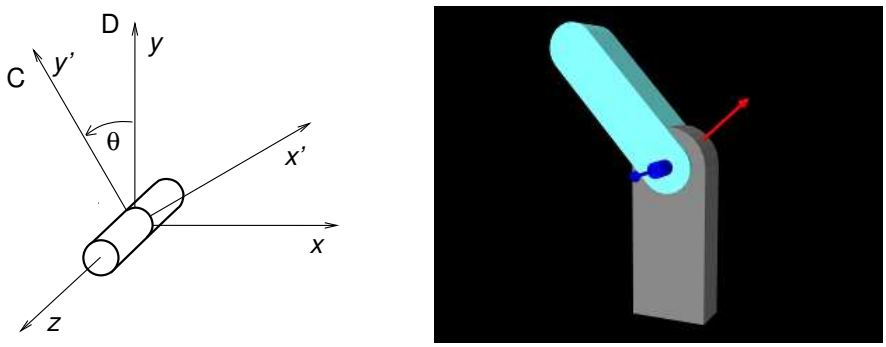


Figure 3.12: Coordinate frames (left) and demo model (right) for the hinge joint.

The HingeJoint (Figure 3.12) is a 1 DOF joint that constrains motion between frames C and D to a simple rotation about the z axis of D. It implements six constraints and one coordinate θ (Table 3.1), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The value and ranges for θ are exported by the properties theta and thetaRange, and the θ coordinate index is defined by the constant `THETA_IDX`. For rendering, the properties shaftLength and shaftRadius control the size of a shaft drawn about the rotation axis, using the faceColor rendering property. A demo is provided by `artisynth.demos.mech.HingeJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
HingeJoint (bodyA, bodyB, originD, zaxis)
```

creates a hinge joint with a specified origin and z axis direction for frame D (in world coordinates), and frames C and D coincident.

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts translation along z
3	bilateral	restricts rotation about x
4	bilateral	restricts rotation about y
5	unilataral	enforces limits on θ
0	θ	counter-clockwise rotation of C about the z axis

Table 3.1: Constraints (top) and coordinates (bottom) for the hinge joint.

3.4.2 Slider Joint

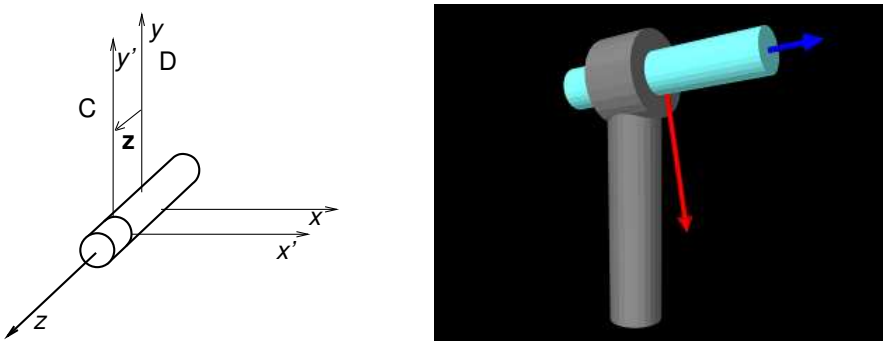


Figure 3.13: Coordinate frames (left) and demo model (right) for the slider joint.

The SliderJoint (Figure 3.13) is a 1 DOF joint that constrains motion between frames C and D to a simple translation along the z axis of D. It implements six constraints and one coordinate z (Table 3.2), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The value and ranges for z are exported by the properties z and zRange, and the z coordinate index is defined by the constant `Z_IDX`. For rendering, the properties shaftLength and shaftRadius control the size of a shaft drawn about the sliding axis, using the faceColor rendering property. A demo is provided by `artisynth.demos.mech.SliderJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
SliderJoint (bodyA, bodyB, originD, zaxis)
```

creates a slider joint with a specified origin and z axis direction for frame D (in world coordinates), and frames C and D coincident.

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts rotation about x
3	bilateral	restricts rotation about y
4	bilateral	restricts rotation about z
5	unilataral	enforces limits on the z coordinate
0	z	translation of C along the z axis

Table 3.2: Constraints (top) and coordinates (bottom) for the slider joint.

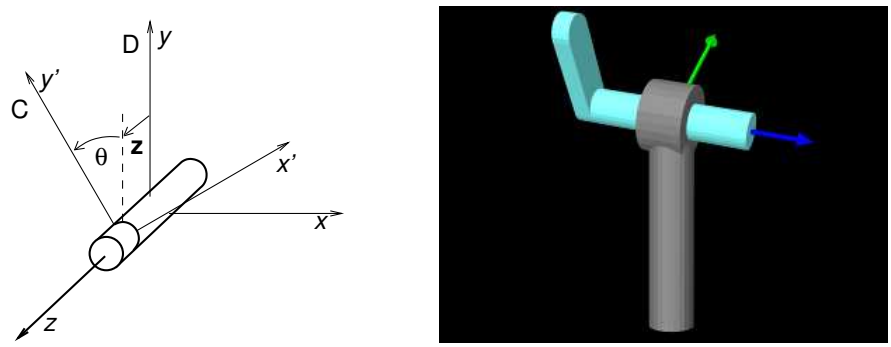


Figure 3.14: Coordinate frames (left) and demo model (right) for the cylindrical joint.

3.4.3 Cylindrical Joint

The CylindricalJoint (Figure 3.14) is a 2 DOF joint that constrains motion between frames C and D to translation and rotation along and about the z axis of D. It implements six constraints and two coordinates z and θ (Table 3.3), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The value and ranges for z and θ are exported by the properties z, theta, zRange and thetaRange, and the coordinate indices are defined by the constants `Z_IDX` and `THETA_IDX`. For rendering, the properties shaftLength and shaftRadius control the size of a shaft drawn about the sliding/rotation axis, using the faceColor rendering property. A demo is provided by `artisynth.demos.mech.CylindricalJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
CylindricalJoint (bodyA, bodyB, originD, zaxis)
```

creates a cylindrical joint with a specified origin and z axis direction for frame D (in world coordinates), and frames C and D coincident.

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts rotation about x
3	bilateral	restricts rotation about y
4	unilataral	enforces limits on the z coordinate
5	unilataral	enforces limits on the θ coordinate
0	z	translation of C along the z axis
1	θ	rotation of C about the z axis

Table 3.3: Constraints (top) and coordinates (bottom) for the cylindrical joint.

3.4.4 Slotted Hinge Joint

The SlottedHingeJoint (Figure 3.15) is a 2 DOF joint that constrains motion between frames C and D to translation along the x axis and rotation about the z axis of D. It implements six constraints and two coordinates x and θ (Table 3.4), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & x \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.24}$$

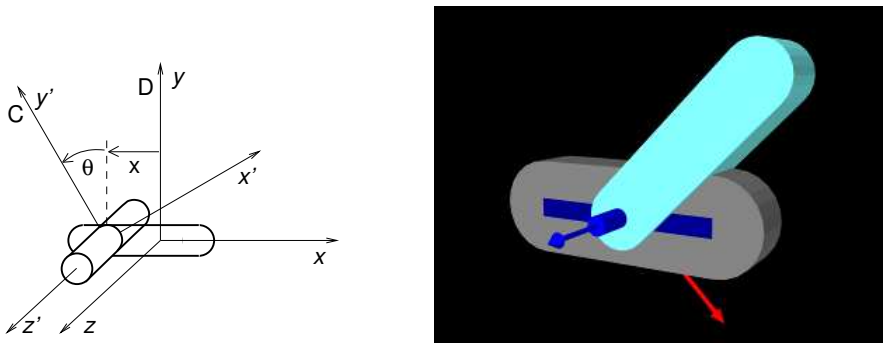


Figure 3.15: Coordinate frames (left) and demo model (right) for the slotted hinge joint.

The value and ranges for x and θ are exported by the properties x, theta, xRange and thetaRange, and the coordinate indices are defined by the constants `X_IDX` and `THETA_IDX`. For rendering, the properties shaftLength and shaftRadius control the size of a shaft drawn about the rotation axis, while slotWidth and slotDepth control the width and depth of a slot drawn along the sliding (x) axis; both are drawn using the faceColor rendering property. When rendering the slot, its bounds along the x axis are set to xRange by default. However, this may be too large, particularly if xRange is unbounded. As an alternate, the property slotRange will be used instead if its range (i.e., the upper bound minus the lower bound) exceeds 0. A demo of `SlottedHingeJoint` is provided by `artisynth.demos.mech.SlottedHingeJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
SlottedHingeJoint (bodyA, bodyB, originD, zaxis)
```

creates a slotted hinge joint with a specified origin and z axis direction for frame D (in world coordinates), and frames C and D coincident.

Index	type/name	description
0	bilateral	restricts translation along y
1	bilateral	restricts translation along z
2	bilateral	restricts rotation about x
3	bilateral	restricts rotation about y
4	unilataral	enforces limits on the x coordinate
5	unilataral	enforces limits on the θ coordinate
0	x	translation of C along the x axis
1	θ	rotation of C about the z axis

Table 3.4: Constraints (top) and coordinates (bottom) for the slotted hinge joint.

3.4.5 Universal Joint

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts translation along z
3	bilateral	restricts rotation about the final x axis of C
4	unilataral	enforces limits on the roll coordinate
5	unilataral	enforces limits on the pitch coordinate
0	θ (roll)	first rotation of C about the z axis of D
1	ϕ (pitch)	second rotation of C about the rotated y' axis

Table 3.5: Constraints (top) and coordinates (bottom) for the universal joint.

The UniversalJoint (Figure 3.16) is a 2 DOF joint that allows C two rotational degrees of freedom with respect to D: a *roll* rotation θ about D's z axis, followed by a *pitch* rotation ϕ about the rotated y' axis. It implements six constraints and

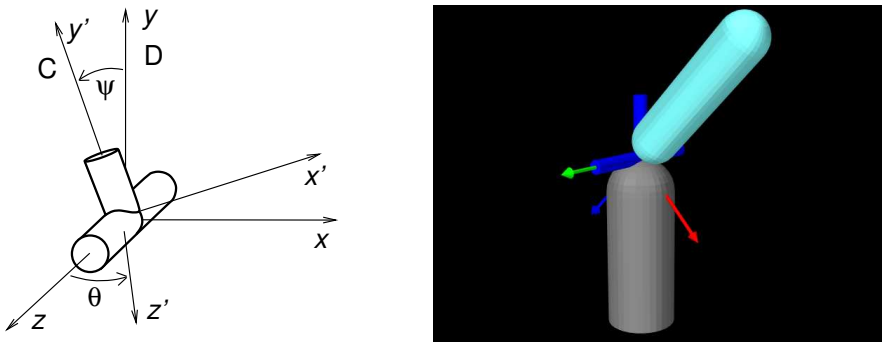


Figure 3.16: Coordinate frames (left) and demo model (right) for the universal joint.

the two coordinates θ and ϕ (Table 3.5), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} c_\theta c_\phi & -s_\theta & c_\theta s_\phi & 0 \\ s_\theta c_\phi & c_\theta & s_\theta s_\phi & 0 \\ -s_\phi & 0 & c_\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where

$$c_\theta \equiv \cos(\theta),\ s_\theta \equiv \sin(\theta),\ c_\phi \equiv \cos(\phi),\ s_\phi \equiv \sin(\phi).$$

The value and ranges for θ and ϕ are exported by the properties roll, pitch, rollRange and pitchRange, and the coordinate indices are defined by the constants `ROLL_IDX` and `PITCH_IDX`. For rendering, the properties shaftLength and shaftRadius control the size of shafts drawn about the roll and pitch axes, while jointRadius specifies the radius of a ball drawn around the origin of D; both are drawn using the faceColor rendering property. A demo is provided by `artisynth.demos.mech.UniversalJointDemo`.

3.4.6 Skewed Universal Joint

The SkewedUniversalJoint (Figure 3.17) is a version of the universal joint in which the pitch axis is skewed relative to its nominal direction by an angle α. More precisely, let x' and y' be the x and y axes of C after the initial roll rotation. For a regular universal joint, the pitch axis is y', whereas for a skewed universal joint it is y' rotated by α clockwise about x'. The joint still has 2 DOF, but the space of allowed rotations is reduced.

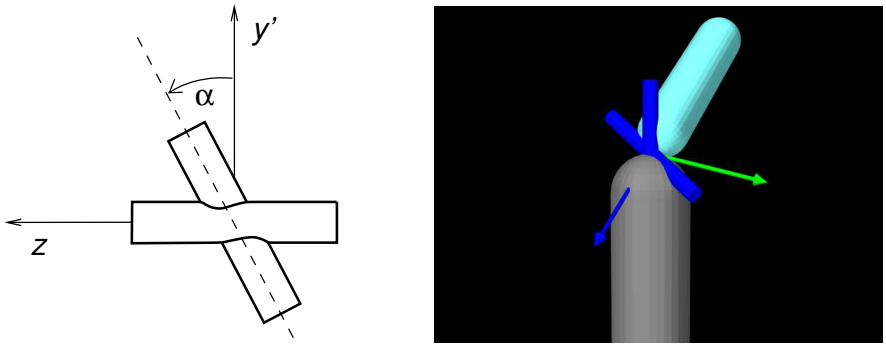


Figure 3.17: Left: diagram for a skewed universal joint, showing the pitch axis (dotted line) skewed by an angle α relative to its nominal direction along the y' axis. Right: demo model with skew angle of 30°.

The constraints and the coordinates are the same as for the universal joint, although the relationship between $\mathbf{T}_{CD}$ is now more complicated. With c_θ, s_θ, c_ϕ, and s_ϕ defined as for the universal joint, $\mathbf{T}_{CD}$ is given by

$$\mathbf{T}_{CD} = \begin{pmatrix} c_\theta c_\phi - s_\theta s_\alpha s_\phi & -s_\theta \beta - s_\alpha c_\theta s_\phi & c_\alpha (c_\theta s_\phi - s_\alpha s_\theta v_\phi) & 0 \\ s_\theta c_\phi + c_\theta s_\alpha s_\phi & c_\theta \beta - s_\alpha s_\theta s_\phi & c_\alpha (s_\theta s_\phi + s_\alpha c_\theta v_\phi) & 0 \\ -c_\alpha s_\phi & c_\alpha s_\alpha v_\phi & s_\alpha^2 + c_\alpha^2 c_\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where

$$c_\alpha \equiv \cos(\alpha), \quad s_\alpha \equiv \sin(\alpha), \quad v_\phi \equiv 1 - c_\phi, \quad \beta \equiv c_\alpha^2 + s_\alpha^2 c_\phi.$$

Rendering is controlled using the properties shaftLength, shaftRadius and jointRadius in the same way as for the `UniversalJoint`. A demo is provided by calling `artisynth.demos.mech.UniversalJointDemo` with the model arguments `-skew <angDeg>`, where `<angDeg>` is the desired skew angle in degrees.

Constructors for skewed universal joints take the standard forms described in Section 3.3.3, with an additional argument at the end indicating the skew angle:

```
SkewedUniveralJoint (bodyA, TCA, bodyB, TCB, skewAngle)

SkewedUniveralJoint (bodyA, bodyB, TDW, skewAngle)

SkewedUniveralJoint (bodyA, bodyB, TCW, TDW, skewAngle)
```

In addition, the constructor

```
SkewedUniveralJoint (bodyA, bodyB, originD, rollAxis, pitchAxis)
```

creates a skewed universal joint specifying the origin of frame D together with the directions of the roll and pitch axes (in world coordinates). Frames C and D are coincident and the skew angle is inferred from the angle between the axes.

3.4.7 Gimbal Joint

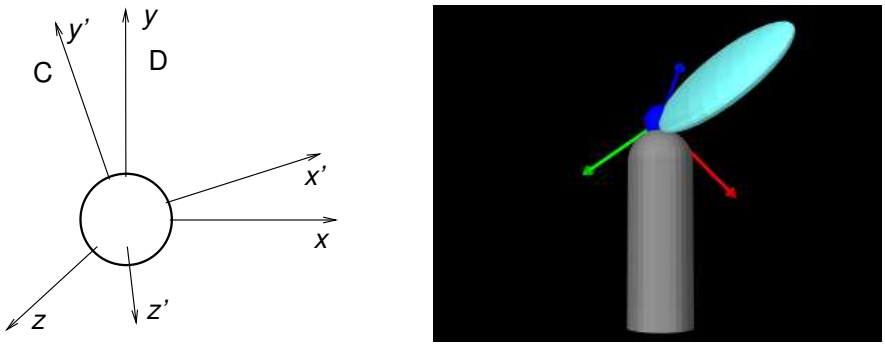


Figure 3.18: Coordinate frames (left; rotation angles not shown) and demo model (right) for the gimbal joint.

The GimbalJoint (Figure 3.18) is a 3 DOF spherical joint that anchors the origins of C and D together but otherwise allows C complete rotational freedom. The rotational degrees of freedom are parameterized by three roll-pitch-yaw angles, denoted by θ, ϕ, ψ, which define a rotation θ about D's z axis, followed by a second rotation ϕ about the rotated y' axis, followed by a third rotation ψ about the final x'' axis. It implements six constraints and the three coordinates θ, ϕ, ψ (Table 3.6), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} c_\theta c_\phi & c_\theta s_\phi s_\psi - s_\theta c_\psi & c_\theta s_\phi c_\psi + s_\theta s_\psi & 0 \\ s_\theta c_\phi & s_\theta s_\phi s_\psi + c_\theta c_\psi & s_\theta s_\phi c_\psi - c_\theta s_\psi & 0 \\ -s_\phi & c_\phi s_\psi & c_\phi c_\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where

$$c_\theta \equiv \cos(\theta),\ s_\theta \equiv \sin(\theta),\ c_\phi \equiv \cos(\phi),\ s_\phi \equiv \sin(\phi),\ c_\psi \equiv \cos(\psi),\ s_\psi \equiv \sin(\psi).$$

The value and ranges for θ, ϕ, ψ are exported by the properties roll, pitch, yaw, rollRange, pitchRange, and yawRange, and the coordinate indices are defined by the constants `ROLL_IDX`, `PITCH_IDX`, and `YAW_IDX`. For rendering, the property jointRadius specifies the radius of a ball drawn around the origin of D, using the faceColor rendering property. A demo is provided by `artisynth.demos.mech.GimbalJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
GimbalJoint (bodyA, bodyB, originD)
```

creates a gimbal joint with a specified origin for frame D (in world coordinates), and frames C and D coincident and world aligned.

> The constraints implementing `GimbalJoint` are designed so that it is immune to *gimbal lock*, in which a degree of freedom is lost when $\phi = \pm\pi/2$. However, the coordinate values themselves are not immune to this singularity, and neither are the unilateral constraints which enforce limits on their values. Therefore, if coordinate limits are implemented, the joint should be deployed so as try and avoid pitch values near $\pm\pi/2$.

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts translation along z
3	unilataral	enforces limits on the roll coordinate
4	unilataral	enforces limits on the pitch coordinate
5	unilataral	enforces limits on the yaw coordinate
0	θ (roll)	first rotation of C about the z axis of D
1	ϕ (pitch)	second rotation of C about the rotated y' axis
2	ψ (yaw)	third rotation of C about the final x'' axis

Table 3.6: Constraints (top) and coordinates (bottom) for the gimbal joint.

3.4.8 Spherical Joint

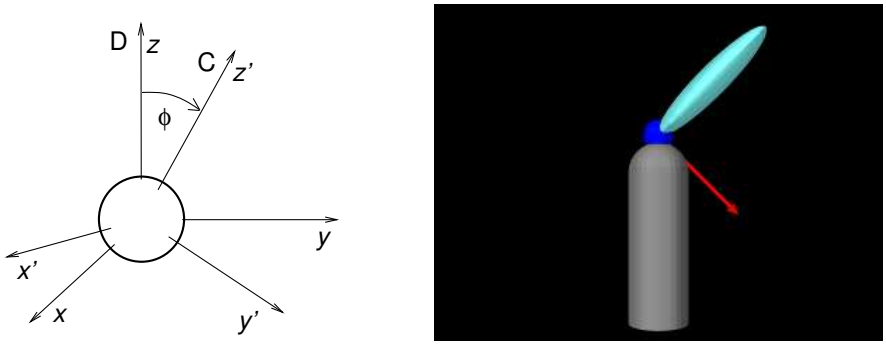


Figure 3.19: Left: coordinate frames of the spherical joint, showing the tilt angle ϕ between the z axes of C and D. Right: demo model for the spherical joint.

The SphericalJoint (Figure 3.19) is a 3 DOF spherical joint that, like GimbalJoint, anchors the origins of C and D together but otherwise allows C complete rotational freedom. `SphericalJoint` does not implement any coordinates, and so is conceptually more like a *ball* joint. However, it does provide two choices for limiting its rotation:

- A limit on the *tilt* angle ϕ between the z axes of D and C, such that

$$\phi \leq \phi_{\max}. \tag{3.25}$$

  This is intended to emulate the limit imposed by a ball joint socket.

- A limit on the total rotation, defined as follows: Let $(\mathbf{u}, \theta)$ be the axis-angle representation of the rotation matrix of $\mathbf{T}_{CD}$, normalized such that $\theta \geq 0$ and $\|u\| = 1$, and let $\mathbf{r}_{\max}$ be a three-vector giving maximum rotation angles with x, y, and z components. Then θ is constrained by

$$\theta \leq \|\mathbf{r}_{\max} \circ \mathbf{u}\|, \tag{3.26}$$

  where $\circ$ denotes the element-wise product. If the components of $\mathbf{r}_{\max}$ are set to a uniform value $\theta_{\max}$, this simplifies to $\theta \leq \theta_{\max}$.

These limits can be enabled by setting the joint's properties isTiltLimited and isRotationLimited, respectively, where enabling one disables the other. The limit values $\phi_{\max}$ and $\mathbf{r}_{\max}$ are managed using the properties maxTilt and maxRotation, and setting either automatically enables tilt or rotation limiting, as appropriate. Finally, the tilt angle ϕ can be queried using the (read-only) tilt property. For rendering, the property jointRadius specifies the radius of a ball drawn around the origin of D, using the faceColor rendering property. A demo of the `SphericalJoint` is provided by `artisynth.demos.mech.SphericalJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
SphericalJoint (bodyA, bodyB, originD)
```

creates a spherical joint with a specified origin for frame D (in world coordinates), and frames C and D coincident and world aligned.

> One should use the rotation limit with some caution, as the orientations which it prohibits can be somewhat hard to predict, particularly when $\mathbf{r}_{\max}$ has non-uniform values.

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts translation along z
3	unilataral	enforces either the "tilt" or "rotation" limits

Table 3.7: Constraints for the spherical joint.

3.4.9 Planar Joint

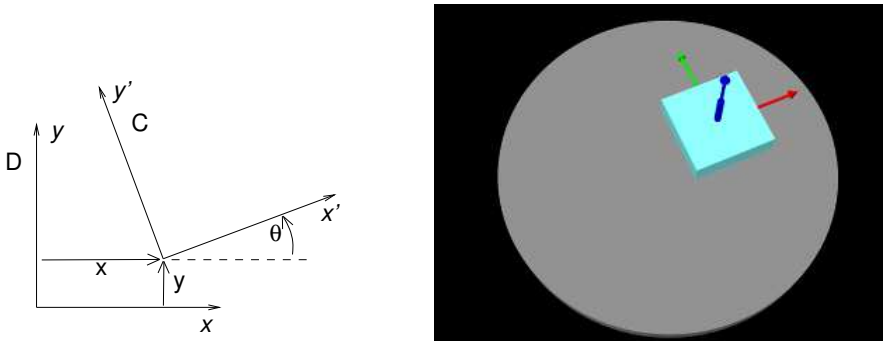


Figure 3.20: Coordinate frames (left) and demo model (right) for the planar joint.

The PlanarJoint (Figure 3.20) is a 3 DOF joint that constrains C to translation in the x-y plane and rotation about the z axis of D. It implements six constraints and three coordinates x, y and θ (Table 3.8), to which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & x \\ \sin(\theta) & \cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The value and ranges for x, y and θ are exported by the properties x, y, theta, xRange, yRange and thetaRange, and the coordinate indices are defined by the constants `X_IDX`, `Y_IDX` and `THETA_IDX`. A planar joint can be rendered as a square centered on the origin of D, using face rendering properties and with a size given by the planeSize property. For example,

```
PlanarJoint joint;
...
joint.setPlaneSize (5.0);
RenderProps.setFaceColor (joint, Color.LIGHT_GRAY);
```

will cause `joint` to be drawn as a light grey square with size 5.0. The default value of planeSize is 0, so drawing the plane is disabled by default. Also, the default faceStyle rendering property for `PlanarConnector` is set to `FRONT_AND_BACK`, so that the plane (when drawn) can be seen from both sides. A shaft about the rotation axis can also be drawn, as controlled by the properties shaftLength and shaftRadius and using the faceColor rendering property. A demo is provided by `artisynth.demos.mech.PlanarJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
PlanarJoint (bodyA, bodyB, originD, zaxis)
```

creates a planar joint with a specified origin and z axis direction for frame D (in world coordinates), and frames C and D coincident.

Index	type/name	description
0	bilateral	restricts translation along z
1	bilateral	restricts rotation about x
2	bilateral	restricts rotation about y
3	unilataral	enforces limits on the x coordinate
3	unilataral	enforces limits on the y coordinate
5	unilataral	enforces limits on the θ coordinate
0	x	translation of C along the x axis of D
1	y	translation of C along the y axis of D
2	θ	rotation of C about the z axis of D

Table 3.8: Constraints (top) and coordinates (bottom) for the planar joint.

3.4.10 Planar Translation Joint

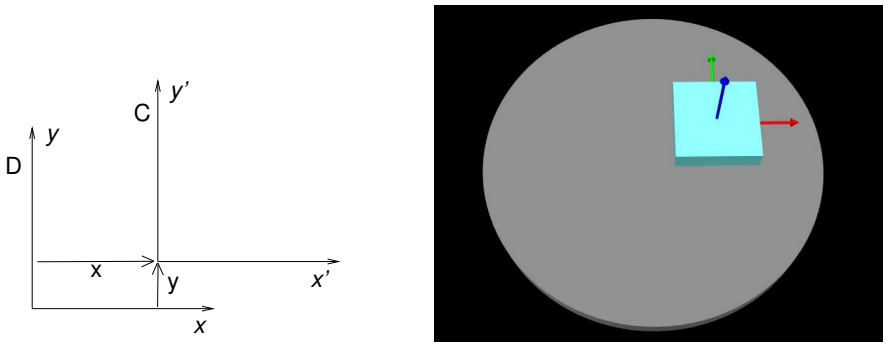


Figure 3.21: Coordinate frames (left) and demo model (right) for the planar translation joint.

Index	type/name	description
0	bilateral	restricts translation along z
1	bilateral	restricts rotation about x
2	bilateral	restricts rotation about y
3	bilateral	restricts rotation about z
4	unilataral	enforces limits on the x coordinate
5	unilataral	enforces limits on the y coordinate
0	x	translation of C along the x axis of D
1	y	translation of C along the y axis of D

Table 3.9: Constraints (top) and coordinates (bottom) for the planar translation joint.

The PlanarTranslationJoint (Figure 3.21) is a 2 DOF joint that is the same as the planar joint without rotation: C is restricted to translation in the x-y plane of D. It implements six constraints and two coordinates x and y (Table 3.9), to

which the joint transform $\mathbf{T}_{CD}$ is related by

$$\mathbf{T}_{CD} = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The value and ranges for x and y are exported by the properties x, y, xRange and yRange, and the coordinate indices are defined by the constants `X_IDX` and `Y_IDX`. A planar translation joint can be rendered as a square centered on the origin of D, using face rendering properties and with a size given by the planeSize property, in the same way as described for `PlanarJoint`. A demo is provided by `artisynth.demos.mech.PlanarJointDemo`.

In addition to the standard constructors described in Section 3.3.3,

```
PlanarTranslationJoint (bodyA, bodyB, originD, zaxis)
```

creates a planar translation joint with a specified origin and z axis direction for frame D (in world coordinates), and frames C and D coincident.

3.4.11 Solid Joint

The SolidJoint is a 0 DOF joint that rigidly constrains C to D. It implements six constraints and no coordinates (Table 3.10) and the resulting $\mathbf{T}_{CD}$ is the identity.

> There aren't normally many uses for solid joints. If one wishes to create a complex rigid body by joining together a variety of shapes, this can be done more efficiently by making these shapes mesh components of a single rigid body (Section 3.2.8).

Index	type/name	description
0	bilateral	restricts translation along x
1	bilateral	restricts translation along y
2	bilateral	restricts translation along z
3	bilateral	restricts rotation about x
4	bilateral	restricts rotation about y
5	bilateral	restricts rotation about z

Table 3.10: Constraints for the solid joint.

3.4.12 Planar Connector

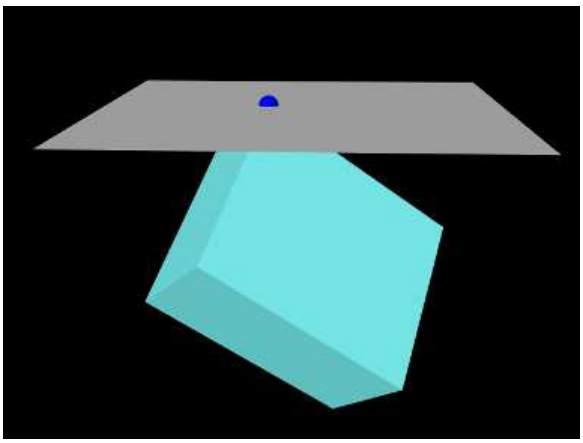


Figure 3.22: Demo model for the planar connector, in which a corner point of a box is constrained to the x-y plane of D.

Index	type/name	description
0	bilateral *or* unilateral	restricts translation along z

Table 3.11: Constraints for the planar connector.

The PlanarConnector (Figure 3.22) is a 5 DOF connector that attaches the origin of C to the x-y plane of D. C is completely free to rotate, and to translate within the x-y plane. Only motion in the z direction is restricted. `PlanarConnector` implements one constraint and has no coordinates (Table 3.11).

A `PlanarConnector` constrains a point on body A (located at the origin of C) to move within a plane on body B. Several planar connectors can be employed to constrain body motions in more complicated ways, although one must be careful to avoid overconstraining the system. The connector can also be configured to function *unilaterally*, via its unilateral property, in which case the point is constrained to lie in the half-space defined by $z \geq 0$ with respect to D. Several unilateral `PlanarConnector`s can therefore be used to implement a cheap and approximate collision mechanism with fixed collision points.

> When set to function unilaterally, overconstraining the system is not an issue because of the way in which ArtiSynth solves unilateral constraints.

A planar connector can be rendered as a square centered on the origin of D, using face rendering properties and with a size given by the planeSize property. The point attached to A can also be rendered using point rendering properties. For example,

```
PlanarConnector connector;
...
connector.setPlaneSize (5.0);
RenderProps.setFaceColor (connector, Color.LIGHT_GRAY);
RenderProps.setSphericalPoints (connector, 0.1, Color.BLUE);
```

will cause `connector` to be drawn as a light grey square with size 5, and for the point on body A to be drawn as a blue sphere with radius 0.1. The default value of planeSize is 0, so drawing the plane is disabled by default. Also, the default faceStyle rendering property for `PlanarConnector` is set to `FRONT_AND_BACK`, so that the plane (when drawn) can be seen from both sides.

Constructors for the `PlanarConnector` include

```
PlanarConnector (bodyA, pCA, bodyB, TDB)

PlanarConnector (bodyA, pCA, TDW)

PlanarConnector (bodyA, bodyA, TDW)
```

where `pCA` gives the connection point of body A *with respect to frame A*, `TDB` gives the transform from frame D to frame B, and `TDW` gives the transform from frame D to world.

3.4.13 Segmented Planar Connector

Index	type/name	description
0	bilateral *or* unilateral	restricts translation normal to the surface

Table 3.12: Constraints for the segmented planar connector.

The SegmentedPlanarConnector (Figure 3.23) is a 5 DOF connector that generalizes `PlanarConnector` to a piecewise linear surface, to which the origin of C is constrained while C is otherwise completely free to rotate. The surface is specified by a sequence of 2D points defining a piecewise linear curve in the x-z plane of D (Figure 3.23, left). This curve does not need to be a function; the segment nearest to C is the one used to enforce the constraint at any given time. The surface has infinite extent and is extrapolated beyond the first and last segments. It implements one constraint and has no coordinates (Table 3.12).

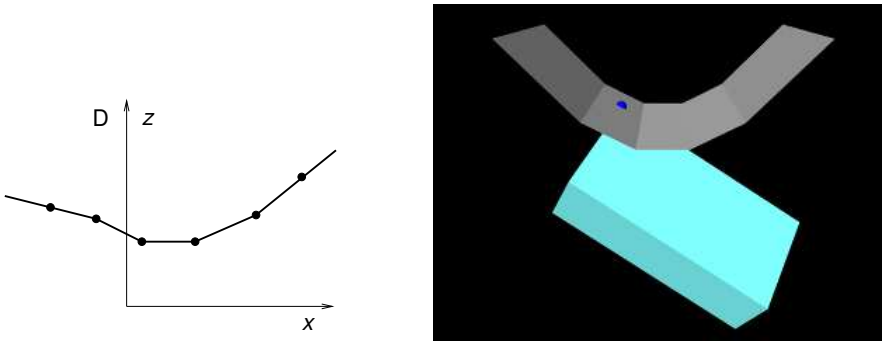


Figure 3.23: Left: cross-section in the x-z plane of frame D showing the segments of a segmented planar connector, with the points defining the segments shown as black dots. Right: demo model for the segmented planar connector.

By appropriate choice of segments, a `SegmentedPlanarConnector` can approximate any surface defined by a curve in the x-z plane. As with `PlanarConnector`, it can also be configured as unilateral, constraining the origin of C to lie on the side of the surface defined by the normal vectors $\mathbf{n}_k$ of each segment k. If $\mathbf{p}_{k-1}$ and $\mathbf{p}_k$ are the points in the x-z plane defining the k-th segment, and $\hat{\mathbf{y}}$ is the y axis unit vector, then $\mathbf{n}_k$ is given by

$$\mathbf{n}_k = \frac{\mathbf{u} \times \hat{\mathbf{y}}}{\|\mathbf{u} \times \hat{\mathbf{y}}\|}, \quad \mathbf{u} \equiv \mathbf{p}_k - \mathbf{p}_{k-1}. \tag{3.27}$$

The properties controlling the rendering of a segmented planar connector are the same as for a planar connector, with each of the individual plane segments drawn as a rectangle whose length along the y axis is controlled by planeSize.

Constructors for a `SegmentedPlanarConnector` are analogous to those used for `PlanarConnector`,

```
SegmentedPlanarConnector (bodyA, pCA, bodyB, TDB, segs)

SegmentedPlanarConnector (bodyA, pCA, TDW, segs)

SegmentedPlanarConnector (bodyA, bodyA, TDW, segs)
```

where `segs` is an additional argument of type `double[]` giving the 2D coordinates defining the segments in the x-z plane.

3.4.14 Legacy Joints

ArtiSynth maintains three legacy joint for compatibility with earlier software:

- RevoluteJoint is identical to the HingeJoint, except that its coordinate θ is oriented *clockwise* about the z axis instead of *counter-clockwise*. Rendering is also done differently, with shafts about the rotation axis drawn using line rendering properties.
- RollPitchJoint is identical to the UniversalJoint, except that its roll-pitch coordinates θ, ϕ are computed with respect to the rotation $\mathbf{R}_{DC}$ from frame D to C, instead of the rotation $\mathbf{R}_{CD}$ from frame C to D. Rendering is also done differently, with shafts along the roll and pitch axes drawn using line rendering properties, and the ball around the origin of D drawn using point rendering properties.
- SphericalRpyJoint is identical to the GimbalJoint, except that its roll-pitch-yaw coordinates θ, ϕ, ψ are computed with respect to the rotation $\mathbf{R}_{DC}$ from frame D to C, instead of the rotation $\mathbf{R}_{CD}$ from frame C to D. Rendering is also done differently, with the ball around the origin of D drawn using point rendering properties.

3.5 Frame springs

Another way to connect two rigid bodies together is to use a *frame spring*, which is a six dimensional spring that generates restoring forces and moments between coordinate frames.

3.5.1 Frame spring coordinate frames

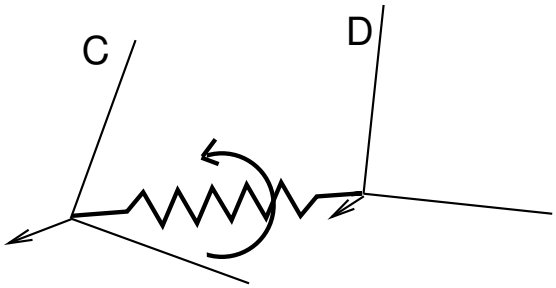


Figure 3.24: A frame spring connecting two coordinate frames D and C.

The basic idea of a frame spring is shown in Figure 3.24. It generates restoring forces and moments on two frames C and D which are a function of $\mathbf{T}_{DC}$ and $\hat{\mathbf{v}}_{DC}$ (the spatial velocity of frame D with respect to frame C).

Decomposing forces into stiffness and damping terms, the force $\mathbf{f}_C$ and moment τ_C acting on C can be expressed as

$$\begin{aligned} \mathbf{f}_C &= \mathbf{f}_k(\mathbf{T}_{DC}) + \mathbf{f}_d(\hat{\mathbf{v}}_{DC}) \\ \tau_C &= \tau_k(\mathbf{T}_{DC}) + \tau_d(\hat{\mathbf{v}}_{DC}). \end{aligned} \tag{3.28}$$

where the translational and rotational forces $\mathbf{f}_k$, $\mathbf{f}_d$, τ_k, and τ_d are general functions of $\mathbf{T}_{DC}$ and $\hat{\mathbf{v}}_{DC}$.

The forces acting on D are equal and opposite, so that

$$\begin{aligned} \mathbf{f}_D &= -\mathbf{f}_C, \\ \tau_D &= -\tau_C. \end{aligned} \tag{3.29}$$

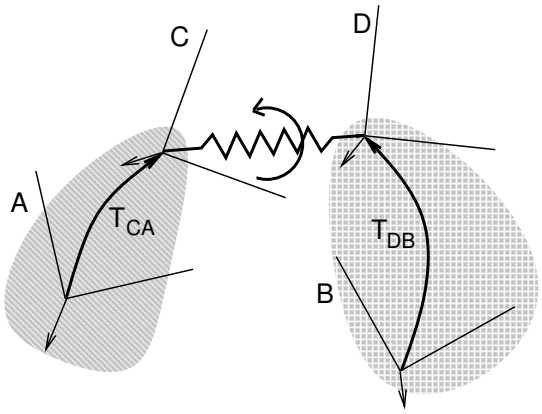


Figure 3.25: A frame spring connecting two rigid bodies A and B.

If frames C and D are attached to a pair of rigid bodies A and B, then a frame spring can be used to connect them in a manner analogous to a joint. As with joints, C and D generally do not coincide with the body frames, and are instead offset from them by fixed transforms $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ (Figure 3.25).

3.5.2 Frame materials

The restoring forces (3.28) generated in a frame spring depend on the *frame material* associated with the spring. Frame materials are defined in the package `artisynth.core.materials`, and are subclassed from FrameMaterial. The most basic type of material is a LinearFrameMaterial, in which the restoring forces are determined from

$$\begin{aligned} \mathbf{f}_C &= \mathbf{K}_t\, \mathbf{x}_{DC} + \mathbf{D}_t\, \mathbf{v}_{DC} \\ \tau_C &= \mathbf{K}_r\, \hat{\theta}_{DC} + \mathbf{D}_r\, \omega_{DC} \end{aligned}$$

where $\hat{\theta}_{DC}$ gives the small angle approximation of the rotational components of $\mathbf{X}_{DC}$ with respect to the x, y, and z axes, and

$$\mathbf{K}_t \equiv \begin{pmatrix} k_{tx} & 0 & 0 \\ 0 & k_{ty} & 0 \\ 0 & 0 & k_{tz} \end{pmatrix}, \mathbf{D}_t \equiv \begin{pmatrix} d_{tx} & 0 & 0 \\ 0 & d_{ty} & 0 \\ 0 & 0 & d_{tz} \end{pmatrix},$$
$$\mathbf{K}_r \equiv \begin{pmatrix} k_{rx} & 0 & 0 \\ 0 & k_{ry} & 0 \\ 0 & 0 & k_{rz} \end{pmatrix}, \mathbf{D}_r \equiv \begin{pmatrix} d_{rx} & 0 & 0 \\ 0 & d_{ry} & 0 \\ 0 & 0 & d_{rz} \end{pmatrix}.$$

are the stiffness and damping matrices. The diagonal values defining each matrix are stored in the 3-dimensional vectors $\mathbf{k}_t$, $\mathbf{k}_r$, $\mathbf{d}_t$, and $\mathbf{d}_r$ which are exposed as the `stiffness`, `rotaryStiffness`, `damping`, and `rotaryDamping` properties of the material. Each of these specifies stiffness or damping values along or about a particular axis. Specifying different values for different axes will result in anisotropic behavior.

Other frame materials offering nonlinear behavior may be defined in `artisynth.core.materials`.

3.5.3 Creating frame springs

Frame springs are implemented by the class FrameSpring. Creating a frame spring generally involves instantiating this class, and then setting the material, the bodies A and B, and the transforms $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$.

A typical construction sequence might look like this:

```
FrameSpring spring = new FrameSpring ("springA");
spring.setMaterial (new LinearFrameMaterial (kt, kr, dt, dr));
spring.setFrames (bodyA, bodyB, TDW);
```

The material is set using setMaterial(). The example above uses a `LinearFrameMaterial`, created with a constructor that sets $\mathbf{k}_t$, $\mathbf{k}_r$, $\mathbf{d}_t$, and $\mathbf{d}_r$ to uniform Isotropic values specified by `kt`, `kr`, `dt`, and `dr`.

The bodies and transforms can be set in the same manner as for joints (Section 3.3.3), with the methods setFrames(bodyA,bodyB,TDW) and setFrames(bodyA,TCA,bodyB,TDB) assuming the role of the `setBodies()` methods used for joints. The former takes D specified in world coordinates and computes $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ assuming that there is no initial spring displacement (i.e., that $\mathbf{T}_{DC} = \mathbf{I}$), while the latter allows $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ to be specified explicitly with $\mathbf{T}_{DC}$ assuming whatever value is implied.

Frame springs and joints are often placed together, using the same transforms $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$, with the spring providing restoring forces to help keep the joint within prescribed bounds.

As with joints, a frame spring can be connected to only a single body, by specifying `frameB` as `null`. Frame B is then taken to be the world coordinate frame W.

3.5.4 Example: two bodies connected by a frame spring

A simple model showing two simplified lumbar vertebrae, modeled as rigid bodies and connected by a frame spring, is defined in

```
artisynth.demos.tutorial.LumbarFrameSpring
```

The definition for the entire model class is shown here:

```
1  package artisynth.demos.tutorial;
2
3  import java.io.IOException;
4  import java.io.File;
5  import java.awt.Color;
6  import artisynth.core.modelbase.*;
7  import artisynth.core.mechmodels.*;
8  import artisynth.core.materials.*;
9  import artisynth.core.workspace.RootModel;
10 import maspack.matrix.*;
```

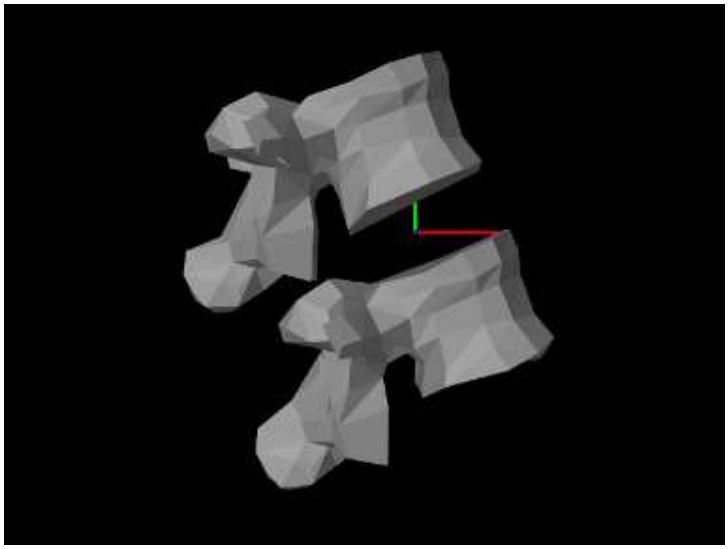


Figure 3.26: LumbarFrameSpring model loaded into ArtiSynth.

```
import maspack.geometry.*;
import maspack.render.*;
import maspack.util.PathFinder;

/**
 * Demo of two rigid bodies connected by a 6 DOF frame spring
 */
public class LumbarFrameSpring extends RootModel {

   double density = 1500;

   // path from which meshes will be read
   private String geometryDir = PathFinder.getSourceRelativePath (
      LumbarFrameSpring.class, "../mech/geometry/");

   // create and add a rigid body from a mesh
   public RigidBody addBone (MechModel mech, String name) throws IOException {
      PolygonalMesh mesh = new PolygonalMesh (new File (geometryDir+name+".obj"));
      RigidBody rb = RigidBody.createFromMesh (name, mesh, density, /*scale=*/1);
      mech.addRigidBody (rb);
      return rb;
   }

   public void build (String[] args) throws IOException {

      // create mech model and set it's properties
      MechModel mech = new MechModel ("mech");
      mech.setGravity (0, 0, -1.0);
      mech.setFrameDamping (0.10);
      mech.setRotaryDamping (0.001);
      addModel (mech);

      // create two rigid bodies and second one to be fixed
      RigidBody lumbar1 = addBone (mech, "lumbar1");
      RigidBody lumbar2 = addBone (mech, "lumbar2");
      lumbar1.setPose (new RigidTransform3d (-0.016, 0.039, 0));
      lumbar2.setDynamic (false);

      // flip entire mech model around
      mech.transformGeometry (
         new RigidTransform3d (0, 0, 0, 0, 0, Math.toRadians (90)));
```

```
52
53        //create and add the frame spring
54        FrameSpring spring = new FrameSpring (null);
55        spring.setMaterial (
56           new LinearFrameMaterial (
57              /*ktrans=*/100, /*krot=*/0.01, /*dtrans=*/0, /*drot=*/0));
58        spring.setFrames (lumbar1, lumbar2, lumbar1.getPose());
59        mech.addFrameSpring (spring);
60
61        // set render properties for components
62        RenderProps.setLineColor (spring, Color.RED);
63        RenderProps.setLineWidth (spring, 3);
64        spring.setAxisLength (0.02);
65        RenderProps.setFaceColor (mech, new Color (238, 232, 170)); // bone color
66     }
67  }
```

For convenience, the code to create and add each vertebrae is wrapped into the method `addBone()` defined at lines 27-32. This method takes two arguments: the `MechModel` to which the bone should be added, and the name of the bone. Surface meshes for the bones are located in `.obj` files located in the directory `../mech/geometry` relative to the source directory for the model itself. PathFinder.getSourceRelativePath() is used to find a proper path to this directory (see Section 2.6) given the model class type (`LumbarFrameSpring.class`), and this is stored in the static string `geometryDir`. Within `addBone()`, the directory path and the bone name are used to create a path to the bone mesh itself, which is in turn used to create a `PolygonalMesh` (line 28). The mesh is then used in conjunction with a `density` to create a rigid body which is added to the `MechModel` (lines 29-30) and returned.

The `build()` method begins by creating and adding a `MechModel`, specifying a low value for gravity, and setting the rigid body damping properties `frameDamping` and `rotaryDamping` (lines 37-41). (The damping parameters are needed here because the frame spring itself is created with no damping.) Rigid bodies representing the vertebrae `lumbar1` and `lumbar2` are then created by calling `addBone()` (lines 44-45), `lumbar1` is translated by setting the origin of its pose to $(-0.016, 0.039, 0)^T$, and `lumbar2` is set to be fixed by making it non-dynamic (line 47).

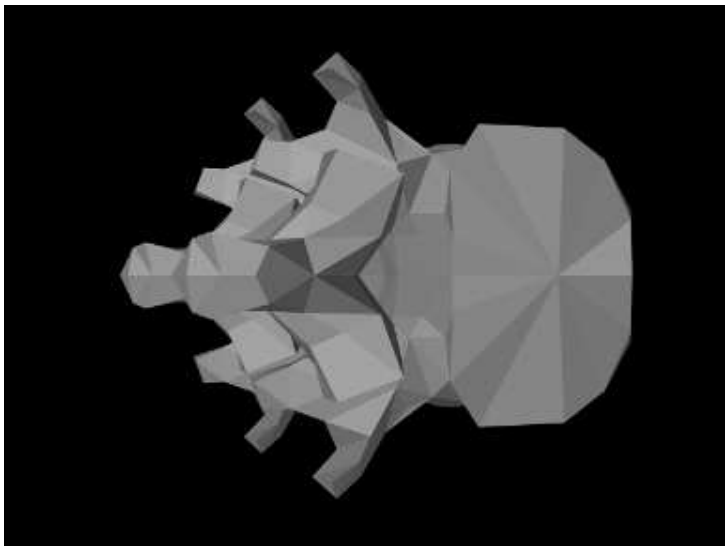


Figure 3.27: LumbarFrameSpring model as it would appear if not rotated about the x axis.

At this point in the construction, if the model were to be loaded, it would appear as in Figure 3.27. To change the viewpoint to that seen in Figure 3.26, we rotate the entire model about the x axis (line 50). This is done using transformGeometry(X), which transforms the geometry of an entire model using a rigid or affine transform. This method is described in more detail in Section 4.8.

The frame spring is created and added at lines 54-59, using the methods described in Section 3.5.3, with frame D set to the (initial) pose of `lumbar1`.

Render properties are set starting at line 62. By default, a frame spring renders as a pair of red, green, blue coordinate axes showing frames C and D, along with a line connecting them. The line width and the color of the connecting line are

controlled by the line render properties `lineWidth` and `lineColor`, while the length of the coordinate axes is controlled by the special frame spring property `axisLength`.

To run this example in ArtiSynth, select All demos > tutorial > LumbarFrameSpring from the Models menu. The model should load and initially appear as in Figure 3.26. Running the model (Section 1.5.3) will cause `lumbar1` to fall slightly under gravity until the frame spring arrests the motion. To get a sense of the spring's behavior, one can interactively apply forces to `lumbar1` using the pull tool (see the section "Pull Manipulation" in the ArtiSynth User Interface Guide).

3.6 Attachments

ArtiSynth provides the ability to rigidly attach dynamic components to other dynamic components, allowing different parts of a model to be connected together. Attachments are made by adding to a `MechModel` special *attachment* components that manage the attachment physics as described briefly in Section 1.2.

3.6.1 Point attachments

Point attachments allow particles and other point-based components to be attached to other, more complex components, such as frames, rigid bodies, or finite element models (Section 6.4). Point attachments are implemented by creating attachment components that are instances of PointAttachment. Modeling applications do not generally handle the attachment components directly, but instead create them implicitly using the following `MechModel` method:

```
attachPoint (Point p1, PointAttachable comp);
```

This attaches a point `p1` to any component which implements the interface PointAttachable, indicating that it is capable creating an attachment to a point. Components that implement `PointAttachable` currently include rigid bodies, particles, and finite element models. The attachment is created based on the the current position of the point and component in question. For attaching a point to a rigid body, another method may be used:

```
attachPoint (Point p1, RigidBody body, Point3d loc);
```

This attaches `p1` to `body` at the point `loc` specified in body coordinates. Finite element attachments are discussed in Section 6.4.

Once a point is attached, it will be in the *attached* state, as described in Section 3.1.3. Attachments can be removed by calling

```
detachPoint (Point p1);
```

3.6.2 Example: model with particle attachments

A model illustrating particle-particle and particle-rigid body attachments is defined in

`artisynth.demos.tutorial.ParticleAttachment`

and most of the code is shown here:

```
1   public Particle addParticle (MechModel mech, double x, double y, double z) {
2      // create a particle at x, y, z and add it to mech
3      Particle p = new Particle (/*name=*/null, /*mass=*/.1, x, y, z);
4      mech.addParticle (p);
5      return p;
6   }
7
8   public AxialSpring addSpring (MechModel mech, Particle p1, Particle p2){
9      // create a spring connecting p1 and p2 and add it to mech
10     AxialSpring spr = new AxialSpring (/*name=*/null, /*restLength=*/0);
11     spr.setMaterial (new LinearAxialMaterial (/*k=*/20, /*d=*/10));
12     spr.setPoints (p1, p2);
13     mech.addAxialSpring (spr);
```

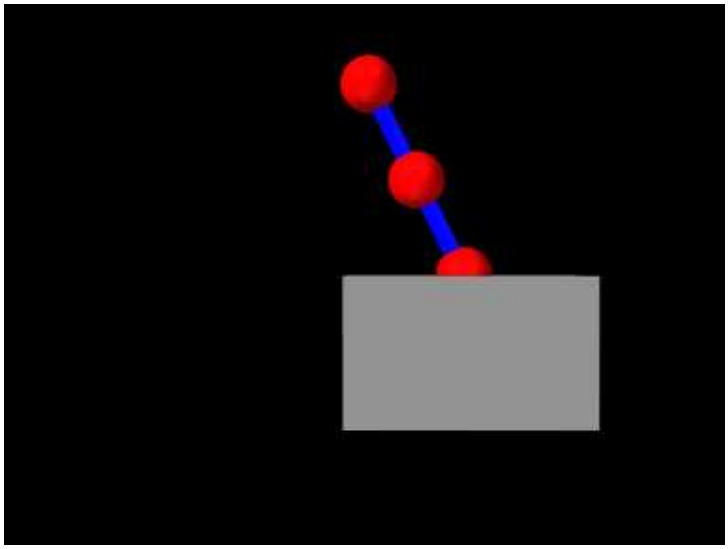


Figure 3.28: ParticleAttachment model loaded into ArtiSynth.

```
14        return spr;
15     }
16
17     public void build (String[] args) {
18
19        // create MechModel and add to RootModel
20        MechModel mech = new MechModel ("mech");
21        addModel (mech);
22
23        // create the components
24        Particle p1 = addParticle (mech, 0, 0, 0.55);
25        Particle p2 = addParticle (mech, 0.1, 0, 0.35);
26        Particle p3 = addParticle (mech, 0.1, 0, 0.35);
27        Particle p4 = addParticle (mech, 0, 0, 0.15);
28        addSpring (mech, p1, p2);
29        addSpring (mech, p3, p4);
30        // create box and set its pose (position/orientation):
31        RigidBody box =
32           RigidBody.createBox ("box", /*wx,wy,wz=*/0.5, 0.3, 0.3, /*density=*/20);
33        box.setPose (new RigidTransform3d (/*x,y,z=*/0.2, 0, 0));
34        mech.addRigidBody (box);
35
36        p1.setDynamic (false);              // first particle set to be fixed
37
38        // set up the attachments
39        mech.attachPoint (p2, p3);
40        mech.attachPoint (p4, box, new Point3d (0, 0, 0.15));
41
42        // increase model bounding box for the viewer
43        mech.setBounds (/*min=*/-0.5, 0, -0.5, /*max=*/0.5, 0, 0);
44        // set render properties for the components
45        RenderProps.setSphericalPoints (mech, 0.06, Color.RED);
46        RenderProps.setCylindricalLines (mech, 0.02, Color.BLUE);
47     }
```

The code is very similar to `ParticleSpring` and `RigidBodySpring` described in Sections 3.1.2 and 3.2.2, except that two convenience methods, `addParticle()` and `addSpring()`, are defined at lines 1-15 to create particles and spring and add them to a `MechModel`. These are used in the `build()` method to create four particles and two springs (lines 24-29), along with a rigid body box (lines 31-34). As with the other examples, particle `p1` is set to be non-dynamic (line 36) in order to fix it in place and provide a ground.

The attachments are added at lines 39-40, with `p2` attached to `p3` and `p4` connected to the box at the location $(0, 0, 0.15)$ in box coordinates.

Finally, render properties are set starting at line 43. In this example, point and line render properties are set for the entire `MechModel` instead of individual components. Since render properties are inherited, this will implicitly set the specified render properties in all sub-components for which these properties are not explicitly set (either locally or in an intermediate ancestor).

To run this example in ArtiSynth, select All demos > tutorial > ParticleAttachment from the Models menu. The model should load and initially appear as in Figure 3.28. Running the model (Section 1.5.3) will cause the box to fall and swing under gravity.

3.6.3 Frame attachments

Frame attachments allow rigid bodies and other frame-based components to be attached to other components, including frames, rigid bodies, or finite element models (Section 6.6). Frame attachments are implemented by creating attachment components that are instances of FrameAttachment.

As with point attachments, modeling applications do not generally handle frame attachment components directly, but instead create and add them implicitly using the following `MechModel` methods:

```
attachFrame (Frame frame, FrameAttachable comp);

attachFrame (Frame frame, FrameAttachable comp, RigidTransform3d TFW);
```

These attach `frame` to any component which implements the interface FrameAttachable, indicating that it is capable of creating an attachment to a frame. Components that implement `FrameAttachable` currently include frames, rigid bodies, and finite element models. For the first method, the attachment is created based on the the current position of the frame and component in question. For the second method, the attachment is created so that the initial pose of the frame (in world coordinates) is described by `TFW`.

Once a frame is attached, it will be in the *attached* state, as described in Section 3.1.3. Frame attachments can be removed by calling

```
detachFrame (Frame frame);
```

> While it is possible to create composite rigid bodies using `FrameAttachments`, this is much less computationally efficient (and less accurate) than creating a single rigid body through mesh merging or similar techniques.

3.6.4 Example: model with frame attachments

A model illustrating rigidBody-rigidBody and frame-rigidBody attachments is defined in

```
artisynth.demos.tutorial.FrameBodyAttachment
```

Most of the code is identical to that for `RigidBodyJoint` as described in Section 3.3.5, except that the joint is further to the left and connects `bodyB` to ground, rather than to `bodyA`, and the initial pose of `bodyA` is changed so that it is aligned vertically. `bodyA` is then connected to `bodyB`, and an auxiliary frame is created and attached to `bodyA`, using code at the end of the `build()` method as shown here:

```
1   public void build (String[] args) {
2
3      ... create model mostly similar to RigidBodyJoint ...
4
5      // now connect bodyA to bodyB using a FrameAttachment
6      mech.attachFrame (bodyA, bodyB);
7
8      // create an auxiliary frame and add it to the mech model
9      Frame frame = new Frame();
10     mech.addFrame (frame);
```

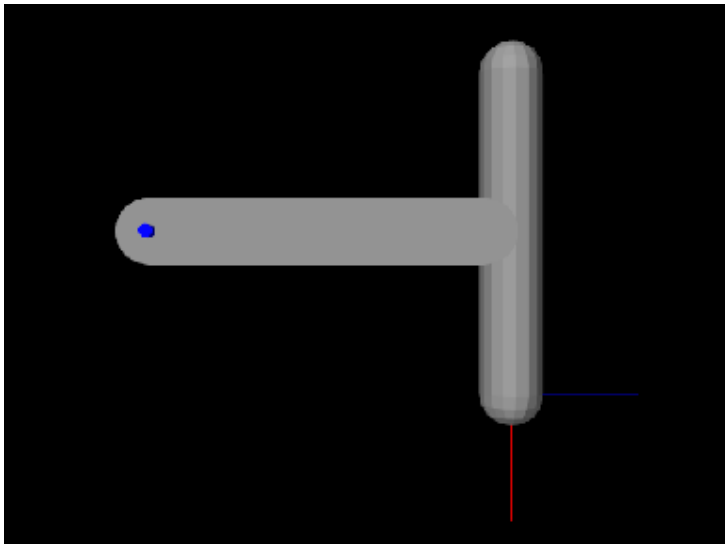


Figure 3.29: FrameBodyAttachment model loaded into ArtiSynth.

```
11
12        // set the frames axis length > 0 so we can see it
13        frame.setAxisLength (4.0);
14        // set the attached frame's pose to that of bodyA ...
15        RigidTransform3d TFW = new RigidTransform3d (bodyA.getPose());
16        // ... plus a translation of lenx2/2 along the x axis:
17        TFW.mulXyz (lenx2/2, 0, 0);
18        // finally, attach the frame to bodyA
19        mech.attachFrame (frame, bodyA, TFW);
20     }
```

To run this example in ArtiSynth, select All demos > tutorial > FrameBodyAttachment from the Models menu. The model should load and initially appear as in Figure 3.28. The frame attached to `bodyA` is visible in the lower right corner. Running the model (Section 1.5.3) will cause both bodies to fall and swing about the joint under gravity.

Chapter 4

Mechanical Models II

This section provides additional material on building basic multibody-type mechanical models.

4.1 Simulation control properties

Both RootModel and MechModel contain properties that control the simulation behavior.

4.1.1 maxStepSize

One of the most important of these is `maxStepSize`. By default, simulation proceeds using the `maxStepSize` value defined for the root model. A `MechModel` (or any other type of `Model`) contained in the root model's `models` list may also request a smaller step size by specifying a smaller value for its own `maxStepSize` property. For all models, the `maxStepSize` may be set and queried using

```
void setMaxStepSize (double maxh);
double getMaxStepSize();
```

4.1.2 integrator

Another important simulation property is `integrator` in `MechModel`, which determines the type of integrator used for the physics simulation. The value type of this property is the enumerated type `MechSystemSolver.Integrator`, for which the following values are currently defined:

ForwardEuler

First order forward Euler integrator. Unstable for stiff systems.

SymplecticEuler

First order symplectic Euler integrator, more energy conserving that forward Euler. Unstable for stiff systems.

RungeKutta4

Fourth order Runge-Kutta integrator, quite accurate but also unstable for stiff systems.

ConstrainedBackwardEuler

First order backward order integrator. Generally stable for stiff systems.

Trapezoidal

Second order trapezoidal integrator. Generally stable for stiff systems, but slightly less so than `ConstrainedBackwardEuler`.

The term "Unstable for stiff systems" means that the integrator is likely to go unstable in the presence of "stiff" systems, which typically include systems containing finite element models, unless the simulation step size is set to an extremely small value. The default value for `integrator` is `ConstrainedBackwardEuler`.

> Stiff systems tend to arise in models containing interconnected deformable elements, for which the step size should not exceed the propagation time across the smallest element, an effect known as the Courant-Friedrichs-Lewy (CFL) condition. Larger stiffness and damping values decrease the propagation time and hence the allowable step size.

4.1.3 stabilization

Another `MechModel` simulation property is `stabilization`, which controls the stabilization method used to correct drift from position constraints and correct interpenetrations due to collisions. The value type of this property value is the enumerated type `MechSystemSolver.PosStabilization`, which presently has two values:

GlobalMass

Uses only a diagonal mass matrix for the MLCP that is solved to determine the position corrections. This is the default method.

GlobalStiffness

Uses a stiffness-corrected mass matrix for the MLCP that is solved to determine the position corrections. Slower than `GlobalMass`, but more likely to produce stable results, particularly for problems involving FEM collisions.

4.2 Units

ArtiSynth is primarily "unitless", in the sense that it does not define default units for the fundamental physical quantities of time, length, and mass. Although time is generally understood to be in seconds, and often declared as such in method arguments and return values, there is no hard requirement that it be interpreted as seconds. There are no assumptions at all regarding length and mass. Some components may have default parameter values that reflect a particular choice of units, such as `MechModel`'s default gravity value of $(0, 0, -9.8)^T$, which is associated with the MKS system, but these values can always be overridden by the application.

Nevertheless, it is important, and up to the application developer to ensure, that units be *consistent*. For example, if one decides to switch length units from meters to centimeters (a common choice), then all units involving length will have to be scaled appropriately. For example, density, whose fundamental units are m/d^3, where m is mass and d is distance, needs to be scaled by $1/100^3$, or 0.000001, when converting from meters to centimeters.

Table 4.1 lists a number of common physical quantities used in ArtiSynth, along with their associated fundamental units.

4.2.1 Scaling units

For convenience, many ArtiSynth components, including `MechModel`, implement the interface ScalableUnits, which provides the following methods for scaling mass and distance units:

```
scaleDistance (s);     // scale distance units by s
scaleMass (s);         // scale mass units by s
```

A call to one of these methods should cause all physical quantities within the component (and its descendants) to be scaled as required by the fundamental unit relationships as shown in Table 4.1.

Converting a `MechModel` from meters to centimeters can therefore be easily done by calling

```
mech.scaleDistance (100);
```

As an example, adding the following code to the end of the `build()` method in `RigidBodySpring` (Section 3.2.2)

unit	fundamental units	
time	t	
distance	d	
mass	m	
velocity	d/t	
acceleration	d/t^2	
force	md/t^2	
work/energy	md^2/t^2	
torque	md^2/t^2	same as energy (somewhat counter intuitive)
angular velocity	$1/t$	
angular acceleration	$1/t^2$	
rotational inertia	md^2	
pressure	$m/(dt^2)$	
Young's modulus	$m/(dt^2)$	
Poisson's ratio	1	no units; it is a ratio
density	m/d^3	
linear stiffness	m/t^2	
linear damping	m/t	
rotary stiffness	md^2/t^2	same as torque
rotary damping	md^2/t	
mass damping	$1/t$	used in FemModel
stiffness damping	t	used in FemModel

Table 4.1: Physical quantities and their representation in terms of the fundamental units of mass (m), distance (d), and time (t).

```
System.out.println ("length=" + spring.getLength());
System.out.println ("density=" + box.getDensity());
System.out.println ("gravity=" + mech.getGravity());
mech.scaleDistance (100);
System.out.println ("");
System.out.println ("scaled length=" + spring.getLength());
System.out.println ("scaled density=" + box.getDensity());
System.out.println ("scaled gravity=" + mech.getGravity());
```

will scale the distance units by 100 and print the values of various quantities before and after scaling. The resulting output is:

```
length=0.5
density=20.0
gravity=0.0 0.0 -9.8

scaled length=50.0
scaled density=2.0E-5
scaled gravity=0.0 0.0 -980.0
```

> It is important not to confuse scaling units with scaling the actual geometry or mass. Scaling units should change all physical quantities so that the simulated behavior of the model remains unchanged. If the distance-scaled version of `RigidBodySpring` shown above is run, it should behave exactly the same as the non-scaled version.

4.3 Render properties

All ArtiSynth components that are renderable maintain a property `renderProps`, which stores a RenderProps object that contains a number of subproperties used to control an object's rendered appearance.

In code, the `renderProps` property for an object can be set or queried using the methods

property	purpose	usual default value
visible	whether or not the component is visible	`true`
alpha	transparency for diffuse colors (range 0 to 1)	1 (opaque)
shading	shading style: (`FLAT`, `SMOOTH`, `METAL`, `NONE`)	`FLAT`
shininess	shininess parameter (range 0 to 128)	32
specular	specular color components	`null`
faceStyle	which polygonal faces are drawn (`FRONT`, `BACK`, `FRONT_AND_BACK`, `NONE`)	`FRONT`
faceColor	diffuse color for drawing faces	`GRAY`
backColor	diffuse color used for the backs of faces. If `null`, `faceColor` is used.	`null`
drawEdges	hint that polygon edges should be drawn explicitly	`false`
colorMap	color mapping properties (see Section 4.3.3)	`null`
normalMap	normal mapping properties (see Section 4.3.3)	`null`
bumpMap	bump mapping properties (see Section 4.3.3)	`null`
edgeColor	diffuse color for edges	`null`
edgeWidth	edge width in pixels	1
lineStyle:	how lines are drawn (`CYLINDER`, `LINE`, or `SPINDLE`)	`LINE`
lineColor	diffuse color for lines	`GRAY`
lineWidth	width in pixels when `LINE` style is selected	1
lineRadius	radius when `CYLINDER` or `SPINDLE` style is selected	1
pointStyle	how points are drawn (`SPHERE` or `POINT`)	`POINT`
pointColor	diffuse color for points	`GRAY`
pointSize	point size in pixels when `POINT` style is selected	1
pointRadius	sphere radius when `SPHERE` style is selected	1

Table 4.2: Render properties and their default values.

```
setRenderProps (RenderProps props); // set render properties
RenderProps getRenderProps();       // get render properties (read-only)
```

Render properties can also be set in the GUI by selecting one or more components and the choosing Set render props ... in the right-click context menu. More details on setting render properties through the GUI can be found in the section "Render properties" in the ArtiSynth User Interface Guide.

For many components, the default value of `renderProps` is `null`; i.e., no `RenderProps` object is assigned by default and render properties are instead inherited from ancestor components further up the hierarchy. The reason for this is because `RenderProps` objects are fairly large (many kilobytes), and so assigning a unique one to every component could consume too much memory. Even when a `RenderProps` object is assigned, most of its properties are inherited by default, and so only those properties which are explicitly set will differ from those specified in ancestor components.

4.3.1 Render property taxonomy

In general, the properties in `RenderProps` are used to control the color, size, and style of the three primary rendering primitives: faces, lines, and points. Table 4.2 contains a complete list. Values for the `shading`, `faceStyle`, `lineStyle` and `pointStyle` properties are defined using the following enumerated types: Renderer.Shading, Renderer.FaceStyle, Renderer.PointStyle, and Renderer.LineStyle. Colors are specified using `java.awt.Color`.

To increase and improve their visibility, both the line and point primitives are associated with styles (`CYLINDER`, `SPINDLE`, and `SPHERE`) that allow them to be rendered using 3D surface geometry.

Exactly how a component interprets its render properties is up to the component (and more specifically, up to the rendering method for that component). Not all render properties are relevant to all components, particularly if the rendering does not use all of the rendering primitives. For example, Particle components use only the point primitives and AxialSpring components use only the line primitives. For this reason, some components use subclasses of `RenderProps`, such as PointRenderProps and LineRenderProps, that expose only a subset of the available render properties. All renderable components provide the method createRenderProps() that will create and return a `RenderProps` object suitable for that component.

4.3.2 Setting render properties

When setting render properties, it is important to note that the value returned by getRenderProps() should be treated as *read-only* and should *not* be used to set property values. For example, applications should *not* do the following:

```
particle.getRenderProps().setPointColor (Color.BLUE);
```

This can cause problems for two reasons. First, `getRenderProps()` will return `null` if the object does not currently have a `RenderProps` object. Second, because `RenderProps` objects are large, ArtiSynth may try to share them between components, and so by setting them for one component, the application my inadvertently set them for other components as well.

Instead, `RenderProps` provides a static method for each property that can be used to set that property's value for a specific component. For example, the correct way to set `pointColor` is

```
RenderProps.setPointColor (particle, Color.BLUE);
```

One can also set render properties by calling setRenderProps() with a predefined `RenderProps` object as an argument. This is useful for setting a large number of properties at once:

```
RenderProps props = new RenderProps();
props.setPointColor (Color.BLUE);
props.setPointRadius (2);
props.setPointStyle (RenderProps.PointStyle.SPHERE);

...

particle.setRenderProps (props);
```

For setting each of the color properties within `RenderProps`, one can use either `Color` objects or `float[]` arrays of length 3 giving the RGB values. Specifically, there are methods of the form

```
props.setXXXColor (Color color)
props.setXXXColor (float[] rgb)
```

as well as the static methods

```
RenderProps.setXXXColor (Renderable r, Color color)
RenderProps.setXXXColor (Renderable r, float[] rgb)
```

where `XXX` corresponds to `Point`, `Line`, `Face`, `Edge`, and `Back`. For `Edge` and `Back`, both `color` and `rgb` can be given as `null` to clear the indicated color. For the specular color, the associated methods are

```
props.setSpecular (Color color)
props.setSpecular (float[] rgb)
RenderProps.setSpecular (Renderable r, Color color)
RenderProps.setSpecular (Renderable r, float[] rgb)
```

> Note that even though components may use a subclass of `RenderProps` internally, one can always use the base `RenderProps` class to set values; properties which are not relevant to the component will simply be ignored.

Finally, as mentioned above, render properties are inherited. Values set high in the component hierarchy will be inherited by descendant components, unless those descendants (or intermediate components) explicitly set overriding values. For example, a `MechModel` maintains its own `RenderProps` (and which is never null). Setting its `pointColor` property to `RED` will cause *all* point-related components within that `MechModel` to be rendered as red *except* for components that set their `pointColor` to a different property.

There are typically three levels in a `MechModel` component hierarchy at which render properties can be set:

- The `MechModel` itself;
- Lists containing components;

- Individual components.

For example, consider the following code:

```
MechModel mech = new MechModel ("mech");

Particle p1 = new Particle (/*name=*/null, 2, 0, 0, 0);
Particle p2 = new Particle (/*name=*/null, 2, 1, 0, 0);
Particle p3 = new Particle (/*name=*/null, 2, 1, 1, 0);

mech.addParticle (p1);
mech.addParticle (p2);
mech.addParticle (p3);

RenderProps.setPointColor (mech, Color.BLUE);
RenderProps.setPointColor (mech.particles(), Color.GREEN);
RenderProps.setPointColor (p3, Color.RED);
```

Setting the `MechModel` render property `pointColor` to `BLUE` will cause all point-related items to be rendered blue by default. Setting the `pointColor` render property for the particle list (returned by `mech.particles()`) will override this and cause all particles in the list to be rendered green by default. Lastly, setting `pointColor` for `p3` will cause it to be rendered as red.

4.3.3 Texture mapping

Render properties can also be set to apply texture mapping to objects containing polygonal meshes in which texture coordinates have been set. Supported is provided for color, normal and bump mapping, although normal and bump mapping are only available under the OpenGL 3 version of the ArtiSynth renderer.

Texture mapping is controlled through the colorMap, normalMap, and bumpMap properties of `RenderProps`. These are composite properties with a default value of `null`, but applications can set them to instances of ColorMapProps, NormalMapProps, and BumpMapProps, respectively, to provide the source images and parameters for the associated mapping. The two most important properties exported by all of these `MapProps` objects are:

enabled

A boolean indicating whether or not the mapping is enabled.

fileName

A string giving the file name of the supporting source image.

`NormalMapProps` and `BumpMapProps` also export scaling, which scales the x-y components of the normal map or the depth of the bump map. Other exported properties control mixing with underlying colors, and how texture coordinates are both filtered and managed when they fall outside the canonical range $[0, 1]$. Full details on texture mapping and its support by the ArtiSynth renderer are given in the "Rendering" section of the Maspack Reference Manual.

To set up a texture map, one creates an instance of the appropriate `MapProps` object and uses this to set either the colorMap, normalMap, or bumpMap property of `RenderProps`. For a specific renderable, the map properties can be set using the static methods

```
void RenderProps.setColorMap (Renderable r, ColorMapProps tprops);
void RenderProps.setNormalMap (Renderable r, NormalMapProps tprops);
void RenderProps.setBumpMap (Renderable r, BumpMapProps tprops);
```

When initializing the `PropMaps` object, it is often sufficient to just set enabled to `true` and `fileName` to the full path name of the source image. Normal and bump maps also often require adjustment of their scaling properties. The following static methods are available for setting the enabled and fileName subproperties within a renderable:

```
void RenderProps.setColorMapEnabled (Renderable r, boolean enabled);
void RenderProps.setColorMapFileName (Renderable r, String fileName);

void RenderProps.setNormalMapEnabled (Renderable r, boolean enabled);
```

```
void RenderProps.setNormalMapFileName (Renderable r, String fileName);

void RenderProps.setBumpMapEnabled (Renderable r, boolean enabled);
void RenderProps.setBumpMapFileName (Renderable r, String fileName);
```

Normal and bump mapping only work under the OpenGL 3 version of the ArtiSynth viewer, and also do not work if the shading property of `RenderProps` is set to `NONE` or `FLAT`.

Texture mapping properties can be set within ancestor nodes of the component hierarchy, to allow file names and other parameters to be propagated throughout the hierarchy. However, when this is done, it is still necessary to ensure that the corresponding mapping properties for the relevant descendants are non-`null`. That's because mapping properties themselves are not inherited; only their subproperties are. If a mapping property for any given object is `null`, the associated mapping will be disabled. A non-`null` mapping property for an object will be created automatically by calling one of the `setXXXEnabled()` methods listed above. So when setting up ancestor-controlled mapping, one may use a construction like this:

```
RenderProps.setColorMap (ancestor, tprops);
RenderProps.setColorMapEnabled (descendant0, true);
RenderProps.setColorMapEnabled (descendant1, true);
```

Then colorMap sub-properties set within `ancestor` will be inherited by `descendant0` and `descendant1`.

As indicated above, texture mapping will only be applied to components containing rendered polygonal meshes for which appropriate texture coordinates have been set. Determining such texture coordinates that produce appropriate results for a given source image is often non-trivial; this so-called "u-v mapping problem" is difficult in general and is highly dependent on the mesh geometry. ArtiSynth users can handle the problem of assigning texture coordinates in several ways:

- Use meshes which already have appropriate texture coordinates defined for a given source image. This generally means that mesh is specified by a file that contains the required texture coordinates. The mesh should then be read from this file (Section 2.5.5) and then used in the construction of the relevant components. For example, the application can read in a mesh containing texture coordinates and then use it to create a `RigidBody` via the method RigidBody.createFromMesh().
- Use a simple mesh object with predefined texture coordinates. The class MeshFactory provides the methods

  ```
  PolygonalMesh createRectangle (width, height, xdivs, ydivs, addTextureCoords);

  PolygonalMesh createSphere (radius, nslices, nlevels, addTextureCoords)
  ```

  which create rectangular and spherical meshes, along with canonical canonical texture coordinates if `addTextureCoords` is `true`. Coordinates generated by `createSphere()` are defined so that $(0,0)$ and $(1,1)$ map to the spherical coordinates $(-\pi,\pi)$ (at the south pole) and $(\pi,0)$ (at the north pole). Source images can be relatively easy to find for objects with canonical coordinates.
- Compute custom texture coordinates and set them within the mesh using setTextureCoords().

An example where texture mapping is applied to spherical meshes to make them appear like tennis balls is defined in

`artisynth.demos.tutorial.SphericalTextureMapping`

and listing for this is given below:

```
1 package artisynth.demos.tutorial;
2
3 import java.awt.Color;
4
5 import maspack.geometry.*;
```

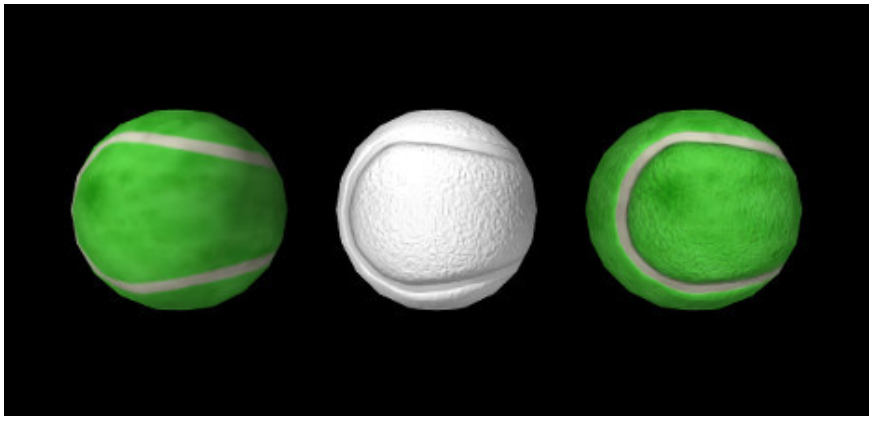


Figure 4.1: Color and bump mapping applied to spherical meshes. Left: color mapping only. Middle: bump mapping only. Right: combined color and bump mapping.

```
import maspack.matrix.RigidTransform3d;
import maspack.render.*;
import maspack.render.Renderer.ColorMixing;
import maspack.render.Renderer.Shading;
import maspack.util.PathFinder;
import maspack.spatialmotion.SpatialInertia;
import artisynth.core.mechmodels.*;
import artisynth.core.workspace.RootModel;

/**
 * Simple demo showing color and bump mapping applied to spheres to make them
 * look like tennis balls.
 */
public class SphericalTextureMapping extends RootModel {

   RigidBody createBall (
      MechModel mech, String name, PolygonalMesh mesh, double xpos) {
      double density = 500;
      RigidBody ball =
         RigidBody.createFromMesh (name, mesh.clone(), density, /*scale=*/1);
      ball.setPose (new RigidTransform3d (/*x,y,z=*/xpos, 0, 0));
      mech.addRigidBody (ball);
      return ball;
   }

   public void build (String[] args) {

      // create MechModel and add to RootModel
      MechModel mech = new MechModel ("mech");
      addModel (mech);

      double radius = 0.0686;
      // create the balls
      PolygonalMesh mesh = MeshFactory.createSphere (
         radius, 20, 10, /*texture=*/true);

      RigidBody ball0 = createBall (mech, "ball0", mesh, -2.5*radius);
      RigidBody ball1 = createBall (mech, "ball1", mesh, 0);
      RigidBody ball2 = createBall (mech, "ball2", mesh, 2.5*radius);

      // set up the basic render props: no shininess, smooth shading to enable
      // bump mapping, and an underlying diffuse color of white to combine with
      // the color map
      RenderProps.setSpecular (mech, Color.BLACK);
      RenderProps.setShading (mech, Shading.SMOOTH);
```

```
51        RenderProps.setFaceColor (mech, Color.WHITE);
52        // create and add the texture maps (provided free courtesy of
53        // www.robinwood.com).
54        String dataFolder = PathFinder.expand (
55           "${srcdir SphericalTextureMapping}/data");
56
57        ColorMapProps cprops = new ColorMapProps();
58        cprops.setEnabled (true);
59        // no specular coloring since ball should be matt
60        cprops.setSpecularColoring (false);
61        cprops.setFileName (dataFolder + "/TennisBallColorMap.jpg");
62
63        BumpMapProps bprops = new BumpMapProps();
64        bprops.setEnabled (true);
65        bprops.setScaling ((float)radius/10);
66        bprops.setFileName (dataFolder + "/TennisBallBumpMap.jpg");
67
68        // apply color map to balls 0 and 2. Can do this by setting color map
69        // properties in the MechModel, so that properties are controlled in one
70        // place - but we must then also explicitly enable color mapping in
71        // the surface mesh components for balls 0 and 2.
72        RenderProps.setColorMap (mech, cprops);
73        RenderProps.setColorMapEnabled (ball0.getSurfaceMeshComp(), true);
74        RenderProps.setColorMapEnabled (ball2.getSurfaceMeshComp(), true);
75
76        // apply bump map to balls 1 and 2. Again, we do this by setting
77        // the render properties for their surface mesh components
78        RenderProps.setBumpMap (ball1.getSurfaceMeshComp(), bprops);
79        RenderProps.setBumpMap (ball2.getSurfaceMeshComp(), bprops);
80     }
81  }
```

The `build()` method uses the internal method `createBall()` to generate three rigid bodies, each defined using a spherical mesh that has been created with MeshFactory.createSphere() with `addTextureCoords` set to `true`. The remainder of the `build()` method sets up the render properties and the texture mappings. Two texture mappings are defined: a color mapping and bump mapping, based on the images `TennisBallColorMap.jpg` and `TennisBallBumpMap.jpg` (Figure 4.2), both located in the subdirectory `data` relative to the demo source file. PathFinder.expand() is used to determine the full data folder name relative to the source directory. For the bump map, it is important to set the scaling property to adjust the depth amplitude to relative to the sphere radius.

Color mapping is applied to balls 0 and 2, and bump mapping to balls 1 and 2. This is done by setting color map and/or bump map render properties in the components holding the actual meshes, which in this case is the mesh components for the balls' surfaces meshes, obtained using `getSurfaceMeshComp()`. As mentioned above, it is also possible to set these render properties in an ancestor component, and that is done here by setting the colorMap render property of the `MechModel`, but then it is also necessary to enable color mapping within the individual mesh components, using `RenderProps.setColorMapEnabled()`.

To run this example in ArtiSynth, select All demos > tutorial > SphericalTextureMapping from the Models menu. The model should load and initially appear as in Figure 4.1. Note that if ArtiSynth is run with the legacy OpenGL 2 viewer (command line option `-GLVersion 2`), bump mapping will not be supported and will not appear.

4.4 Point-to-point muscles

Point-to-point muscles are a simple type of component in biomechanical models that provide muscle-activated forces acting along a line between two points. ArtiSynth provides this through Muscle, which is a subclass of AxialSpring that generates an active muscle force in response to its `excitation` property. The excitation property can be set and queried using the methods

```
setExcitation (double a);
double getExcitation();
```

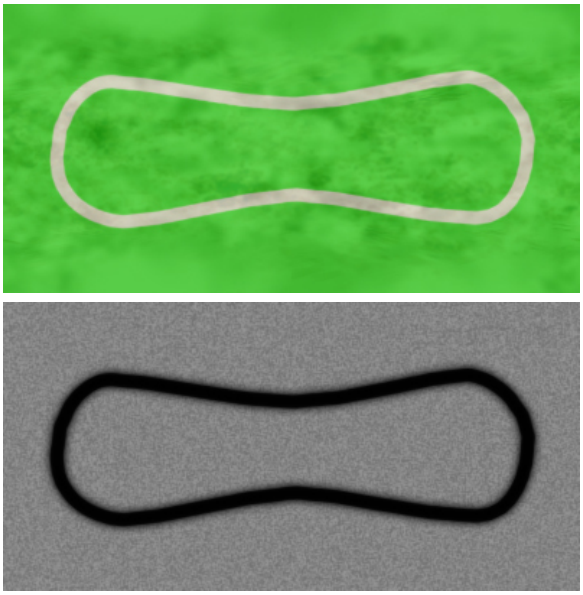


Figure 4.2: Color and bump map images used in the texture mapping example. These map to spherical coordinates on the mesh.

4.4.1 Muscle materials

As with `AxialSprings`, `Muscle` components use an AxialMaterial to compute the applied force $f(l,\dot{l},a)$ in response to the muscle's length l, length velocity $\dot{l}$, and excitation signal a. Usually the force is the sum of a *passive* component plus an *active* component that arises in response to the excitation signal.

The default `AxialMaterial` for a `Muscle` is SimpleAxialMuscle, which is essentially an activated version of LinearAxialMaterial and which computes a simple force according to

$$f(l,\dot{l}) = k(l-l_0) + d\dot{l} + m_f a \tag{4.1}$$

where k and d are stiffness and damping terms, a is the excitation value, and m_f is the maximum excitation force. k, d and m_f are exposed through the properties `stiffness`, `damping`, and `maxForce`.

More complex muscle materials are typically used for biomechanical modeling applications, generally with nonlinear passive terms and active terms that depend on the muscle length l. Some of those available in ArtiSynth include ConstantAxialMuscle, BlemkerAxialMuscle, MasoudMillardLAM, and PeckAxialMuscle.

ConstantAxialMuscle

This is a simple muscle material that has a contractile force proportional to its activation, a constant passive tension, and linear damping, as follows:

$$f(\dot{l}) = a f_{max} + f_{passive} f_{max} + d\dot{l} \tag{4.2}$$

The parameters for this material include:

	Property	Description
f_{max}	`maxForce`	the maximum contractile force
$f_{passive}$	`passiveFraction`	the proportion of f_{max} to apply as passive tension
d	`damping`	damping
a	`Muscle:excitation`	the `Muscle`'s activation

LinearAxialMuscle

This is a simple muscle material that has a linear relationship between length and tension, as well as linear damping, as follows:

$$l' = \frac{l - l_{opt}}{l_{max} - l_{opt}}, \text{with } 0 \leq l' \leq 1 \text{ enforced} \tag{4.3}$$

$$f(l, \dot{l}) = a f_{max} l' + f_{passive} f_{max} l' + d\dot{l} \tag{4.4}$$

The parameters for this material include:

	Property	Description
l_{max}	`maxLength`	the length at which maximum force is generated
l_{opt}	`optLength`	the length at which zero active force is generate
f_{max}	`maxForce`	the maximum contractile force
$f_{passive}$	`passiveFraction`	the proportion of f_{max} to apply as passive tension
d	`damping`	damping
a	`Muscle:excitation`	the `Muscle`'s activation

BlemkerAxialMuscle

This muscle material generates a force as described in [2]. It is the axial muscle equivalent to the constitutive equation along the muscle fiber direction specified in the BlemkerMuscle FEM material.

PeckAxialMuscle

This muscle material generates a force as described in [14]. It has a typical Hill-type active force-length relationship (modeled as a cosine), but the passive force-length properties are linear. This muscle model was empirically verified for jaw muscles during wide jaw opening [14].

MasoudMillardLAM

This muscle material generates a force as described in [11] for the rigid tendon case.

4.4.2 Example: Muscle attached to a rigid body

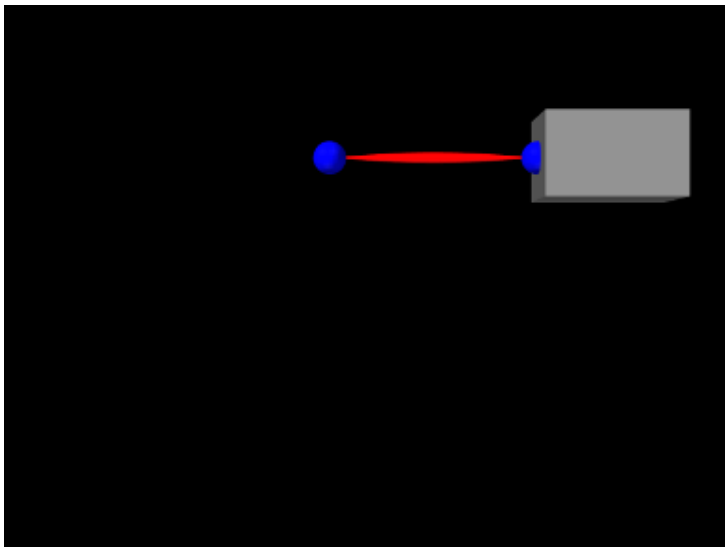


Figure 4.3: SimpleMuscle model loaded into ArtiSynth.

A simple model showing a single muscle connected to a rigid body is defined in

```
artisynth.demos.tutorial.SimpleMuscle
```

This model is identical to `RigidBodySpring` described in Section 3.2.2, except that the code to create the spring is replaced with code to create a muscle with a `SimpleAxialMuscle` material:

```
        // create the muscle:
        muscle = new Muscle ("mus", /*restLength=*/0);
        muscle.setPoints (p1, mkr);
        muscle.setMaterial (
           new SimpleAxialMuscle (/*stiffness=*/20, /*damping=*/10, /*maxf=*/10));
```

Also, so that the muscle renders differently, the rendering style for lines is set to `SPINDLE` using the convenience method

```
        RenderProps.setSpindleLines (muscle, 0.02, Color.RED);
```

To run this example in ArtiSynth, select All demos > tutorial > SimpleMuscle from the Models menu. The model should load and initially appear as in Figure 4.3. Running the model (Section 1.5.3) will cause the box to fall and sway under gravity. To see the effect of the `excitation` property, select the muscle in the viewer and then choose Edit properties ... from the right-click context menu. This will open an editing panel that allows the muscle's properties to be adjusted interactively. Adjusting the `excitation` property using the adjacent slider will cause the muscle force to vary.

4.5 Collision Handling

Collision handling in ArtiSynth is implemented by a collision handling mechanism build into `MechModel`. Collisions are disabled by default, but can be enabled between rigid and deformable bodies (finite element models in particular), and more generally between any body that implements the interface Collidable.

It is important to understand that collision handling is both computationally expensive and, due to it's discontinuous nature, less accurate than other aspects of the simulation. ArtiSynth therefore provides a number of ways to selectively control collision handling between different pairs of bodies.

Collision handling is also challenging because if collisions are enabled among n objects, then one needs to be able to easily specify the characteristics of up to $O(n^2)$ collision interactions, while also managing the fact that these interactions are highly transient. In ArtiSynth, collision handling is done by a CollisionManager that is a sub-component of each MechModel. The collision manager maintains default collision behaviors among certain groups of collidable objects, while also allowing the user to override the default behaviors by setting override behaviors for specific component pairings.

4.5.1 Enabling collisions in code

Collisions can be enabled as either a default behavior between all bodies, a default behavior between certain *groups* of bodies, or a specific override behavior between individual pairs of bodies.

> This section describes how to enable collisions in code. However, it is also possible to set some aspects of collision behavior interactively within the ArtiSynth GUI. See the section "Collision handling" in the ArtiSynth User Interface Guide.

The default collision behavior between all collidables can be controlled using the method

```
    setDefaultCollisionBehavior (enabled, mu);
```

where `enabled` is `true` or `false` depending on whether collisions are enabled, and `mu` is the coefficient of Coulomb (or dry) friction. The `mu` value is ignored if `enabled` is `false`. `mu` can also be left undefined by specifying a value less than 0, in which case the friction coefficient is inherited from the global friction value accessed by the methods

```
    setFriction (mu);
    double getFriction();
```

The default collision behavior can also be controlled using the method

```
    setDefaultCollisionBehavior (behavior);
```

where `behavior` is a CollisionBehavior object that specifies both *enabled* and *mu*, along with other, more detailed collision properties (see Section 4.5.3).

In addition, a default collision behavior can be set for generic groups of collidables using

```
setDefaultCollisionBehavior (group0, group1, enabled, mu);
setDefaultCollisionBehavior (group0, group1, behavior);
```

where `group0` and `group1` are static instances of the special `Collidable` subclass Collidable.Group that represents the groups described in Table 4.3. The groups `Collidable.Rigid` and `Collidable.Deformable` denote collidables that are rigid (such as RigidBody) or deformable (such as FEM models like FemModel3d). (If a collidable is deformable, then its isDeformable() method returns `true`.) The group `Collidable.AllBodies` denotes both rigid and deformable bodies, and `Collidable.Self` is used to enabled self-collision. which is described in greater detail in Section 4.5.4. `mu` again is ignored if `enabled` is `false`, and if `mu` is less than zero, the friction is undefined and inherited from the global value accessed using `setFriction(mu)` and `getFriction()`.

Collidable group	description
Collidable.Rigid	rigid collidables (e.g., rigid bodies)
Collidable.Deformable	deformable collidables (e.g, FEM models)
Collidable.AllBodies	rigid and deformable collidables
Collidable.Self	enables self-intersection for composite deformable collidables
Collidable.All	rigid and deformable collidables and self-intersection

Table 4.3: Collision group types.

A call to one of the `setDefaultCollisionBehavior()` methods will override the effects of previous calls. So for instance, the code sequence

```
setDefaultCollisionBehavior (true, 0);
setDefaultCollisionBehavior (Collidable.Deformable, Collidable.Rigid, false, 0);
setDefaultCollisionBehavior (true, 0.2);
```

will initially enable collisions between all bodies with a friction coefficient of 0, then *disable* collisions between deformable and rigid bodies, and finally re-enable collisions between all bodies with a friction coefficient of 0.2.

The default collision behavior between any pair of collidable groups can be queried using

```
CollisionBehavior getDefaultCollisionBehavior (group0, group1);
```

For this call, `group0` and `group1` are restricted to the primary groups `Collidable.Rigid`, `Collidable.Deformable`, and `Collidabe.Self`, since individual behaviors are not maintained for the composite groups `Collidable.AllBodies` and `Collidable.All`.

In addition to default behaviors, overriding behaviors for individual collidables or pairs of collidables can be controlled using

```
setCollisionBehavior (collidable0, collidable1, enabled, mu);
setCollisionBehavior (collidable0, collidable1, behavior);
```

where `collidable0` is an individual collidable component (such as a rigid body or FEM model), and `collidable1` is either another individual component, or one of the groups of Table 4.3. As with default behaviors, `mu` is ignored if `enabled` is `false`, and if `mu` is less than zero, the friction is undefined and inherited from the global value accessed using `setFriction(mu)` and `getFriction()`.

As an example of setting individual collision behaviors, the calls

```
RigidBody bodA;
FemModel3d femB;

setCollisionBehavior (bodA, Collidable.Deformable, true, 0.1);
setCollisionBehavior (femB, Collidable.AllBodies, true, 0.0);
setCollisionBehavior (bodA, femB, false, 0.0);
```

will enable collisions between `bodA` and all deformable collidables (with friction 0.1), as well as `femB` and all deformable and rigid collidables (with friction 0.0), while specifically `disabling` collisions between `bodA` and `femB`.

The `setCollisionBehavior()` methods work by adding a CollisionBehavior object (Section4.5.3) to the collision manager as a sub-component. With `setCollisionBehavior(collidable0,collidable1,behavior)`, the behavior object is created and supplied by the application. With `setCollisionBehavior(enable,mu)`, the behavior object is created automatically and returned by the method. Once an override behavior has been specified, then it can be queried using

```
getCollisionBehavior (collidable0, collidable1);
```

This method will return `null` if no override behavior for the pair in questions has been previously set using one of the `setCollisionBehavior()` methods.

> Because behaviors are proper components, it is *not* permissible to add them to the collision manager twice. Specifically, the following will produce an error:
>
> ```
> CollisionBehavior behav = new CollisionBehavior();
> behav.setDrawIntersectionContours (true);
> mesh.setCollisionBehavior (col0, col1, behav);
> mesh.setCollisionBehavior (col2, col3, behav); // ERROR
> ```
>
> However, if desired, a new behavior can be created from an existing one:
>
> ```
> CollisionBehavior behav = new CollisionBehavior();
> behav.setDrawIntersectionContours (true);
> mesh.setCollisionBehavior (col0, col1, behav);
> behav = new CollisionBehavior(behav);
> mesh.setCollisionBehavior (col2, col3, behav); // OK
> ```

To determine the collision behavior (default or override) that actually controls a specific pair of collidables, one may use the method

```
getActingCollisionBehavior (collidable0, collidable1);
```

where `collidable0` and `collidable1` must both be specific collidable components and cannot be a group (such as `Collidable.Rigid` or `Collidable.All`). If one of the collidables is a *compound* collidable (Section 4.5.4), or has a collidability setting (Section 4.5.5) that prevents collisions, there may be no consistent acting behavior, in which case the method returns `null`.

Collision behaviors take priority over each other in the following order:

1. behaviors specified using `setCollisionBehavior()` involving two *specific* collidables.
2. behaviors specified using `setCollisionBehavior()` involving one *specific* collidable and a *group* of collidables (indicated by a `Collidable.Group`), with later specifications taking priority over earlier ones.
3. default behaviors specified using `setDefaultCollisionBehavior()`.

An override behavior specified with `setCollisionBehavior()` can later be removed using

```
clearCollisionBehavior (collidable0, collidable1);
```

and *all* override behaviors in a `MechModel` can be removed with

```
clearCollisionBehaviors ();
```

Note that this latter call does *not* remove default behaviors specified with `setDefaultCollisionBehavior()`.

> Note: It is usually necessary to ensure that collisions are *disabled* between adjacent bodies connected by joints, since otherwise these would be forced into a state of permanent collision.

4.5.2 Example: Collision with a plane

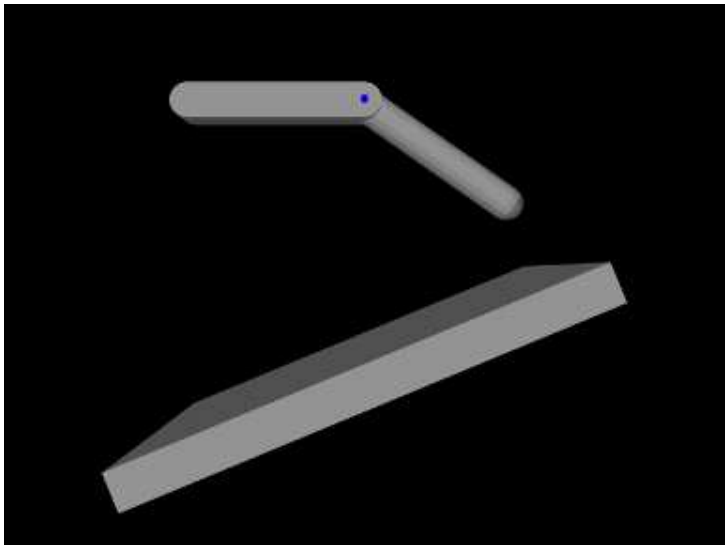


Figure 4.4: JointedCollide model loaded into ArtiSynth.

A simple model illustrating collision between two jointed rigid bodies and a plane is defined in

```
artisynth.demos.tutorial.JointedCollide
```

This model is simply a subclass of `RigidBodyJoint` that overrides the `build()` method to add an inclined plane and enable collisions between it and the two connected bodies:

```
1    public void build (String[] args) {
2
3       super.build (args);
4
5       bodyB.setDynamic (true);  // allow bodyB to fall freely
6
7       // create and add the inclined plane
8       RigidBody base = RigidBody.createBox ("base", 25, 25, 2, 0.2);
9       base.setPose (new RigidTransform3d (5, 0, 0, 0, 1, 0, -Math.PI/8));
10      base.setDynamic (false);
11      mech.addRigidBody (base);
12
13      // turn on collisions
14      mech.setDefaultCollisionBehavior (true, 0.20);
15      mech.setCollisionBehavior (bodyA, bodyB, false);
16   }
```

The superclass `build()` method called at line 3 creates everything contained in `RigidBodyJoint`. The remaining code then alters that model: `bodyB` is set to be dynamic (line 5) so that it will fall freely, and an inclined plane is created from a thin box that is translated and rotated and then set to be be non-dynamic (lines 8-11). Finally, collisions are enabled by setting the default collision behavior (line 14), and then specifically disabling collisions between `bodyA` and `bodyB` (line 15). As indicated above, the latter step is necessary because the joint would otherwise keep the two bodies in a permanent state of collision.

To run this example in ArtiSynth, select All demos > tutorial > JointedCollide from the Models menu. The model should load and initially appear as in Figure 4.4. Running the model (Section 1.5.3) will cause the jointed assembly to collide with and slide off the inclined plane.

4.5.3 Collision behaviors

As mentioned above, the CollisionBehavior objects described above can be used to control other aspects of the contact and collision beyond friction and enabling. In particular, a `CollisionBehavior` exports the following properties:

enabled

A boolean that determines if collisions are enabled.

friction

A double giving the coefficient of Coulomb friction, typically in the range $[0, 0.5]$. The default value is 0. Setting friction to a non-zero value increases the simulation time, since extra constraints must be added to the system to accommodate the friction.

penetrationTol

A double controlling the amount of interpenetration that is permitted in order to ensure contact stability (see Section 4.6.4). If not specified, the system will inherit this property from the `MechModel`, which computes a default penetration tolerance based on the overall size of the model.

compliance

A double which adds a compliance (inverse stiffness) to the collision behavior, so that the contact has a "springiness". The default value for this is 0 (no compliance). See Section 4.6.3.

damping

A double which, if compliance is non-zero, specifies a damping to accompany the compliant behavior. The default value is 0. When compliance is specified, it is usually necessary to set the damping to a non-zero value to prevent bouncing. See Section 4.6.3.

collisionPointTol

A double which, for contacts between rigid bodies, specifies a minimum distance between contact points that is used to reduce the number of contacts. If not explicitly specified, the system computes a default value for this based on the overall size of the model.

reduceConstraints

A boolean which, if `true`, indicates that the system should try to reduce the number of contacts between deformable bodies with limited degrees of freedom, so that the resulting contact problem is not ill-conditioned. The default value is `false`. See Section 4.6.3.

method

An instance of CollisionBehavior.Method that controls how the contact constraints for the collision response are generated. There are several methods available, and some are experimental. The more standard methods are:

Method	Constraint generation
VERTEX_PENETRATION	constraints generated from interpenetrating vertices
CONTOUR_REGION	constraints generated from planes fit to each mesh contact region
INACTIVE	no constraints generated

These are described in more detail in Section 4.6.1.

colliderType

An instance of CollisionManager.ColliderType that specifies the underlying mechanism used to determine the collision information between two meshes. The choice of collider may restrict which collision *methods* (described above) are allowed. Collider types available at present include:

Type	Description
AJL_CONTOUR	uses mesh intersection contour to find penetrating regions and vertices
TRI_INTERSECTION	uses triangle intersections to find penetrating regions and vertices
SIGNED_DISTANCE	uses a signed distance field to find penetrating vertices

These are described in more detail in Section 4.6.1.

In addition to the above, CollisionBehavior exports other properties that control the rendering of collisions (Section 4.6.5).

It should be noted that most collision behavior properties are also properties of the MechModel's collision manager, which can be accessed in code using getCollisionManager() (or graphically via the navigation panel). Since collision behaviors are sub-components of the collision manager, properties set in the collision manager will be inherited by any behaviors for which they have not been explicitly set. This is the easiest way to specify behavior properties generally, as for example:

```
CollisionManager cm = mech.getCollisionManager();
cm.setReduceConstraints (true);
```

Some other properties, like penetrationTol, are properties not of the collision manager, but of the MechModel, and so can be set globally there.

To set collision properties in a more fine-grained manner, one can either call setDefaultCollisionBehavior() or setCollisionBehavior() with an appropriately set CollisionBehavior object:

```
RigidBody bodA;

CollisionBehavior behav = new CollisionBehavior (enabled, mu);
behav.setPenetrationTol (0.001);
setDefaultCollisionBehavior (Collidable.Deformable, Collidable.Rigid, behav);

behav.setPenetrationTol (0.003);
setCollisionBehavior (bodA, Collidable.Rigid, behav);
```

For behaviors that are already set, one may use getDefaultCollisionBehavior() or getCollisionBehavior() to obtain the behavior and then set the desired properties directly:

```
RigidBody bodA;
CollisionBehavior behav;

behav = getDefaultCollisionBehavior (Collidable.Deformable, Collidable.Rigid);
behav.setPenetrationTol (0.001);
behav = getCollisionBehavior (bodA, Collidable.Rigid);
behav.setPenetrationTol (0.003);
```

Note however that getDefaultCollisionBehavior() only works for Collidable.Rigid, Collidable.Deformable, and Collidable.Self, and that getCollisionBehavior() only works for a collidable pair that has been previously specified with setCollisionBehavior(). One may also use getActingCollisionBehavior() (described above) to obtain the behavior (default or otherwise) responsible for a specific pair of collidables, although in some instances no such single behavior exists and the method will then return null.

There are two constructors for CollisionBehavior:

```
CollisionBehavior ();
CollisionBehavior (boolean enable, double mu);
```

The first creates a behavior with the enabled property set to false and the other properties set to their default (generally inherited) values. The second creates a behavior with the enabled and friction properties explicitly specified by enabled and mu, and other properties set to their default values. If mu is less than zero or enabled is false, then the friction property is set to be inherited so that its value is controlled by the global friction property in the collision manager (and accessed using the MechModel methods setFriction(mu) and getFriction()).

4.5.4 Self-collision and collidable hierarchies

At present, ArtiSynth does not support the detection or handling of self-collision within single meshes. However, self-collision can still be effected by allowing a collidable to have multiple *sub-collidables* and then enabling collisions between some or all of these.

Any descendant component of a Collidable A which is itself Collidable is considered to be a sub-collidable of A. Certain types of components maintain sub-collidables by default. For example, some components (such as finite element

models; Section 6) maintain a list of meshes in a child component list named `meshes`; these can be used to implement self-collision as described below. A collidable that contains sub-collidables is known as a *compound* collidable, and its isCompound() method will return `true`. Likewise, if a component is a sub-collidable, then its getCollidableAncestor()) method will return its nearest collidable ancestor.

> A collidable does not need to be an immediate child component of a collidable A in order to be a sub-collidable of A; it need only be a descendant of A.

> At present, collidable hierarchies are assumed to have a depth no greater than one (i.e., no sub-collidable is itself a compound collidable), and also sub-collidables are assumed to have the same type (rigid or deformable) as the ancestor.

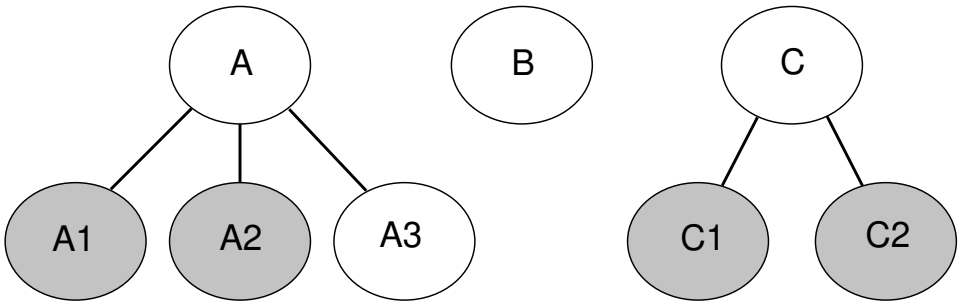


Figure 4.5: A collection of collidable components, where A possesses sub-collidables A1, A2, and A3, B is solitary, and C possesses sub-collidables C1 and C2. Internal collisions are enabled among those sub-collidables which are shaded gray.

In general, an ArtiSynth model will contain a collection of collidables, some of which are compound and others which are solitary (Figure 4.5). Within a collection of collidables:

- Actual collisions happen only between leaf collidables; compound collidables are used only for grouping purposes.
- Leaf collidables are also instances of the sub-interface CollidableBody, which provides the additional information needed to actually determine collisions and compute the response (Section 4.6).
- By default, the sub-collidables of a compound component A will only collide among themselves if self-collision is specified for A (via either a default or override collision behavior). If self-collision is specified for A, then collisions may occur only among those sub-collidables for which *internal* collisions are enabled. Internal collisions are enabled for a collidable if its collidable property (Section 4.5.5) is set to either `ALL` or `INTERNAL`.
- Self-collision is also only possible among the sub-collidables of A if A is itself deformable; i.e., its isDeformable() method returns `true`.
- Sub-collidables may collide with collidables outside their hierarchy if their collidable property is set to either `ALL` or `EXTERNAL`.
- Collision among specific pairs of sub-collidables may also be explicitly enabled or disabled with an override behavior set using one of the `setCollisionBehavior()` methods.
- Specifying a collision behavior among two compound collidables A and B which are *not* within the same hierarchy will cause that behavior to be specified among all sub-collidables of A and B whose `collidable` property enables the collision.

Subject to the above conditions, self-collisions can be enabled among the sub-collidables of all deformable collidables by calling

```
setDefaultCollisionBehavior (Collidable.DEFORMABLE, Collidable.SELF, true, mu);
```

or, for a specific collidable `C`, by calling either

```
setDefaultCollisionBehavior (C, Collidable.SELF, true, mu);

... OR ...

setDefaultCollisionBehavior (C, C, true, mu);
```

For a more complete example, refer to Figure 4.5, assume that components A, B and C are deformable, and that self-collision is allowed among those sub-collidables which are shaded gray (A1 and A2 for A, B1 and B2 for B). Then:

```
// Set default collision among deformable components with friction = 0.2:
setDefaultCollisionBehavior (
   Collidable.DEFORMABLE, Collidable.DEFORMABLE, true, 0.2);
// Collisions are now enabled between A1, A2, and A3 and each of B, C1, and
// C2, and between B and C1 and C2, but not among A1, A2, and A3 or C1 and C2.

// Enable self-collision between A1 and A2 and B1 and B2 with friction = 0:
setDefaultCollisionBehavior (Collidable.DEFORMABLE, Collidable.SELF, true, 0);

// Specifically disable collision between B and A3:
setCollisionBehavior (B, A3, false);

// Specifically enable collision between A3 and C with friction = 0.3:
setCollisionBehavior (A3, C, true, 0.3);
// This behavior will be applied between A3 and each of C1 and C2.

// Disable self-collision within A:
setCollisionBehavior (A, A, false);
// This will disable all self-collisions among A1, A2 and A3.
```

4.5.5 Collidability

Each collidable component maintains a `collidable` property (which can be queried using getCollidable()) which specifically enables or disables the ability of that collidable to collide with other collidables.

The `collidable` property value is of the enumerated type Collidable.Collidability, which has four possible settings:

OFF

All collisions disabled: the collidable will not collide with anything.

INTERNAL

Internal (self) collisions enabled: the collidable may only collide with other Collidables with which it shares a common ancestor.

EXTERNAL

External collisions enabled: the collidable may only collide with other Collidables with which it does *not* share a common ancestor.

ALL

All collisions (both self and external) enabled: the collidable may collide with any other Collidable.

Note that collidability only *enables* collisions. In order for collisions to actually occur between two collidables, a default or override collision behavior must also be specified for them in the MechModel.

4.5.6 Collision response

Sometimes, an application may wish to know details about the collision interactions between a specific collidable and one or more other collidables. These details may include items such as contact forces or the penetration regions between the opposing meshes. Applications can track this information by creating CollisionResponse components for the collidables in question and adding them to the `MechModel` using setCollisionResponse():

```
CollisionResponse resp = new CollisionResponse();
mech.setCollisionResponse (collidable0, collidable1, resp);
```

An alternate version of `setCollisionResponse()` creates the response component internally and returns it:

```
CollisionResponse resp = mech.setCollisionResponse (collidable0, collidable1);
```

During every simulation step, the `MechModel` will update its response components to reflect the current collision conditions between the associated collidables.

The first collidable specified for a collision response must be a specific collidable component, while the second may be either another collidable or a group of collidables represented by a Collidable.Group:

```
CollisionResponse r0, r1, r2;
RigidBody bodA;
FemModel3d femB;

// collision information between bodA and femB only:
r0 = setCollisionResponse (bodA, femB);
// collision information between femB and all rigid bodies:
r1 = setCollisionResponse (femB, Collidable.Rigid);
// collision information between femB and all bodies and self-collisions:
r2 = setCollisionResponse (femB, Collidable.All);
```

When a compound collidable is specified, the response component will collect collision information for all its sub-collidables.

If desired, applications can also instantiate responses themselves and add them to the collision manager:

```
CollisionResponse resp = new CollisionResponse();
mech.setCollisionResponse (femA, femB, resp);
```

However, as with behaviors, the same response cannot be added to an application twice:

```
CollisionResponse resp = new CollisionResponse();
mech.setCollisionResponse (femA, femB, resp);
mech.setCollisionResponse (femC, femD, resp); // ERROR
```

The complete set of methods for managing collision responses are similar to those for behaviors,

```
setCollisionResponse (collidable0, collidable1, response);
setCollisionResponse (collidable0, collidable1);
getCollisionResponse (collidable0, collidable1);
clearCollisionResponse (collidable0, collidable1);
clearCollisionResponses ();
```

where getCollisionResponse() and clearCollisionResponse() respectively return and clear response components that have been previously set using one of the `setCollisionResponse()` methods.

Information that can be queried by a CollisionResponse component includes whether or not the collidable is in collision, contact forces acting on the vertices of the colliding meshes, the penetration regions of each mesh associated with the collision, and the underlying CollisionHandler objects that maintain the contact constraints between each colliding mesh. This information may be queried with the following methods:

```
// Queries if the collidables associated with this response are in contact
boolean inContact();

// Returns a map specifying the vertex-specific contact forces acting on all
// the deformable bodies associated with the first or second collidable, as
// specified by cidx=0 or cidx=1.
Map<Vertex3d,Vector3d> getContactForces(int cidx);

// Returns a map specifying the vertex-specific contact pressures acting on all
// the deformable bodies associated with the first or second collidable, as
// specified by cidx=0 or cidx=1.
```

```
   Map<Vertex3d,Vector3d> getContactForces(int cidx);

   // Returns a list of all the penetration regions on either the first
   // or second collidable, as specified by cidx=0 or cidx=1. Penetration
   // regions are available only if the collision manager's collider
   // type is set to AJL_CONTOUR.
   ArrayList<PenetrationRegion> getPenetrationRegions(int cidx);

   // Returns the CollisionHandlers for all currently active collisions
   // associated with the collidables of this response. Note that the
   // body ordering in the handlers may be reversed.
   ArrayList<CollisionHandler> getHandlers();
```

A typical usage scenario for collision responses is to create them before the simulation is run, possibly in the root model `build()` method, and then query them when the simulation is running, such from the `apply()` method of a `Monitor` (Section 5.3). For example, in the root model `build()` method, the response could be created with the call

```
CollisionResponse resp = mech.setCollisionResponse (femA, femB);
```

and then used in some runtime code as follows:

```
Map<Vertex3d,Vector3d> collMap = resp.getContactForces (0);
```

If for some reason it is difficult to store a reference to the response between its construction and its use, then `getCollisionResponse()` can be used to retrieve it:

```
CollisionResponse resp = mech.getCollisionResponse (femA, femB);
Map<Vertex3d,Vector3d> collMap = resp.getContactForces (0);
```

4.5.7 Nested MechModels

It is possible in ArtiSynth for one `MechModel` to contain other nested `MechModels` within its component hierarchy. This raises the question of how collisions within the nested models are controlled. The general rule for this is the following:

> The collision behavior for two collidables `colA` and `colB` is determined by whatever behavior (either default or override) is specified by the lowest `MechModel` in the component hierarchy that contains both `colA` and `colB`.

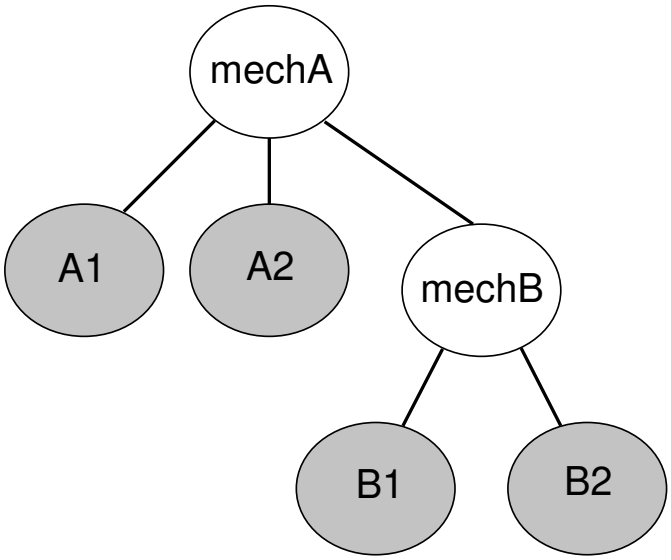


Figure 4.6: A `MechModel` containing collidables `B1` and `B2`, nested within another containing collidables `A1` and `A2`.

For example, consider Figure 4.6 where we have a `MechModel` (`mechB`) containing collidables `B1` and `B2`, nested within another `MechModel` (`mechA`) containing collidables `A1` and `A2`. Then consider the following code fragment:

```
mechB.setDefaultCollisionBehavior (true, 0.2);
mechA.setCollisionBehavior (B1, Collidable.AllBodies, true, 0.0);
mechA.setCollisionBehavior (B1, A1, false);
mechA.setCollisionBehavior (B1, B2, false); // Error!
```

The first line enables default collisions within `mechB` (with $\mu = 0$), controlling the interaction between `B1` and `B2`. However, collisions are still disabled within `mechA`, meaning `A1` and `A2` will not collide either with each other or with `B1` or `B2`. The second line enables collisions between `B1` and any other body within `mechA` (i.e., `A1` and `A2`), while the third line disables collisions between `B1` and `A1`. Finally, the fourth line results in an error, since `B1` and `B2` are both contained within `mechB` and so their collision behavior cannot be controlled from `mechA`.

While it is not legal to specify a specific behavior for collidables contained a `MechModel` from a higher level `MechModel`, it *is* legal to create collision response components for the same pair within both models. So the following code fragment would be allowed and would create response components in both `mechA` and `mechB`:

```
mechB.setCollisionResponse (B1, B2);
mechA.setCollisionResponse (B1, B2);
```

4.6 Collision Implementation and Rendering

As mentioned in Section 4.5.4, the leaf collidables which actually provide collision interaction are also instances of CollidableBody, which provides methods returning the information needed to compute collisions and their response. Some of these methods include:

```
PolygonalMesh getCollisionMesh();

boolean hasDistanceGrid();

DistanceGridComp getDistanceGridComp();
```

getCollisionMesh() returns the surface mesh which is used as the basis for collision. If hasDistanceGrid() returns `true`, then the body also maintains a signed distance grid for the mesh, which can be obtained using getDistanceGridComp() and is used by the collider type `SIGNED_DISTANCE` (Section 4.6.1).

4.6.1 Collision methods and collider types

The ArtiSynth collision mechanism works by using a *collider* (described further below) to find the intersections between the surface meshes of each collidable object. Information provided by the collider is then used to generate *contact constraints*, according to a *collision method*, that prevent further collision and resolve any existing interpenetration.

Collision methods are specified by collision behavior's method property, which is an instance of CollisionBehavior.Method, as mentioned in Section 4.5.3. Some of the standard methods are:

VERTEX_PENETRATION

A contact constraint is generated for each mesh vertex that interpenetrates the other mesh, with the normal direction determined by finding the nearest point on the opposing mesh. This method is the default for collisions involving deformable bodies which have enough degrees of freedom (DOF) that the resulting number of contacts does not overconstrain the system. Using this method for collisions between rigid bodies, or low DOF deformable bodies, will generally result in an overconstrained system unless the constraints are either regularized using compliance or constraint reduction is enabled (see Section 4.6.3).

CONTOUR_REGION

Contact constraints are generated by fitting a plane to each mesh contact region and then projecting the region onto that plane. Constraints are then created from points on the perimeter of this projection, with the normal direction being given by the plane normal. This is the default method for collision between rigid bodies or low DOF deformable bodies. Trying to use this method for FEM-based deformable bodies will result in an error.

DEFAULT

Selects the method most appropriate depending on the DOFs of the colliding bodies and whether they are rigid or deformable . This is the default setting.

INACTIVE

No constraints are generated. This will result in no collision response.

The contact information used by the collision method is generated by a collider. Colliders are described by instances of CollisionManager.ColliderType and can be specified by the colliderType property in either the collision manager (as the default), or in the collision behavior for a specific pair of collidables. Since different colliders may provide different collision information, some colliders may restrict the type of collision method that may be used.

Three collider types are presently supported:

AJL_CONTOUR

A bounding-box hierarchy is used to locate all triangle intersections between the two surface meshes, and the intersection points are then connected to find the (piecewise linear) intersection contours. The surface meshes must (at present) be triangular, closed, and manifold. The contours are then used to identity the contact regions on each surface mesh, which can be used to determine interpenetrating vertices and contact area. Intersection contours and the contact constraints generated from them are shown in Figure 4.7.

TRI_INTERSECTION

A legacy method which also uses a bounding-box hierarchy to locate all triangle intersections between the two surface meshes. However, contours are not generated. Instead, contact regions are estimated by grouping the intersection points together, and penetrating vertices are computed separately using point-mesh distance queries based on the bounding-box hierarchy. This latter step requires iterating over all mesh vertices, which may be slow for large meshes.

SIGNED_DISTANCE

Uses a grid-based signed distance field on one mesh to quickly determine the penetrating vertices of the opposite mesh, along with the penetration distance and normals. This is only available for collidable pairs where at least one of the bodies maintains a signed distance grid which can be obtained using getDistanceGridComp(). Advantages include speed (often an order of magnitude faster that the colliders based on triangle intersection) and the fact that the opposite mesh does *not* have to be triangular, closed, or manifold. However, signed distance fields can (at present) only be computed for fixed meshes, and so at least one colliding body must be rigid. The signed distance field also does not yet yield contact region information, and so the collision method is restricted to VERTEX_PENETRATION. Contacts generated from a signed distance field are illustrated in Figure Figure 4.8.

> Because the `SIGNED_DISTANCE` collider type currently supports only the `VERTEX_PENETRATION` method, its use between rigid bodies, or low DOF deformable bodies, will generally result in an overconstrained system unless the constraints are either regularized using compliance or constraint reduction is enabled. See Section 4.6.3.

4.6.2 Collision meshes and signed distance grids

As mentioned above, collidable bodies declare the method getCollisionMesh(), which returns a mesh defining the collision surface. This is either used directly, or, for `SIGNED_DISTANCE` collision handling, to compute a signed distance grid which is in turn used to handle collisions.

For RigidBody objects, the mesh returned by `getCollisionMesh()` is typically the same as the surface mesh. However, it is possible to change this, by adding additional meshes to the body and modifying mesh collidability (see Section 3.2.8). The collision mesh is then formed as the sum of all polygonal meshes in the body's `meshes` list whose collidable property is `true`.

> The *sum* operation used to create the `RigidBody` collision mesh uses addMesh(), which simply adds all vertices and faces together. While the result works correctly for collisions, it does not represent a proper CSG union operation (such as that described in Section 2.5.7) and may contain interpenetrating and overlapping features.

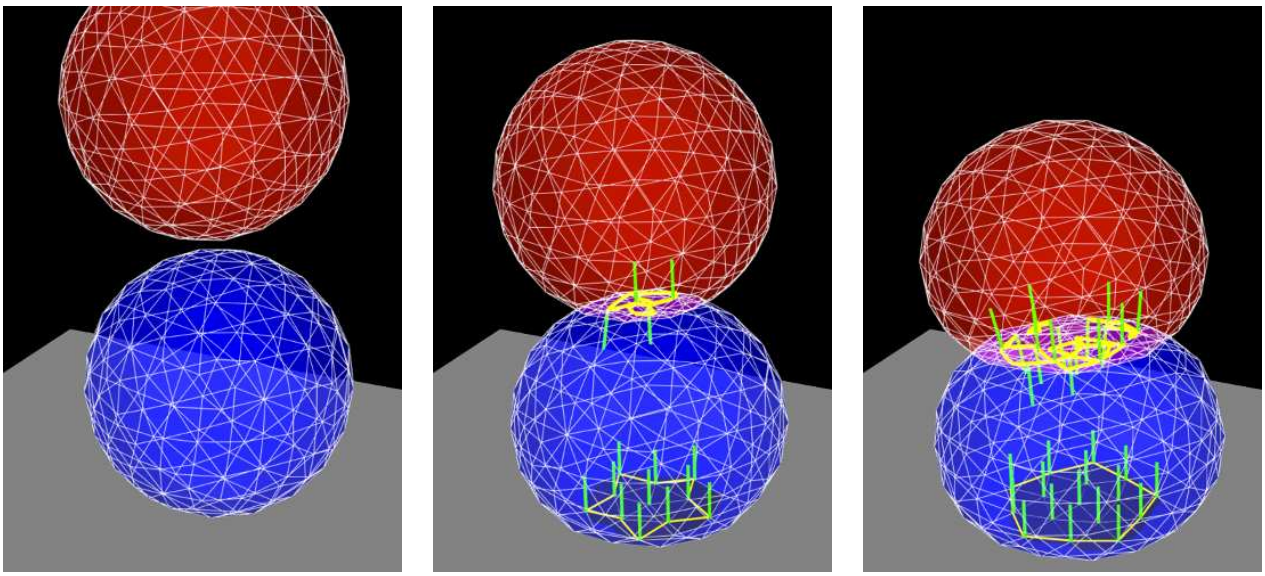


Figure 4.7: Time sequence of contact handling between two deformable models falling under gravity, showing the intersection contours (yellow) and the contact normals (green lines).

Collidable bodies also declare the methods hasDistanceGrid() and getDistanceGridComp(). If the former returns `true`, then the body has a distance grid and the latter returns a DistanceGridComp containing it. A distance grid is a regular 3D grid, with uniformly arranged vertices and cells, that is used to represent a scalar distance field, with distance values stored implicitly at each vertex and interpolated within cells. Distance grids are used for `SIGNED_DISTANCE` collision handling, and by default are generated on-demand, using an automatically chosen resolution and the mesh returned by getCollisionMesh() as the surface against which distances are computed.

When used for collision handling, values within a distance grid are interpolated trilinearly within each cell. This means that the effective collision surface is actually the trilinearly interpolated isosurface corresponding to a distance of 0. This surface will differ somewhat from the original surface returned by `getCollisionMesh()`, in a manner that depends on the grid resolution. Consequently, when using `SIGNED_DISTANCE` collision handling, it is important to be able to visualize the trilinear isosurface, and possibly modify it by adjusting the grid resolution. The `DistanceGridComp` returned by `getDistanceGridComp()` exports properties to facilitate both of these requirements, as described in detail in Section 4.7.

4.6.3 Overconstrained contact and regularization

An issue that can arise with the `VERTEX_PENETRATION` collision method (Section 4.6.1), which by default is employed for deformable bodies but can also be set for rigid bodies, is the problem of overconstrained contact, in which the number of contact constraints exceeds the number of degrees of freedom (DOF) available amongst the dynamic components for handling the contact. When this occurs, an error message will typically appear in the application's terminal output, such as

```
Pardiso: num perturbed pivots=12
```

and the simulation itself may go unstable.

Overconstrained contact does not typically occur with regular FEM models because the number of contact constraints does not usually exceed the number of FEM nodes impacted by the collision. However, it can occur for collisions involving embedded meshes (Section 6.3.2) when the mesh has a finer resolution than the embedding FEM, or skinned meshes (Chapter 8) when the number of contacts exceeds the available DOF of the underlying master bodies.

At present there are two general ways to handle overconstrained contact:

1. Contact constraint reduction

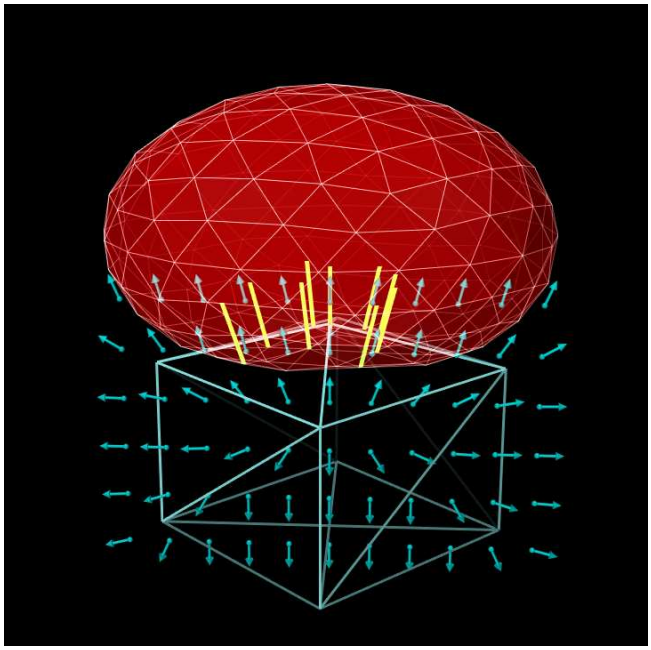


Figure 4.8: Contacts generated by a signed distance field. A deformable ellipsoid (red, top) is colliding with a solid prism (cyan, bottom). A signed distance field for the prism (the grid points and normals of which are shown partially by the dark cyan arrows) is used to locate penetrating vertices of the ellipsoid and hence generate contact constraints (yellow lines).

2. Contact regularization

Constraint reduction involves having the collision manager explicitly try to reduce the number of collision constraints to match the available DOFs, and is enabled by setting the reduceConstraints property for either the collision manager *or* the CollisionBehavior for a particular collision pair to `true`. The default value of reduceConstraints is `false`. As with all properties, it can be set interactively in the GUI, or in code using the property's accessor methods,

```
boolean getReduceContraints()
void setReduceContraints (boolean enable)
```

as illustrated by the following code fragments:

```
MechModel mech;
...

// enable constraint reduction for all collisions:
mech.getCollisionManager().setReduceContraints (true);

// enable constraint reduction for collisions between bodyA and bodyB:
CollisionBehavior behav =
   mech.setCollisionBehavior (bodyA, bodyB, true);
behav.setReduceContraints (true);
```

Contact regularization is a different technique in which contact constraints are made "soft" by adding compliance and damping terms. This means that they no longer remove DOFs from the system, and so overconstraining cannot occur, but contact penetration is no longer strictly enforced and the overall computation time can be increased.

> The regularization described here is analogous to that used for joints and connectors, discussed in Section 3.3.7.

Regularization can be enabled by setting the compliance and damping properties for either the collision manager *or* the CollisionBehavior for a particular collision pair. Compliance is the *inverse* of stiffness, and so a value of 0 (the default) implies "infinitely" stiff contact constraints. Regularization is enabled whenever the compliance is set to a value greater

than 0, with stiffness decreasing as compliance increases. While it is not required to set the damping property, it is usually desirable to set it to a value that approximates critical damping in order to prevent "bouncing" contact behavior. Compliance and damping can be set in the GUI by editing the properties of either the collision manager or a specific collision behavior, or set in code using the accessor methods

```
double getCompliance()
void setCompliance (double c)

double getDamping()
void setDamping (double d)
```

as is illustrated by the following code fragments:

```
MechModel mech;
double compliance;
double damping;

... determine compliance and damping values ...

// set damping and compliance for all collisions:
mech.getCollisionManager().setCompliance (compliance);
mech.getCollisionManager().setDamping (damping);

// enable compliance and damping between bodyA and bodyB:
CollisionBehavior behav =
   mech.setCollisionBehavior (bodyA, bodyB, true);
behav.setCompliance (compliance);
behav.setDamping (damping);
```

An important question is what values to choose for compliance and damping. At present, there is no automatic way to determine these, and so some experimental parameter tuning is often necessary. Compliance introduces an approximately linear stiffness into each contact constraint, with a stiffness constant K that results in a force acting along the constraint direction (i.e., the contact normal) with a magnitude f is given by

$$f = Kd \tag{4.5}$$

where d is the contact penetration depth. Appropriate values of K will depend on the desired maximum penetration depth and other loadings acting on the body. The mass of the collidable bodies may also be relevant if one wants to control how fast the contact stabilizes. (The mass of every CollidableBody can be determined by its `getMass()` method.) Since K acts on a per-contact basis, the resulting total force will increase with the number of contacts.

Once K is determined, the compliance value C is simply the inverse: $C = 1/K$. Furthermore, the damping D can then be estimated based on the desired damping ratio ζ, using the formula

$$D = 2\zeta\sqrt{KM} \tag{4.6}$$

where M is the combined mass of the collidable bodies (or the mass of one body if the other is non-dynamic). Typically, the desired damping will be close to critical damping, for which $\zeta = 1$.

An example that allows a user to play with contact regularization is given in Section 4.6.6.

4.6.4 Penetration tolerance and limitations

ArtiSynth's attempt to separate colliding bodies at the end of each time step may cause a jittering behavior around the colliding area, as the surface collides, separates, and re-collides. This can usually be stabilized by maintaining a certain interpenetration distance during contact. This distance is controlled by the `MechModel` property `penetrationTol`. ArtiSynth attempts to compute a suitable default value for this property, but for some applications it may be necessary to control the value explicitly using the `MechModel` methods

```
setPenetrationTol (double dist);
double getPenetrationTol();
```

Another issue is that because ArtiSynth currently uses static collision detection, it is possible for objects that are fast enough or thin enough to completely pass through each other in one simulation step. This means that for thin objects, it is important to keep the step size small enough to prevent such undetected interpenetration.

ArtiSynth also uses a "box" friction approximation [7] to compute dry friction, instead of the polyhedralized friction cones common in the multibody dynamics literature [1, 15]. This allows for a less expensive and more robust computation at the expense of some accuracy.

4.6.5 Contact rendering

As mentioned above, `CollisionBehavior` also contains properties that can enable and control the rendering of artifacts associated with the contact. These include intersection contours and contact points and normals, as described below. Colors, line widths, and other graphical details are controlled by the generic render properties (Section 4.3) of the collision manager, or by render properties which can be set individually within the behavior components.

By default, contact rendering is disabled. To enable it, one must set the collision manager to be *visible*, which can be done using a code fragment like the following:

```
RenderProps.setVisible (mechModel.getCollisionManager(), true);
```

Specific properties of CollisionBehavior that control contract rendering include:

drawIntersectionContours

Boolean value requesting drawing of the intersection contours between collidable meshes, using the edgeWidth and edgeColor render properties.

drawIntersectionPoints

Boolean value requesting drawing of the interpenetrating vertices on each collidable mesh, using the pointStyle, pointSize, pointRadius, and pointColor render properties.

drawContactNormals

Boolean value requesting drawing of the normals associated with each contact constraint, using the lineStyle, lineRadius, lineWidth, and lineColor render properties. The length of the normals is controlled by the contactNormalLen property of the collision manager, which will be set to an appropriate default value if not set explicitly.

drawContactForces

Boolean value requesting drawing of the forces associated with each contact constraint, using the lineStyle, lineRadius, edgeWidth, and edgeColor (or lineColor if edgeColor is `null`) render properties. The forces are drawn as line segments starting at each contact point and running parallel to the contact normal, with a length given by the current contact impulse value multiplied by the contactForceLenScale property of the collision manager (which has a default value of 1). The reason for using edgeWidth and edgeColor instead of lineWidth and lineColor is to allow the application to set the render properties such that both normals and forces can be visible if both are being rendered at the same time.

drawColorMap

An enum of type CollisionBehavior.ColorMapType requesting that a color map be drawn over the contact regions showing a scalar value such as penetration depth or contact force pressure. The collidable (0 or 1) onto which the color map is drawn is controlled by the colorMapCollidable property (described below). The range of values used for generating the map is controlled by the colorMapRange property of either the collision manager or the behavior, as described below. The values are mapped onto colors using the colorMap property of the collision manager.

Values of CollisionBehavior.ColorMapType include:

NONE

The color map is disabled. This is the default.

PENETRATION_DEPTH

The color map shows penetration depth of one collidable mesh with respect to the other. If one or both collidable meshes are open, then the penetration computation works more reliably if the colliderType property is set to `AJL_CONTOUR` (Section 4.5.3).

CONTACT_PRESSURE

The color map shows the contact pressures over the contact region. Contact pressures are determined by examining the forces at the contact constraints and then distributing these over the surrounding faces.

The color map itself is drawn as a patch formed from the collidable's collision mesh, using faces and vertices associated with the collision region. The vertices of the patch are set to colors corresponding to the associated value (e.g., penetration depth or pressure) at that vertex, and the surrounding faces are colored appropriately. The resolution of the color map is thus determined by the resolution of the collision mesh, and so the application must ensure this is high enough to ensure proper results. If the mesh has only a few triangles (e.g., a rectangle with two triangles per face), the color interpolation may be spread over an unreasonably large area. Examples of color map usage are given in Sections 4.6.7 and 6.12.2.

colorMapCollidable Integer value of either 0 or 1 specifying the collidable onto which color maps should be drawn when the drawColorMap property is set to a value other than `NONE`. 0 or 1 selects either the first or second collidable associated with the collision behavior.

colorMapRange

Composite property of the type ScalarRange that controls the range of values used for color map rendering. Sub properties include interval, which gives the value range itself, and updating, which specifies how this interval should be updated: `FIXED` (no updating), `AUTO_EXPAND` (interval is expanded as values increase or decrease), and `AUTO_FIT` (interval is reset to fit the values at every render step).

When the colorMapRange value for a `CollisionBehavior` is set to `null` (which is the default), the colorMapRange of the `CollisionManager` is used instead. This has the advantage of ensuring that the same color map range will be used across all collision interactions.

colorMapInterpolation

An enum of type Renderer.ColorInterpolation that explicitly specifies how colors in any rendered color map should be interpolated. `RGB` and `HSV` (the default) specify interpolation in RGB and HSV space, respectively. HSV interpolation is the default as it is generally better suited to rendering maps that are purely color-based.

Most of the above properties are also present in the `CollisionManager`, from which they are inherited by all behaviors that do not set them explicitly.

Generic render properties within the collision manager can be set in the same way as the visibility, using the `RenderProps` methods presented in Section 4.3.2:

```
Renderable cm = mechModel.getCollisionManager();
RenderProps.setEdgeWidth (cm, 2);
RenderProps.setEdgeColor (cm, Color.Red);
```

As mentioned above, generic render properties can also be set individually for specific behaviors. This can be done using code fragments like this:

```
CollisionBehavior behav = mechModel.getCollisionBehavior (bodA, bodB);
RenderProps.setLineWidth (behav, 2);
RenderProps.setLineColor (behav, Color.Blue);
```

To access these properties on a read-only basis, one can do

```
RenderProps props = mechModel.getCollisionManager().getRenderProps();

... OR ...

RenderProps props = behav.getRenderProps();
```

4.6.6 Example: Rendering normals and contours

A simple model illustrating contact rendering is defined in

```
artisynth.demos.tutorial.BallPlateCollide
```

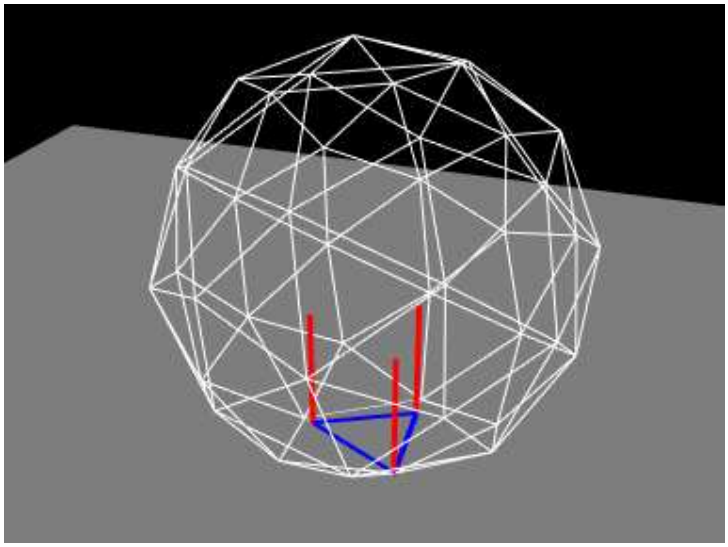


Figure 4.9: BallPlateCollide showing contact normals (red) and collision contour (blue) of the ball colliding with the plate.

This shows a ball colliding with a plate, while rendering the resulting contact normals in red and the intersection contours in blue. This demo also allows the user to experiment with contact regularization (Section 4.6.3) by setting compliance and damping properties in a control panel.

The complete source code is shown below:

```
package artisynth.demos.tutorial;

import java.awt.Color;

import artisynth.core.gui.ControlPanel;
import artisynth.core.mechmodels.CollisionManager;
import artisynth.core.mechmodels.MechModel;
import artisynth.core.mechmodels.RigidBody;
import artisynth.core.workspace.RootModel;
import maspack.matrix.RigidTransform3d;
import maspack.render.RenderProps;
import maspack.render.Renderer;

public class BallPlateCollide extends RootModel {

   public void build (String[] args) {

      // create MechModel and add to RootModel
      MechModel mech = new MechModel ("mech");
      addModel (mech);

      // create and add the ball and plate
      RigidBody ball = RigidBody.createIcosahedralSphere ("ball", 0.8, 0.1, 1);
      ball.setPose (new RigidTransform3d (0, 0, 2, 0.4, 0.1, 0.1));
      mech.addRigidBody (ball);
      RigidBody plate = RigidBody.createBox ("plate", 5, 5, 0.4, 1);
      plate.setDynamic (false);
      mech.addRigidBody (plate);

      // turn on collisions
      mech.setDefaultCollisionBehavior (true, 0.20);

      // make ball transparent so that contacts can be seen more clearly
      RenderProps.setFaceStyle (ball, Renderer.FaceStyle.NONE);
      RenderProps.setShading (ball, Renderer.Shading.NONE);
```

```
36        RenderProps.setDrawEdges (ball, true);
37        RenderProps.setEdgeColor (ball, Color.WHITE);
38
39        // enable rendering of contacts normals and contours
40        CollisionManager cm = mech.getCollisionManager();
41        RenderProps.setVisible (cm, true);
42        RenderProps.setLineWidth (cm, 3);
43        RenderProps.setLineColor (cm, Color.RED);
44        RenderProps.setEdgeWidth (cm, 3);
45        RenderProps.setEdgeColor (cm, Color.BLUE);
46        cm.setDrawContactNormals (true);
47        cm.setDrawIntersectionContours (true);
48
49        // create a control panel to allow contact regularization to be set
50        ControlPanel panel = new ControlPanel();
51        panel.addWidget (mech.getCollisionManager(), "compliance");
52        panel.addWidget (mech.getCollisionManager(), "damping");
53        addControlPanel (panel);
54     }
55  }
```

The `build()` method starts by creating and adding a `MechModel` in the usual way (lines 19-20). The ball and plate are both created as rigid bodies (lines 22-28), with the ball pose set so that its origin is above the plate at (0, 0, 2) and its orientation is perturbed so that it will not land on the plate symmetrically (line 24). Collisions between the ball and plate are enabled at line 31, with a friction coefficient of 0.2. To allow better visualization of the contacts, the ball is made transparent by disabling the drawing of faces, and instead enabling the drawing of edges in white with no shading (lines 33-37).

Rendering of contacts and normals is established by setting the render properties of the collision manager (lines 39-47). First, the collision manager is set visible (which it is not by default). Then lines (used to render the contact normals) and edges (used to render to intersection contour) are set to red and blue, each with a pixel width of 3. Drawing of the normals and contour is enabled at lines 46-47.

Lastly, for interactively controlling registration, a control panel is built to allow users to adjust the collision manager's compliance and damping properties (lines 49-53).

To run this example in ArtiSynth, select All demos > tutorial > BallPlateCollide from the Models menu. When run, the ball will collide with the plate and the contact normals and collision contours will be drawn as shown in Figure 4.9.

To enable contact regularization, set the compliance to a non-zero value. A value of 0.001 (which corresponds to a contact stiffness of 1000) causes the ball to bounce considerably when it lands. To counteract this bouncing, the damping should be set to a non-zero value. Since the ball has a mass of 0.21, formula (4.6) suggests that critical damping (for which $\zeta = 1$) can be achieved with $D \approx 30$. This does in fact stop the bouncing. Increasing the compliance to 0.01 results in the ball penetrating the plate by a noticeable amount.

4.6.7 Example: Rendering a color map

As described above, it is possible to use the drawColorMap property of the collision behavior to render a color map over the contact area showing a scalar value such as penetration depth or contact pressure. A simple example of this is defined in

```
artisynth.demos.tutorial.PenetrationRender
```

which sets drawColorMap to `PENETRATION_DEPTH` in order to display the penetration depth of one hemispherical mesh with respect to another.

> The code to render contact pressure is very similar, with drawColorMap instead set to `CONTACT_PRESSURE`. An example for finite elements models is shown in Section 6.12.2.

As mentioned above, proper results require that the collision mesh for the collidable on which the map is being drawn has a sufficiently high resolution.

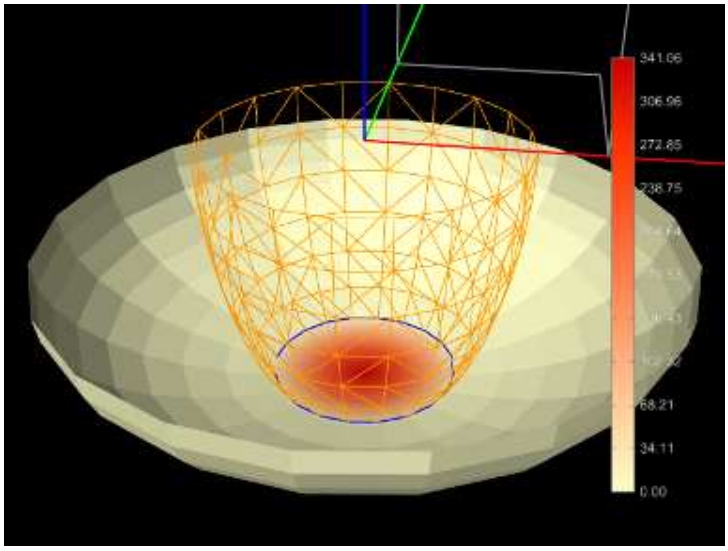


Figure 4.10: PenetrationRender showing the penetration depth of the bottom mesh with respect to the top, with red indicating greater depth. A translation dragger fixture at the top is being used to move the top mesh around, while the penetration range and associated colors are displayed on the color bar at the right.

The complete source code is shown below:

```
1  package artisynth.demos.tutorial;
2
3  import java.awt.Color;
4
5  import maspack.geometry.PolygonalMesh;
6  import maspack.geometry.MeshFactory;
7  import maspack.matrix.RigidTransform3d;
8  import maspack.render.*;
9  import maspack.render.Renderer.FaceStyle;
10 import maspack.render.Renderer.Shading;
11 import artisynth.core.mechmodels.*;
12 import artisynth.core.mechmodels.CollisionManager.ColliderType;
13 import artisynth.core.mechmodels.CollisionBehavior.ColorMapType;
14 import artisynth.core.util.ScalarRange;
15 import artisynth.core.workspace.RootModel;
16 import artisynth.core.renderables.ColorBar;
17 import maspack.render.color.JetColorMap;
18
19 public class PenetrationRender extends RootModel {
20
21    // Convenience method for creating colors from [0-255] RGB values
22    private static Color createColor (int r, int g, int b) {
23       return new Color (r/255.0f, g/255.0f, b/255.0f);
24    }
25
26    private static Color CREAM = createColor (255, 255, 200);
27    private static Color GOLD = createColor (255, 150, 0);
28
29    // Creates and returns a rigid body built from a hemispherical mesh.  The
30    // body is centered at the origin, has a radius of 'rad', and the z axis is
31    // scaled by 'zscale'.
32    RigidBody createHemiBody (
33       MechModel mech, String name, double rad, double zscale, boolean flipMesh) {
```

```
34
35       PolygonalMesh mesh = MeshFactory.createHemisphere (
36          rad, /*slices=*/20, /*levels=*/10);
37       mesh.scale (1, 1, zscale); // scale mesh in the z direction
38       if (flipMesh) {
39          // flip upside down is requested
40          mesh.transform (new RigidTransform3d (0, 0, 0, 0, 0, Math.PI));
41       }
42       RigidBody body = RigidBody.createFromMesh (
43          name, mesh, /*density=*/1000, /*scale=*/1.0);
44       mech.addRigidBody (body);
45       body.setDynamic (false);  // body is only parametrically controlled
46       return body;
47    }
48
49    // Creates and returns a ColorBar renderable object
50    public ColorBar createColorBar() {
51       ColorBar cbar = new ColorBar();
52       cbar.setName("colorBar");
53       cbar.setNumberFormat("%.2f");      // 2 decimal places
54       cbar.populateLabels(0.0, 1.0, 10); // Start with range [0,1], 10 ticks
55       cbar.setLocation(-100, 0.1, 20, 0.8);
56       cbar.setTextColor (Color.WHITE);
57       addRenderable(cbar);               // add to root model's renderables
58       return cbar;
59    }
60
61    public void build (String[] args) {
62       MechModel mech = new MechModel ("mech");
63       addModel (mech);
64
65       // create first body and set its rendering properties
66       RigidBody body0 = createHemiBody (mech, "body0", 2, -0.5, false);
67       RenderProps.setFaceStyle (body0, FaceStyle.FRONT_AND_BACK);
68       RenderProps.setFaceColor (body0, CREAM);
69
70       // create second body and set its pose and rendering properties
71       RigidBody body1 = createHemiBody (mech, "body1", 1, 2.0, true);
72       body1.setPose (new RigidTransform3d (0, 0, 0.75));
73       RenderProps.setFaceStyle (body1, FaceStyle.NONE); // set up
74       RenderProps.setShading (body1, Shading.NONE);     // wireframe
75       RenderProps.setDrawEdges (body1, true);           // rendering
76       RenderProps.setEdgeColor (body1, GOLD);
77
78       // create and set a collision behavior between body0 and body1, and make
79       // collisions INACTIVE since we only care about graphical display
80       CollisionBehavior behav = new CollisionBehavior (true, 0);
81       behav.setMethod (CollisionBehavior.Method.INACTIVE);
82       behav.setDrawColorMap (ColorMapType.PENETRATION_DEPTH);
83       behav.setColorMapCollidable (1); // show penetration of mesh 0
84       mech.setCollisionBehavior (body0, body1, behav);
85
86       CollisionManager cm = mech.getCollisionManager();
87       // works better with open meshes if AJL_CONTOUR is selected
88       cm.setColliderType (ColliderType.AJL_CONTOUR);
89       // set other rendering properities in the collision manager:
90       RenderProps.setVisible (cm, true);    // enable collision rendering
91       cm.setDrawIntersectionContours(true); // draw contours ...
92       RenderProps.setEdgeWidth (cm, 3);     // with a line width of 3
93       RenderProps.setEdgeColor (cm, Color.BLUE); // and a blue color
94       // create a custom color map for rendering the penetration depth
95       JetColorMap map = new JetColorMap();
96       map.setColorArray (
97          new Color[] {
98             CREAM,                         // no penetration
```

```
99              createColor (255, 204, 153),
100             createColor (255, 153, 102),
101             createColor (255, 102, 51),
102             createColor (255, 51, 0),
103             createColor (204, 0, 0),     // most penetration
104          });
105       cm.setColorMap (map);
106       // set color map range to "auto fit".
107       cm.setColorMapRange (new ScalarRange(ScalarRange.Updating.AUTO_FIT));
108
109       // create a separate color bar to show depth values associated with the
110       // color map
111       ColorBar cbar = createColorBar();
112       cbar.setColorMap (map);
113    }
114
115    public void prerender(RenderList list) {
116       super.prerender(list); // call the regular prerender method first
117
118       // Update the color bar labels based on the collison manager's
119       // color map range that was updated in super.prerender().
120       //
121       // Object references are obtained by name using 'findComponent'. This is
122       // more robust than using class member variables, since the latter will
123       // be lost if we save and restore this model from a file.
124       ColorBar cbar = (ColorBar)(renderables().get("colorBar"));
125       MechModel mech = (MechModel)findComponent ("models/mech");
126       RigidBody body0 = (RigidBody)mech.findComponent ("rigidBodies/body0");
127       RigidBody body1 = (RigidBody)mech.findComponent ("rigidBodies/body1");
128
129       CollisionManager cm = mech.getCollisionManager();
130       ScalarRange range = cm.getColorMapRange();
131       cbar.updateLabels(0, 1000*range.getUpperBound());
132    }
133 }
```

To improve visibility, the example uses two rigid bodies, each created from an open hemispherical mesh using the method `createHemiBody()` (lines 32-47). Because this example is strictly graphical, the bodies are set to be non-dynamic so that they can be moved around using the viewer's graphical dragger fixtures (see the section "Transformer Tools" in the ArtiSynth User Interface Guide). Rendering properties for each body are set at lines 67-68 and 73-76, with the top body being rendered as a wireframe to improve visibility.

Lines 80-84 create and set a collision behavior between the two bodies with the drawColorMap property set to `PENETRATION_DEPTH`. Because for this example we want to show only the penetration and don't want a collision response, we set the collision method to be Method.INACTIVE. At line 88, the collider type used by the collision manager is set to ColliderType.AJL_CONTOUR, since this provides more reliable penetration calculations for open meshes. Other rendering properties are set for the collision manager at lines 90-107, including a custom color map that varies between `CREAM` (the color of the mesh) for no penetration and dark red for maximum penetration. The updating of the color map range in the collision manager is set to `AUTO_FIT` so that it will be recomputed at every time step. (Since the collision manager's color map range is set to "auto fit" by default, this is shown for illustrative purposes only. It is also possible to override the collision manager's color map range by setting the colorMapRange property in specific collision behaviors.)

At line 111, a color bar is created and added to the scene, using the method `createColorBar()` (lines 50-59), to explicitly show the depth that corresponds to the different colors. The color bar is given the same color map that is used to render the depth. Since the depth range is updated every time step, it is also necessary to update the corresponding labels in the color bar. This is done by overriding the root model's `prerender()` method (lines 115-132), where we obtain the color map range for the collision manager and use it to update the color bar labels. (Note that `super.prerender(list)` is called *first*, since the color map ranges are updated there.) References to the color bar, `MechModel`, and bodies are obtained using the CompositeComponent methods get() and findComponent(). This is more robust that storing these references in member variables, since the latter would be lost if the model is saved to and reloaded from a file.

To run this example in ArtiSynth, select All demos > tutorial > PenetrationRender from the Models menu. When run, the meshes will collide and render the penetration depth of the bottom mesh, as shown in Figure 4.10.

When defining the color map for rendering (lines 99-108 in the example), it is recommended that the color corresponding to zero be set to the face color of the collidable mesh. This will allow the color map to blend properly to the regular mesh color.

4.7 Distance Grids and Components

Distance grids, implemented by the class DistanceGrid, are currently used in ArtiSynth for both collision handling (Section 4.6.1) and spring and muscle wrapping around general surfaces (Section 7.3). A distance grid is a regular three dimensional grid that is used to interpolate a scalar distance field and its associated normals. For collision handling and wrapping purposes, this distance function is assumed to be signed and is used to estimate the penetration depth and associated normal direction of a point within some 3D object.

A distance grid consists of $n_x \times n_y \times n_z$ evenly spaced vertices partitioning a volume into $r_x \times r_y \times r_z$ cuboid cells, with

$$r_x = n_x - 1, \quad r_y = n_y - 1, \quad r_z = n_z - 1.$$

The grid has overall widths w_x, w_y, w_z along each axis, so that the widths of each cell are w_x/r_x, w_y/r_y, w_z/r_z.

Scalar distances and normal vectors are stored at each vertex, and interpolated at points within each cell using trilinear interpolation. Distance values (although not normals) can also be interpolated *quadratically* over a composite *quadratic* cell composed of $2 \times 2 \times 2$ regular cells. This provides a smoother result than trilinear interpolation and is currently used for muscle wrapping. To ensure that all points within the grid can be assigned a unique quadratic cell, the grid resolution is restricted so that r_x, r_y, r_z are always even. Distance grids are typically generated automatically from mesh data. More details on the actual functionality of distance grids is contained in the DistanceGrid API documentation.

When used within ArtiSynth, distance grids are generally contained within an encapsulating DistanceGridComp component, which maintains a mesh (or list of meshes) used to generate the grid, and exports properties for controlling its resolution, mesh fit, and visualization. For any object that implements CollidableBody (which includes RigidBody), its distance grid component can be obtained using the method

```
DistanceGridComp getDistanceGridComp();
```

A distance grid maintains its own local coordinate system, which is related to world coordinates by a *local-to-world* transform that can be queried and set via the component methods setLocalToWorld() and getLocalToWorld(). For grids associated with a rigid body, the local coordinate system is synchronized with the rigid body's coordinate system (which is queried and set via setPose() and getPose()). The grid's axes are nominally aligned with the local coordinate system, although they can be subject to an additional orientation offset (e.g., see the description of the property fitWithOBB below).

By default, a `DistanceGridComp` generates its grid automatically from its mesh data, within an axis-aligned bounding volume with the resolutions r_x, r_y, r_z chosen automatically. However, in some cases it may be necessary to control the resolution explicitly. `DistanceGridComp` exports the following properties to control both the grid's resolution and how it is fit around the mesh data:

resolution A vector of 3 integers that specifies the resolutions r_x, r_y, r_z. If any value in this vector is set ≤ 0, then all values are set to zero and the maxResolution property is used to determine the grid divisions instead.

maxResolution Sets the default maximum cell resolution that should be used when constructing the grid. This is the number of cells that should be used along the *longest* bounding volume axis, with the cell counts along the other axes adjusted to maintain a uniform cell size. If all three values of resolution are > 0, those will be used to specify the cell resolutions instead.

fitWithOBB If `true`, offsets the orientation of the grid's x, y, and z axes (with respect to local coordinates) to align with an oriented bounding box fit to the mesh. Otherwise, the grid axes are aligned with those of the local coordinate frame.

marginFraction Specifies the fractional amount by which the mesh should be extended with respect to a bounding box fit around the mesh(es). The default value is `0.1`.

As with all properties, these can be set either interactively (either using a custom control panel or by selecting the component and then choosing Edit properties ... from the right-click context menu), or through their set/get accessor methods. For example, resolution and maxResolution can be set and queried via the methods:

```
void setResolution (Vector3i res)
Vector3i getResolution()

void setMaxResolution (int max)
int getMaxResolution()
```

When used for either collisions or wrapping, the distance values interpolated by a distance grid are used to determine whether a point is inside or outside some 3D object, with the inside/outside boundary being the isosurface surface corresponding to a distance value of 0. This isosurface surface (which differs depending on whether trilinear or quadratic distance interpolation is used) therefore represents the effective collision boundary, and it may be somewhat different from the mesh surface used to generate it. It may be smoother, or may have discretization artifacts, depending on both the smoothness and complexity of the mesh and the grid's resolution (Figure 4.11). It is therefore important to be able to visualize both the trilinear and quadratic isosurfaces. `DistanceGridComp` provides a number of properties to control this along with other aspects of grid visualization:

renderProps Render properties that control the colors, styles, and sizes (or widths) used to render faces, lines and points (Section 4.3). Point, line and edge properties are used for rendering grid vertices, normals, and cell edges, while face properties are used for rendering the isosurface. One can select which components are visible by zeroing appropriate render properties: zeroing `pointRadius` (or `pointSize`, if the `pointStyle` is `POINT`) disables drawing of the vertices; zeroing `lineRadius` (or `lineWidth`, if the `lineStyle` is `LINE`) disables drawing of the normals; and zeroing `edgeWidth` disables drawing of the cell edges. For an illustration, see Figure 4.12.

renderGrid If set to `true`, causes the grid's vertices, normals and cell edges to be rendered, using the render properties as described above.

renderRanges Can be used to restrict which part of the distance grid is drawn. The value is a string which specifies the vertex ranges along the x, y, and z axes. In general, the grid will have $n_x \times n_y \times n_z$ vertices along the x, y, and z axes, where n_x, n_y, and n_z are each one greater than the cell resolutions r_x, r_y, and r_z. The range string should contain three range specifications, one for each axis, where each specification is either `*` (all vertices), `n:m` (vertices in the index range `n` to `m`, inclusive), or `n` (vertices only at index `n`). A range specification of `"* * *"` (or `"*"`) means draw all vertices, which is the default behavior. Other examples include:

```
"* 7 *"      all vertices along x and z, and those at index 7 along y
"0 2 3"      a single vertex at indices (0, 2, 3)
"0:3 4:5 *"  all vertices between 0 and 3 along x, and 4 and 5 along y
```

For an illustration, see Figure 4.12.

renderSurface If set to `true`, renders the grid's isosurface, as determined by the surfaceType property. See Figure 4.11, right.

surfaceType Controls the interpolation used to form the isosurface rendered in response to the renderSurface property. `QUADRATIC` (the default) specifies a quadratic isosurface, while `TRILINEAR` specifies a trilinear isosurface.

surfaceDistance Controls the level set value used to determine the isosurface. To render the isosurface used for collision handling or muscle wrapping, this value should be 0.

When visualizing the isosurface for a distance grid, it is generally convenient to also turn *off* visualization for the meshes used to generate the grid. For `RigidBody` objects, this can be accomplished easily using the convenience property gridSurfaceRendering. If set `true`, it will cause the isosurface to be rendered *instead* of its mesh components. The isosurface type will be that indicated by the grid component's surfaceType property, and the rendering will occur independently of the visibility settings for the meshes or the grid component.

4.8 Transforming geometry

Certain ArtiSynth components, including `MechModel`, implement the interface TransformableGeometry, which allows the geometric transformation of the component's attributes (such as meshes, points, frame locations, etc.), along with its descendant components. The interface provides the method

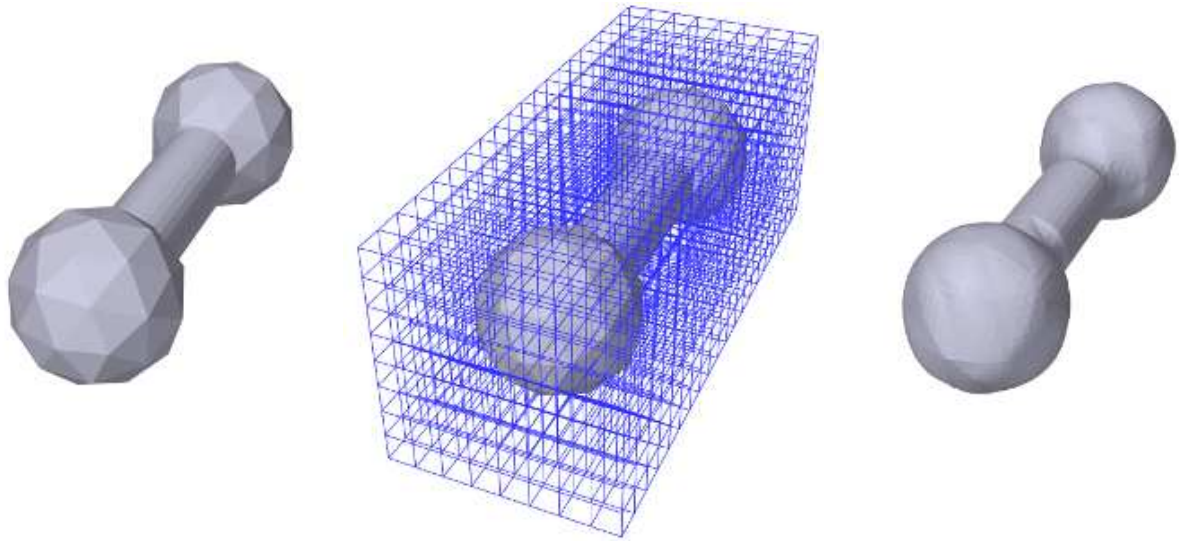


Figure 4.11: A mesh used to generate a distance grid, (left), along with a visualization of the grid itself (middle) and the corresponding quadratic isosurface (right). Notice how in this case the quadratic isosurface is smoother than the coarser features of the original mesh.

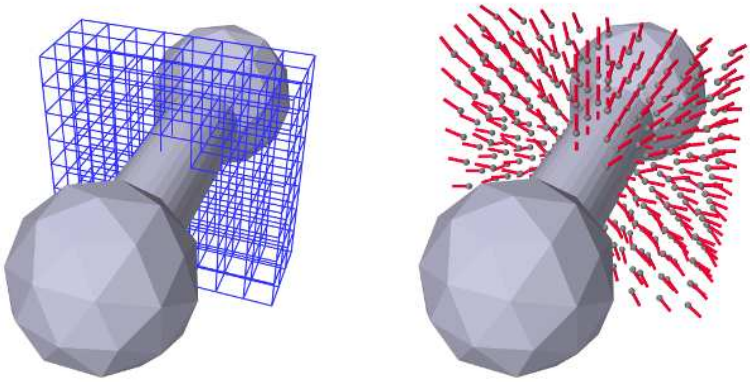


Figure 4.12: Rendering a subportion of a distance grid restricted along the x axis by setting renderRanges to `"9:12 * *"`, with render properties set to show the grid cells (left), and the vertices and normals (right).

```
public void transformGeometry (AffineTransform3dBase X);
```

where `X` is an AffineTransform3dBase that may be either a RigidTransform3d or a more general AffineTransform3d (Section 2.2).

transformGeometry(X) can be used to translate, rotate, shear or scale components. It can be applied to an entire model or individual components. Unlike scaleDistance(), it actually changes the physical geometry and so may change the simulation behavior. For example, applying `transformGeometry()` to a RigidBody will cause the shape of its mesh to change, which will change its mass if its inertiaMethod property is set to `DENSITY`. Figure 4.13 shows a simplified illustration of both rigid and affine transformations being applied to a model.

The example below shows how to apply a transformation to a model in code. In it, a `MechModel` is first scaled by the factors 1.5, 2, and 3 along the x, y, and z axes, and then flipped upside down using a `RigidTransform3d` that rotates it by 180 degrees about the x axis:

```
MechModel mech;

... build mech model ...
```

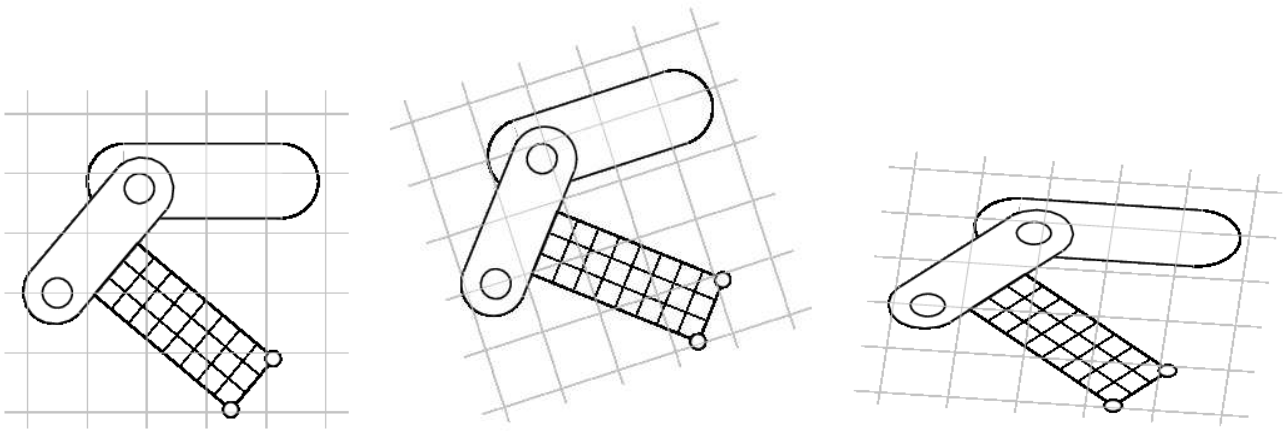


Figure 4.13: Simple illustration of a model (left) undergoing a rigid transformation (middle) and an affine transformation (right).

```
AffineTransform3d X = new AffineTransform3d();
X.applyScaling (1.5, 2, 3);
mech.transformGeometry (X);

RigidTransform3d T =
   new RigidTransform3d (/*x,y,z=*/0, 0, 0, /*r,p,y=*/0, 0, Math.PI);
mech.transformGeometry (T);
```

4.8.1 Nonlinear transformations

The TransformableGeometry interface also supports general, nonlinear geometric transforms. This can be done using a GeometryTransformer, which is an abstract class for performing general transformations. To apply such a transformation to a component, one can create and initialize an appropriate subclass of `GeometryTransformer` to perform the desired transformation, and then apply it using the static `transform` method of the utility class TransformGeometryContext:

```
ModelComponent comp;     // component to be transformed
GeometryTransformer gtr; // transformer to do the transforming

... instantiate and initialize the transformer ...

TransformGeometryContext.transform (comp, gtr, /*flags=*/0);
```

At present, the following subclasses of `GeometryTransformer` are available:

RigidTransformer

Implements rigid 3D transformations.

AffineTransformer

Implements affine 3D transformations.

FemGeometryTransformer

Implements a general transformation, using the deformation field induced by a finite element model.

TransformGeometryContext also supplies the following convenience methods to apply transformations to components or collections of components:

```
void transform (Iterable<TransformableGeometry>, GeometryTransformer, int);
void transform (TransformableGeometry[], GeometryTransformer, int);
```

```
void transform (TransformableGeometry, AffineTransform3dBase, int);
void transform (Iterable<TransformableGeometry>, AffineTransform3dBase, int);
void transform (TransformableGeometry[], AffineTransform3dBase, int);
```

The last three of these methods create an instance of either RigidTransformer or AffineTransformer for the supplied AffineTransform3dBase. In fact, most `TransformableGeometry` components implement their transformGeometry(X) method as follows:

```
public void transformGeometry (AffineTransform3dBase X) {
   TransformGeometryContext.transform (this, X, 0);
}
```

The FemGeometryTransformer class is derived from the class DeformationTransformer, which uses the single method getDeformation() to obtain deformation field information at a specified reference position:

```
void getDeformation (Vector3d p, Matrix3d F, Vector3d r)
```

If the deformation field is described by $\mathbf{x}' = f(\mathbf{x})$, then for a given reference position $\mathbf{r}$ (in undeformed coordinates), this method should return the deformed position $\mathbf{p} = f(\mathbf{r})$ and the deformation gradient

$$\mathbf{F} \equiv \frac{\partial f}{\partial \mathbf{x}} \tag{4.7}$$

evaluated at $\mathbf{r}$.

FemGeometryTransformer obtains $f(\mathbf{x})$ and $\mathbf{F}$ from a FemModel3d (see Section 6) whose elemental rest positions enclose the components to be transformed, using the fact that a finite element model creates an implied piecewise-smooth deformation field as it deviates from its rest position. For each reference point $\mathbf{r}$ needed by the transformation process, `FemGeometryTransformer` finds the FEM element whose rest volume encloses $\mathbf{r}$, and then uses the corresponding shape function coordinates to compute $f(\mathbf{x})$ and $\mathbf{F}$ from the element's deformation. If the FEM model does *not* enclose $\mathbf{r}$, the nearest element is used to determine the shape function coordinates (however, this calculation becomes less accurate and meaningful the farther $\mathbf{r}$ is from the FEM model). Transformations based on FEM models are further illustrated in Section 4.8.2, and by Figure 4.15. Full details on ArtiSynth finite element models are given in Section 6.

Besides FEM models, there are numerous other ways to create deformation fields, such as radial basis functions, thin plate splines, etc. Some of these may be more appropriate for a particular application and can provide deformations that are globally smooth (as opposed to piecewise smooth). It should be relatively easy for an application to create its own subclass of `DeformationTransformer` to implement the deformation of choice by overriding the single getDeformation() method.

4.8.2 Example: the FemModelDeformer class

An FEM-based geometric transformation of a `MechModel` is facilitated by the class FemModelDeformer, which one can add to an existing `RootModel` to transform the geometry of a `MechModel` already located within that `RootModel`. `FemModelDeformer` subclasses FemModel3d to include a FemGeometryTransformer, and provides some utility methods to support the transformation process.

A `FemModelDeformer` can be added to a `RootModel` by adding the following code fragment to the end of the `build()` method:

```
public void build (String[] args) {

   ... build the model ...

   FemModelDeformer deformer =
      new FemModelDeformer ("deformer", this, /*maxn=*/10);
   addModel (deformer);
   // add a control panel (this is optional)
   addControlPanel (deformer.createControlPanel());
}
```

When the deformer is created, its constructor searches the specified `RootModel` to locate the first top-level `MechModel`. It then creates a hexahedral FEM grid around this model, with `maxn` specifying the number of cells along the maximum

dimension. Material and mass properties of the model are computed automatically from the underlying `MechModel` dimensions (but can be altered if necessary after construction). When added to the `RootModel`, the deformer becomes another top-level model that can be deformed independently of the `MechModel` to create the required deformation field, as described below. It also supplies application-defined menu items that appear under the Application menu in the ArtiSynth menu bar (see Section 5.5). The deformer's `createControlPanel()` can also be used to create a `ControlPanel` (Section 5.1) that controls the visibility of the FEM model and the dynamic behavior of both it and the `MechModel`.

An example is defined in

```
artisynth.demos.tutorial.DeformedJointedCollide
```

where the `JointedCollide` example of Section 4.5.2 is extended to include a `FemModelDeformer` using the code described above.

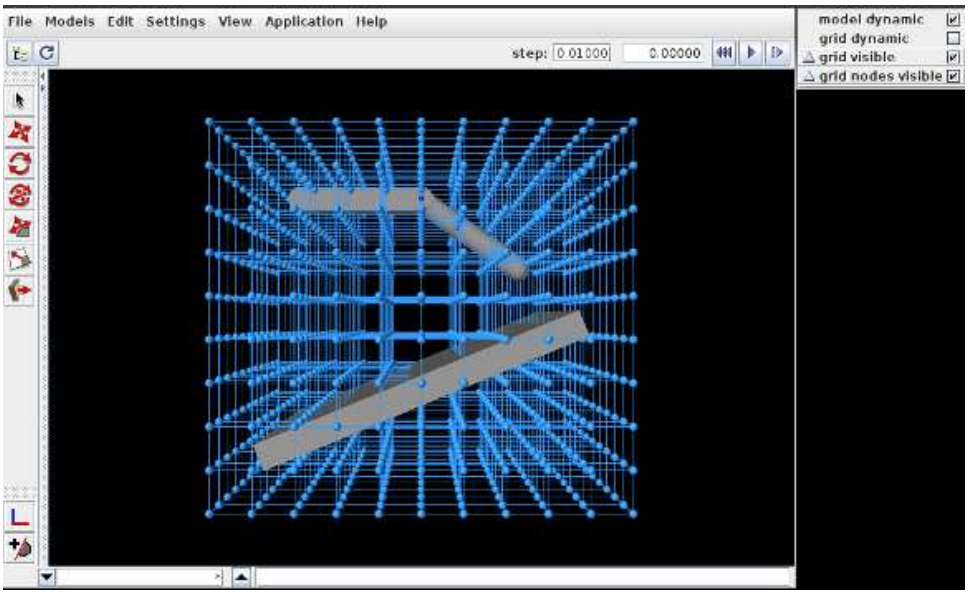


Figure 4.14: The `DeformedJointedCollide` example initially loaded into ArtiSynth.

To load this example in ArtiSynth, select All demos > tutorial > DeformedJointedCollide from the Models menu. The model should load and initially appear as in Figure 4.14, where the control panel appears on the right.

The underlying `MechModel` (or "model") can now be transformed by first deforming the FEM model (or "grid") and then using the resulting deformation field to effect the transformation:

1. Make the model non-dynamic and the grid dynamic by unchecking model dynamic and checking grid dynamic in the control panel. This means that when simulation is run, the model will be inert while the grid will respond physically.

2. Deform the grid using simulation. One easy way to do this is to fix certain nodes, generally on or near the grid boundary, and then move some of these using the translation or transrotator tool while simulation is running. To fix a set of nodes, select them in the viewer, choose Edit properties ... from the right-click context menu, and then uncheck their dynamic property. To easily select a large number of nodes without also selecting model components or grid elements, one can specify `FemNode` in the selection filter widget. (See the sections "Transformer Tools" and "Selection filtering" in the ArtiSynth User Interface Guide.)

3. After the grid has been deformed, choose deform from the Application menu in the ArtiSynth toolbar to transform the model. Afterwards, the transformation can be undone by choosing undo, and the grid can be reset by choosing reset grid.

4. To run the deformed model after the transformation, it should again be made dynamic by checking model dynamic in the control panel. The itself grid can be made non-dynamic, and it and/or its nodes can be made invisible by unchecking grid visible and/or grid nodes visible in the control panel.

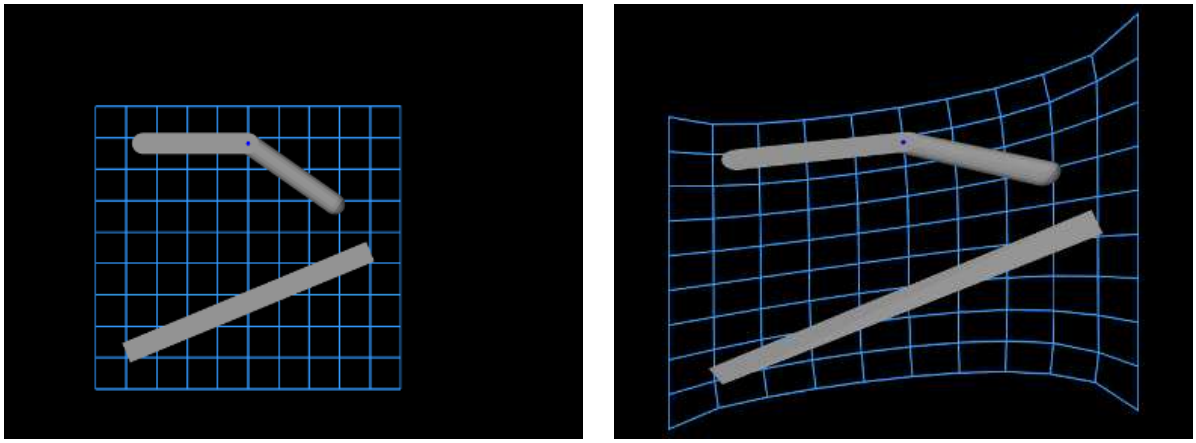


Figure 4.15: Deformation achieved in `DeformedJointedCollide`, showing both the model and grid (using an orthographic view) before and after the deformation.

The result of a possible deformation is shown in Figure 4.15.

> Note: `FemModelDeformer` is not intended to provide a general purpose solution to nonlinear geometric transformations. Rather, it is mainly intended to illustrate the capabilities of GeometryTransformer and the TransformableGeometry interface.

4.8.3 Implementation and behavior

As indicated above, the management of transforming the geometry for one or more components is handled by the TransformGeometryContext class. The transform operations themselves are carried out by this class's apply() method, which (a) assembles all the components that need to be transformed, (b) performs the actual transform operations, (c) invokes any required updating actions on other components, and finally (d) notifies parent components of the change using a GeometryChangeEvent.

To support this, ArtiSynth components which implement TransformableGeometry must also supply the methods

```
public void addTransformableDependencies (
   TransformGeometryContext context, int flags);

public void transformGeometry (
   GeometryTransformer gtr, TransformGeometryContext context, int flags);
```

The first method, addTransformableDependencies(context,flags), is called in step (a) to add to the context any additional components which should be transformed along with this component. This includes any descendants which should be transformed, since the transformation of these should not generally be done within `transformGeometry(gtr,context,flags)`.

The second method, transformGeometry(gtr,context,flags), is called in step (b) to perform the actual transformation on this component. It should use the supplied geometry transformer `gtr` to transform its attributes, as well as `context` to query what other components are also being transformed and to request any needed updating actions to be called in step (c). The `flags` argument specifies conditions associated with the transformation, which at the moment may currently include:

TG_SIMULATING

The system is currently simulating, and therefore it may not be desirable to transform all attributes;

TG_ARTICULATED

Rigid body articulation constraints should be enforced as the transform proceeds.

Full details for all this are given in the documentation for TransformGeometryContext.

The transforming behavior of a component is up to its implementing method, but the following rules are generally observed:

1. Transformable descendants are also transformed, by using `addTransformableDependencies()` to add them to the context as described above;
2. When the nodes of an FEM model (Section 6) are transformed, the rest positions are also transformed if the system is not simulating (i.e., if the `TG_SIMULATING` flag is not set). This also causes the mass of the adjacent nodes to be recomputed from the densities of the adjacent elements;
3. When dynamic components are transformed, any attachments and constraints associated with them are updated appropriately, but only if the system is not simulating. Non-transforming dynamic components that are attached to transforming components as slaves are generally updated so as to follow the transforming components to which they are attached.

4.8.4 Use in model registration

Transforming model geometry can obviously be used as part of the process of creating subject-specific biomechanical and anatomical models. However, registration will generally require more that geometric transformation, since other properties, such as material stiffnesses, densities, and maximum forces will generally need to be adjusted as well. As a specific example, when applying a geometric transform to a model containing `AxialSprings`, the `restLength` properties of the springs will be unchanged, whereas the initial lengths may be, resulting in a different applied forces and physical behavior.

4.9 General component arrangements

As discussed in Section 1.1.5 and elsewhere, a `MechModel` provides a number of predefined child components for storing particles, rigid bodies, springs, constraints, and other components. However, applications are not required to store their components in these containers, and may instead create any sort of component arrangement desired.

For example, suppose that one wishes to create a biomechanical model of both the right and left human arms, consisting of bones, point-to-point muscles, and joints. The standard containers supplied by MechModel would require that all the components be placed within the following containers:

```
rigidBodies          // all bones
axialSprings         // all point-to-point muscles
connectors           // all joints
```

Instead of this, one may wish to set up a more appropriate component hierarchy, such as

```
leftArm              // left-arm components
   bones             //   left bones
   muscles           //   left muscles
   joints            //   left joints
rightArm             // right-arm components
   bones             //   right bones
   muscles           //   right muscles
   joints            //   right joints
```

To do this, the application `build()` method can create the necessary hierarchy and then populate it with whatever components are desired. Before simulation begins (or whenever the model structure is changed), the `MechModel` will recursively traverse the component hierarchy and update whatever internal structures are needed to run the simulation.

4.9.1 Container components

The generic class ComponentList can be used as a container for model components of a specific type. It can be created using a declaration of the form

```
ComponentList<Particle> list = new ComponentList<Type> (Type.class, name);
```

where `Type` is the class type of the components and `name` is the name for the container. Once the container is created, it should be added to the `MechModel` (or another internal container) and populated with child components of the specified type. For example,

```
MechModel mech;
...
ComponentList<Particle> parts =
   new ComponentList<Particle> (Particle.class, "parts");
ComponentList<Frame> frames =
   new ComponentList<Frame> (Frame.class, "frames");

// add containers to the mech model
mech.add (parts);
mech.add (frames);
```

creates two containers named `"parts"` and `"frames"` for storing components of type `Particle` and `Frame`, respectively, and adds them to a `MechModel` referenced by `mech`.

In addition to `ComponentList`, applications may use several "specialty" container types which are subclasses of `ComponentList`:

RenderableComponentList

A subclass of `ComponentList`, that has its *own* set of render properties which can be inherited by its children. This can be useful for compartmentalizing render behavior. Note that it is *not* necessary to store renderable components in a `RenderableComponentList`; components stored in a `ComponentList` will be rendered too.

PointList

A `RenderableComponentList` that is optimized for rendering points, and also contains its own `pointDamping` property that can be inherited by its children.

PointSpringList

A `RenderableComponentList` designed for storing point-based springs. It contains a `material` property that specifies a default axial material that can be used by its children.

AxialSpringList

A `PointSpringList` that is optimized for rendering two-point axial springs.

If necessary, it is relatively easy to define one's own customized list by subclassing one of the other list types. One of the main reasons for doing so, as suggested above, is to supply default properties to be inherited by the list's descendants.

A component list which declares `ModelComponent` as its type can be used to store `any` type of component, including other component lists. This allows the creation of arbitrary component hierarchies. Generally either `ComponentList<ModelComponent>` or `RenderableComponentList<ModelComponent>` are best suited to implement hierarchical groupings.

4.9.2 Example: a net formed from balls and springs

A simple example showing an arrangement of balls and springs formed into a net is defined in

`artisynth.demos.tutorial.NetDemo`

The `build()` method and some of the supporting definitions for this example are shown below.

```
1    protected double stiffness = 1000.0;   // spring stiffness
2    protected double damping = 10.0;       // spring damping
3    protected double maxForce = 5000.0;    // max force with excitation = 1
4    protected double mass = 1.0;           // mass of each ball
5    protected double widthx = 20.0;        // width of the net along x
```

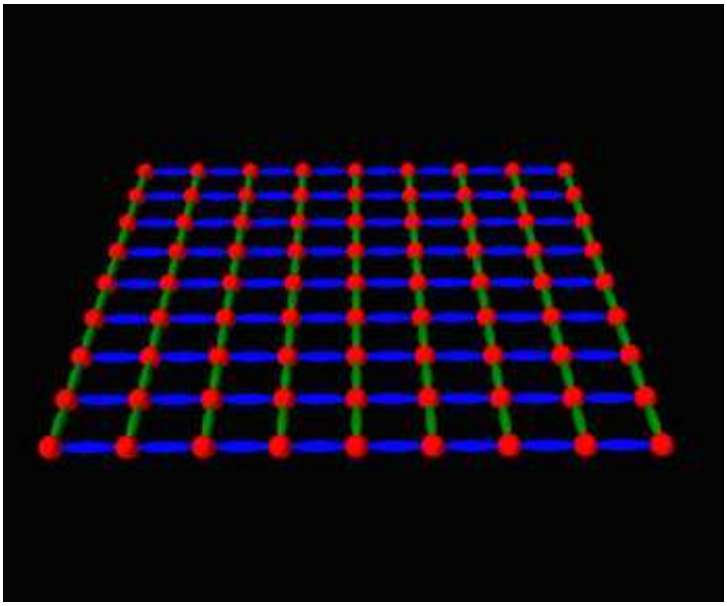


Figure 4.16: NetDemo model loaded into ArtiSynth.

```
   protected double widthy = 20.0;         // width of the net along y
   protected int numx = 8;                 // num balls along x
   protected int numy = 8;                 // num balls along y

   // custom component containers
   protected MechModel mech;
   protected PointList<Particle> balls;
   protected ComponentList<ModelComponent> springs;
   protected RenderableComponentList<AxialSpring> greenSprings;
   protected RenderableComponentList<AxialSpring> blueSprings;

   private AxialSpring createSpring (
      PointList<Particle> parts, int idx0, int idx1) {
      // create a "muscle" spring connecting particles indexed by 'idx0' and
      // 'idx1' in the list 'parts'
      Muscle spr = new Muscle (parts.get(idx0), parts.get(idx1));
      spr.setMaterial (new SimpleAxialMuscle (stiffness, damping, maxForce));
      return spr;
   }

   public void build (String[] args) {

      // create MechModel and add to RootModel
      mech = new MechModel ("mech");
      mech.setGravity (0, 0, -980.0);
      mech.setPointDamping (1.0);
      addModel (mech);

      int nump = (numx+1)*(numy+1); // nump = total number of balls

      // create custom containers:
      balls = new PointList<Particle> (Particle.class, "balls");
      springs = new ComponentList<ModelComponent>(ModelComponent.class,"springs");
      greenSprings = new RenderableComponentList<AxialSpring> (
         AxialSpring.class, "greenSprings");
      blueSprings = new RenderableComponentList<AxialSpring> (
         AxialSpring.class, "blueSprings");

      // create balls in a grid pattern and add to the list 'balls'
```

```
45        for (int i=0; i<=numx; i++) {
46           for (int j=0; j<=numy; j++) {
47              double x = widthx*(-0.5+i/(double)numx);
48              double y = widthy*(-0.5+j/(double)numy);
49              Particle p = new Particle (mass, x, y, /*z=*/0);
50              balls.add (p);
51              // fix balls along the edges parallel to y
52              if (i == 0 || i == numx) {
53                 p.setDynamic (false);
54              }
55           }
56        }
57
58        // connect balls by green springs parallel to y
59        for (int i=0; i<=numx; i++) {
60           for (int j=0; j<numy; j++) {
61              greenSprings.add (
62                 createSpring (balls, i*(numy+1)+j, i*(numy+1)+j+1));
63           }
64        }
65        // connect balls by blue springs parallel to x
66        for (int j=0; j<=numy; j++) {
67           for (int i=0; i<numx; i++) {
68              blueSprings.add (
69                 createSpring (balls, i*(numy+1)+j, (i+1)*(numy+1)+j));
70           }
71        }
72
73        // add containers to the mechModel
74        springs.add (greenSprings);
75        springs.add (blueSprings);
76        mech.add (balls);
77        mech.add (springs);
78
79        // set render properties for the components
80        RenderProps.setLineColor (greenSprings, new Color(0f, 0.5f, 0f));
81        RenderProps.setLineColor (blueSprings, Color.BLUE);
82        RenderProps.setSphericalPoints (mech, widthx/50.0, Color.RED);
83        RenderProps.setCylindricalLines (mech, widthx/100.0, Color.BLUE);
84     }
```

The `build()` method begins by creating a `MechModel` in the usual way (lines 29-30). It then creates a net composed of a set of balls arranged as a uniform grid in the x-y plane, connected by a set of green colored springs running parallel to the y axis and a set of blue colored springs running parallel to the x axis. These are arranged into a component hierarchy of the form

```
balls
springs
   greenSprings
   blueSprings
```

using containers created at lines 37-42. The balls are then created and added to `balls` (lines 45-56), the springs are created and added to `greenSprings` and `blueSprings` (lines 59-71), and the containers are added to the `MechModel` at lines 74-77. The balls along the edges parallel to the y axis are fixed. Render properties are set at lines 80-83, with the colors for `greenSprings` and `blueSprings` being explicitly set to dark green and blue.

> `MechModel`, along with other classes derived from `ModelBase`, enforces *reference containment*. That means that all components referenced by components within a `MechModel` must themselves be contained within the `MechModel`. This condition is checked whenever a component is added directly to a `MechModel` or one of its ancestors. This means that the components must be added to the `MechModel` in an order that ensures any referenced components are already present. For example, in the `NetDemo` example above, adding the particle list *after* the spring list would generate an error.

To run this example in ArtiSynth, select All demos > tutorial > NetDemo from the Models menu. The model should load and initially appear as in Figure 4.16. Running the model will cause the net to fall and sway under gravity. When the ArtiSynth navigation panel is opened and expanded, the component hierarchy will appear as in Figure 4.17. While the standard `MechModel` containers are still present, they are not displayed by default because they are empty.

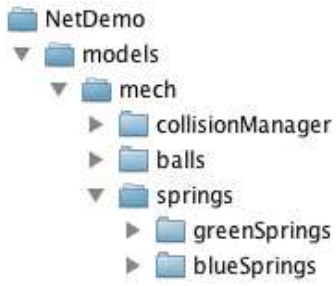


Figure 4.17: NetDemo components displayed in the ArtiSynth navigation panel.

4.9.3 Adding containers to other models

In addition to `MechModel`, application-defined containers can be added to any model that inherits from ModelBase. This includes `RootModel` and `FemModel`. However, at the present time, components added to such containers won't do anything, other than be rendered in the viewer if they are Renderable.

4.10 Custom Joints

If desired, it is also possible for applications to create their own custom joints. This involves creating two custom classes: a *coupling* class that does the constraint computations, and a *joint* class that wraps around it and allows it to connect connectable bodies. Details on how to create these classes are given in Sections 4.10.3 and 4.10.4, after some explanation of the constraint mechanism that underlies joint operation.

This section assumes that the reader is highly familiar with spatial kinematics and dynamics.

4.10.1 Joint constraints

To create a custom joint, it is necessary to understand how joints are implemented. The basic function of a joint is to constraint the set of poses allowed by the joint transform $\mathbf{T}_{CD}$ that relates frame C to D. To do this, the joint imposes restrictions on the six-dimensional spatial velocity $\hat{\mathbf{v}}_{CD}$ that describes how frame C is moving with respect to D. This restriction is done using a set of bilateral constraints $\mathbf{G}_k$ and (in some cases) unilateral constraints $\mathbf{N}_l$, each of which is a 1×6 matrix that acts to restrict a single degree of freedom in $\hat{\mathbf{v}}_{CD}$ (see Section A.5 for a review of spatial velocities and forces). Bilateral constraints take the form of an equality,

$$\mathbf{G}_k \hat{\mathbf{v}}_{CD} = 0, \tag{4.8}$$

while unilateral constraints take the form of an inequality:

$$\mathbf{N}_l \hat{\mathbf{v}}_{CD} \geq 0. \tag{4.9}$$

These constraints are defined with respect to frame C, and their total number equals the number of DOFs that the joint *removes*. A joint's main computational task is to specify these constraints. ArtiSynth then uses its own knowledge of how frames C and D are connected to bodies A and B (or ground, if there is no body B) to map the individual $\mathbf{G}_k$ and $\mathbf{N}_l$ onto the joint's full bilateral and unilateral constraint matrices $\mathbf{G}_\mathrm{J}$ and $\mathbf{N}_\mathrm{J}$ (see (3.9) and (3.10)) that restrict the body velocities.

As a simple example, consider a cylindrical joint, in which C is free to rotate about and translate along the z axis of D but other motions are restricted. Letting $\mathbf{v}$ and $\boldsymbol{\omega}$ denote the translational and rotational components of $\hat{\mathbf{v}}_{CD}$, such that

$$\hat{\mathbf{v}}_{CD} = \begin{pmatrix} \mathbf{v} \\ \boldsymbol{\omega} \end{pmatrix},$$

we see that the constraints must enforce

$$v_x = v_y = \omega_x = \omega_y = 0. \tag{4.10}$$

This can be accomplished using four constraints defined as follows:

$$\begin{aligned}\mathbf{G}_0 &= (1,0,0,0,0,0)\\ \mathbf{G}_1 &= (0,1,0,0,0,0)\\ \mathbf{G}_2 &= (0,0,0,1,0,0)\\ \mathbf{G}_3 &= (0,0,0,0,1,0).\end{aligned}$$

Constraining velocities is a necessary but insufficient condition for constraint enforcement. Because of numerical errors, as well as the fact that constraints are often nonlinear, the joint transform $\mathbf{T}_{CD}$ will tend to drift away from the joint restrictions as the simulation proceeds, leading to the error $\mathbf{T}_{err}$ described at the end of Section 3.3.1. These errors are corrected during a *position correction* at the end of every simulation time step: the joint first *projects* $\mathbf{T}_{CD}$ onto the nearest valid constraint surface to form $\mathbf{T}_{GD}$, and $\mathbf{T}_{err}$ is then computed from

$$\mathbf{T}_{err} = \mathbf{T}_{CG} = \mathbf{T}_{GD}^{-1}\mathbf{T}_{CD}. \tag{4.11}$$

Because $\mathbf{T}_{err}$ is (usually) small, we can approximate it as a twist $\hat{\boldsymbol{\delta}}_{err}$ representing a small displacement from frame G (which lies on the constraint surface) to frame C. During the position correction, ArtiSynth adjusts the pose of C relative to D in order to try and bring $\hat{\boldsymbol{\delta}}_{err}$ to zero. To do this, it uses an estimate of the *distance* d_k along each constraint to the constraint surface, which it computes from the dot product of $\mathbf{G}_k$ and $\hat{\boldsymbol{\delta}}_{err}$:

$$d_k = \mathbf{G}_k \, \hat{\boldsymbol{\delta}}_{err}. \tag{4.12}$$

ArtiSynth assembles these distances into a composite distance vector $\mathbf{d}_g$ for *all* bilateral constraints, and then uses the system solver to find a displacement $\delta\mathbf{q}$ of the system coordinates that satisfies

$$\mathbf{G}(\mathbf{q})\,\delta\mathbf{q} = -\mathbf{d}_g.$$

Adding $\delta\mathbf{q}$ to the system coordinates $\mathbf{q}$ then reduces the constraint errors. While for nonlinear constraints several steps may be required to bring the error to 0, the process usually converges quickly.

Unlike bilateral constraints, unilateral constraints are one-sided, and take effect, or are *engaged*, only when $\mathbf{T}_{CD}$ encounters an inadmissible region. The constraint then acts to prevent further penetration into the region, via the velocity restriction (4.9), and also to push $\mathbf{T}_{CD}$ *out* of the inadmissible region, using a position correction analogous to that used for bilateral constraints.

Whether or not a unilateral constraint is engaged is determined by its *engaged* value E_l, which takes one of the three values: $\{0, 1, -1\}$, and is updated by the joint implementation as the simulation proceeds. A value of 0 means that the constraint is not engaged, and will *not* be included in the joint's unilateral constraint matrix $\mathbf{N}_\text{J}$. Otherwise, if E_l is 1 or -1, then the constraint is engaged and will be included in $\mathbf{N}_\text{J}$, using $\mathbf{N}_l$ if $E_l = 1$, or its negative $-\mathbf{N}_l$ if $E_l = -1$. E_l therefore defines a *sign* for the constraint. General details on how unilateral constraints should be engaged or disengaged are discussed in Section 4.10.2.

A common use of unilateral constraints is to implement limits on joint coordinate values; this also illustrates the utility of E_l. For example, the cylindrical joint mentioned above may have two coordinates, z and θ, describing the translation and rotation along and about the D frame's z axis. Now suppose we wish to bound z, such that

$$z_\text{min} \le z \le z_\text{max}. \tag{4.13}$$

When these limits are violated, a unilateral constraint can be engaged to limit motion along the z axis. A constraint $\mathbf{N}_z$ that will do this is

$$\mathbf{N}_z = (0,0,1,0,0,0).$$

Whenever $z \le z_\text{min}$, using $\mathbf{N}_z$ in (4.9) will ensure that $\dot{\mathbf{z}} \ge 0$ and hence z will not fall further below the lower bound. On the other hand, when $z \ge z_\text{max}$, we want to employ $-\mathbf{N}_z$ in (4.9) to ensure that $\dot{\mathbf{z}} \le 0$. In other words, lower bounds can be enforced by engaging $\mathbf{N}_l$ with $E_l = 1$, while upper bounds can be enforced with $E_l = -1$.

As with bilateral constraints, constraining velocities is not sufficient; it is also necessary to correct position errors, particularly as unilateral constraints are typically not engaged until the inadmissible region is violated. The position correction procedure is the same: for each engaged unilateral constraint, find a distance d_l along its constraint direction

that indicates the distance to the inadmissible region boundary. ArtiSynth will then assemble these d_l into a composite distance vector $\mathbf{d}_n$ for all unilataral constraints, and solve for a system coordinate displacement $\delta\mathbf{q}$ that satisfies

$$\mathbf{N}(\mathbf{q})\,\delta\mathbf{q} \geq -\mathbf{d}_n. \tag{4.14}$$

Because of the inequality direction in (4.14), distances d_l representing penetration into a inadmissible region must be *negative*. For coordinate bounds such as (4.13), we need to use $d_l = z - z_{\min}$ for the lower bound and $d_l = z_{\max} - z$ for the upper bound. Alternatively, if the unilateral constraint has been included into the projection of C onto G and hence into the error term $\hat{\delta}_{err}$, d_l can be computed from

$$d_l = E_l \mathbf{N}_l\, \hat{\delta}_{err}. \tag{4.15}$$

Note that unilateral constraints for coordinate limits are *not* usually incorporated into the G projection; more on this details are given in Section 4.10.4.

As simulation proceeds, the velocity limits imposed by (4.8) and (4.9) are enforced by bilateral and unilateral constraint forces $\mathbf{f}_k$ and $\mathbf{f}_l$ whose magnitudes are given by

$$\mathbf{f}_k = \mathbf{G}_k^T\,\lambda_k, \qquad \mathbf{f}_l = \mathbf{N}_l^T\,\theta_l, \tag{4.16}$$

where λ_k and θ_l are the Lagrange multipliers computed by the mechanical system solver (and are components of λ or θ in (1.8) and (1.6)). $\mathbf{f}_k$ and $\mathbf{f}_l$ are 6 DOF spatial force vectors, or *wrenches* (Section A.5), which like $\mathbf{G}_k$ and $\mathbf{N}_l$ are expressed in frame C. Because $\mathbf{G}_k^T$ and $\mathbf{N}_l^T$ are proportional to spatial wrenches, they are often themselves referred to as *constraint wrenches*, and within the ArtiSynth codebase are described by a Wrench object.

4.10.2 Unilateral constraint engagement

As mentioned above, joints which implement unilateral constraints must monitor $\mathbf{T}_{CD}$ and the joint coordinates as the simulation proceeds and decide when to engage or disengage them.

Engagement is usually easy: a constraint is engaged whenever $\mathbf{T}_{CD}$ or a joint coordinate hits an inadmissible region. The constraint $\mathbf{N}_l$ is itself a spatial vector that is (locally) perpendicular to the inadmissible region boundary, and E_l is chosen to be either 1 or -1 so that $E_l\mathbf{N}_l$ is directed *away* from the inadmissible region. In the remainder of this section, we shall assume $E_l = 1$.

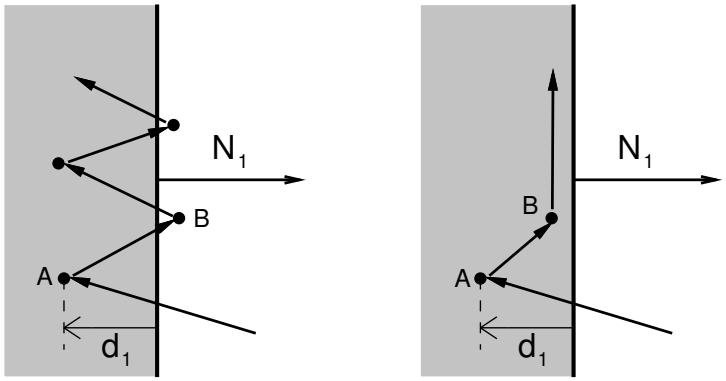


Figure 4.18: Left: A joint configuration at A inside an inadmissible region (gray) is pushed further outside the region than intended B, so that it reenters the region during the next simulation step, resulting in chattering. Right: a deadband solution, in which the position correction is reduced sufficiently so that the joint configuration remains inside the region.

To disengage, we usually want to ensure that the joint configuration is out of the inadmissible region. If we have a constraint $\mathbf{N}_l$, with a local distance d_l defined such that $d_l < 0$ implies the joint is inside the region, then we are out of the region when $d_l > 0$. However, if we use only this as the disengagement criterion, we may encounter a problem known as *chattering*, illustrated in Figure 4.18 (left). An inadmissible region is shown in gray, with a unilateral constraint N_1 perpendicular to its boundary. As simulation proceeds, the joint lands inside the region at an initial point A at the lower left, at a (negative) distance d_1 from the boundary. Ideally the position correction step will move the configuration by $-d_1$ so that it lands right on the region boundary. However, numerical errors and nonlinearities may mean that in fact it lands *outside* the region, at point B. Then on the next step it reenters the region, only to again be pushed out, etc.

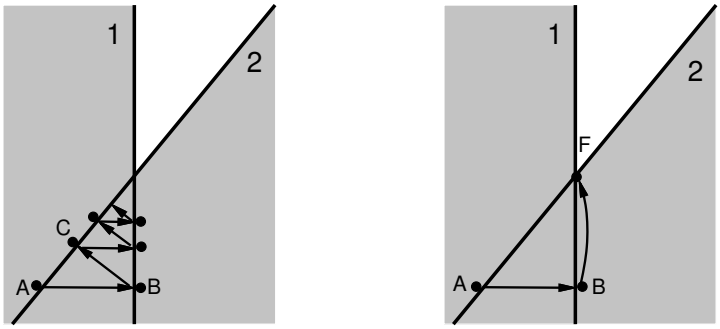


Figure 4.19: Left: constraint oscillation, in which a joint configuration starting at A oscillates between two overlapping inadmissible regions 1 and 2 whose boundaries are not perpendicular. Right: ensuring that constraints stay engaged for more than one simulation step allows the solver to quickly determine a stable solution at F, where the region boundaries intersect.

ArtiSynth implements two solutions to chattering. One is to implement a *deadband*, so that instead of correcting the position by $-d_1$, we correct it by $-d_1 - p_{\text{tol}}$, where p_{tol} is a *penetration tolerance*. This means that the correction will try to leave the joint inside the region by a small amount (Figure 4.18, right) so that chattering is suppressed. The penetration tolerance used depends on the constraint type. Those that are primarily linear use the value of the penetrationTol property, while those that are primarily rotary use the value of the rotaryLimitTol property; both of these are exported as inheritable properties by both MechModel and BodyConnector, with default values computed from the model's overall dimensions.

The second chattering solution is to disengage only when the joint is actively moving away from the region, as determined by $\dot{d}_l > 0$. The disengagement criteria then become

$$d_l > 0 \quad \text{and} \quad \dot{d}_l > 0. \tag{4.17}$$

$\dot{d}_l$ is called the *contact speed* and can be computed from

$$\dot{d}_l = \mathbf{N}_l \hat{\mathbf{v}}_{CD}. \tag{4.18}$$

Another problem, which we call *constraint oscillation*, can occur when we are near two or more overlapping inadmissible regions whose boundaries are not perpendicular. See Figure 4.19 (left), which shows two overlapping regions 1 and 2. The joint starts at point A, inside region 1 but just outside region 2. Since only constraint 1 is engaged, the position correction moves it toward the boundary of 1, overshooting and landing at point B outside of 1 but inside region 2. Constraint 2 now engages, moving the joint to C, where it is past the boundary of 2 but inside 1 again. While the example in the figure converges to the corner where the boundaries of 1 and 2 meet, convergence may be slow and may be prevented entirely by external forcing. While the mechanisms that prevent chattering *may* also prevent oscillation, we find that an additional measure is useful, which is to simply require that *a constraint must be engaged for at least two simulation steps*. The result is shown in Figure 4.19 (right), where after the joint arrives at B, constraint 1 remains engaged along with constraint 2, and the subsequent solution takes the joint directly to point F at the corner where 1 and 2 meet.

4.10.3 Implementing a custom joint

All of the work of computing joint constraints and coordinates, as described in the previous sections, is done within a "coupling" class which is a subclass of RigidBodyCoupling. An instance of this is then embedded within a "joint" class (which is a subclass of JointBase) that supports connections with other bodies, provides rendering, exports various properties, and allows the joint to be attached to a `MechModel`.

For purposes of this discussion, we will assume that these two custom classes are called `CustomCoupling` and `CustomJoint`, respectively. The implementation of `CustomJoint` can be as simple as this:

```
import artisynth.core.mechmodels.ConnectableBody;
import artisynth.core.mechmodels.JointBase;
import maspack.matrix.RigidTransform3d;
```

```
public class CustomJoint extends JointBase {

   public CustomJoint() {
      setCoupling (new CustomCoupling());
   }

   public CustomJoint (
      ConnectableBody bodyA, ConnectableBody bodyB, RigidTransform3d TDW) {
      this(); // call the default constructor
      setBodies(bodyA, bodyB, TDW);
   }
}
```

This creates an instance of `CustomCoupling` and sets it to the (inherited) `myCoupling` attribute inside the default constructor (which is where this normally should be done). Another constructor is provided which uses `setBodies()` to create a joint that is attached to two bodies with the D frame specified in world coordinates. In practice, a joint may also export some properties (such as joint coordinates), provide additional constructors, and implement rendering; one should examine the source code for some existing joints.

4.10.4 Implementing a custom coupling

Implementing a custom coupling constitutes most of the effort in creating a custom joint, since the coupling is responsible for maintaining the constraints $\mathbf{G}_k$ and $\mathbf{N}_l$ that enforce the joint behavior.

Before proceeding, we discuss the coordinate frame in which these constraints are situated. It is often convenient to describe joint constraints with respect to frame C, since rotations are frequently centered there. However, the joint transform $\mathbf{T}_{CD}$ usually contains errors (Section 3.3.1) due to a combination of simulation error and possible joint compliance. To determine these errors, we project C onto another frame G, defined to be the nearest to C that is consistent with the bilateral (and possibly some unilateral) constraints. (This is done by the `projectToConstraints()` method, described below). The result is a joint transform $\mathbf{T}_{GD}$ that is "error free" with respect to bilateral constraints and also consistent with the coordinates (if supported). This makes it convenient to formulate constraints with respect to frame G instead of C, and so this is the convention ArtiSynth uses. In particular, the `updateConstraints()` method, described below, uses $\mathbf{T}_{GD}$, together with the spatial velocity $\hat{\mathbf{v}}_{GD}$ describing the motion of G with respect to C.

An actual custom coupling implementation involves subclassing RigidBodyCoupling and then implementing five abstract methods, the outline of which looks like this:

```
import maspack.matrix.*;
import maspack.spatialmotion.*;
import maspack.util.*;

class CustomCoupling extends RigidBodyCoupling {

   public CustomCoupling() {
      super();
   }

   // Initialize the constraints and coordinates.
   public void initializeConstraints () {
      ...
   }

   // If coordinates are implemented, set TCD from the supplied coordinates.
   public void coordinatesToTCD (RigidTransform3d TCD, VectorNd coords) {
      ...
   }

   // If coordinates are implemented, set their values from TCD.
   public void TCDToCoordinates (VectorNd coords, RigidTransform3d TCD) {
      ...
   }

   // Project TCD to the nearest transform TGD admissible to the
```

```
    // bilateral constraints, and maybe some unilateral constraints.
    public void projectToConstraints (
       RigidTransform3d TGD, RigidTransform3d TCD, boolean updateCoords) {
       ...
    }

    // Update the constraint wrenches, and maybe the engaged
    // and distance settings for some unilateral constraints.
    public void updateConstraints (
       RigidTransform3d TGD, RigidTransform3d TCD, Twist errC,
       Twist velGD, boolean updateEngaged) {
       ...
    }
 }
```

The implementations of these methods are now described in detail.

initializeConstraints()

This method has the signature

```
    public void initializeConstraints ()
```

and is called in the coupling's superclass constructor (i.e., the constructor for `RigidBodyCoupling`). It is responsible for initializing the coupling's constraints and (if supported) coordinates.

Constraints are added using one of the two superclass methods:

```
    RigidBodyConstraint addConstraint (int flags)

    RigidBodyConstraint addConstraint (int flags, Wrench wrench)
```

Each creates a new RigidBodyConstraint and adds it to the coupling's constraint list. *flags* is an or-ed combination of the following flags defined in RigidBodyConstraint:

BILATERAL
: Constraint is bilateral (i.e., an equality). If `BILATERAL` is not specified, the constraint is considered unilateral.

ROTARY
: Constraint primarily restricts rotary motion. If it is unilateral, the joint's rotaryLimitTol property is used for its penetration tolerance.

LINEAR
: Constraint primarily restricts translational motion. If it is unilateral, the joint's penetrationTol property is used for its penetration tolerance.

CONSTANT
: Constraint is constant with respect to frame G. This flag is set automatically if the constraint is created using `addConstraint(flags,wrench)`.

LIMIT
: Constraint is used to enforce limits for a coordinate. This flag is set automatically if the constraint is specified as the limit constraint for a coordinate.

The method `addConstraint(flags,wrench)` takes an additional Wrench argument specifying the (presumed constant) value of the constraint with respect to frame G, and sets the `CONSTANT` flag just described.

Coordinates are added similarly using using one of the two superclass methods:

```
    CoordinateInfo addCoordinate ()

    CoordinateInfo addCoordinate (
       double min, double max, int flags, RigidBodyConstraint limCon)
```

Each creates a new `CoordinateInfo` object (which is an inner class of RigidBodyCoupling), and adds it to the coupling's coordinate list. In the second method, `min` and `max` give the initial range limits, and `limCon`, if non-`null`, specifies a unilateral constraint (previously created using `addConstraint`) for enforcing the limits and causes that constraint's `LIMIT` to be set. The argument `flags` is reserved for future use and should be set to 0. If not specified, the default coordinate limits are $(-\text{inf}, \text{inf})$.

The implementation of `initializeConstraints()` for a coupling that implements a hinge type joint might look like this:

```
public void initializeConstraints () {
   addConstraint (BILATERAL|LINEAR, new Wrench (1, 0, 0, 0, 0, 0));
   addConstraint (BILATERAL|LINEAR, new Wrench (0, 1, 0, 0, 0, 0));
   addConstraint (BILATERAL|LINEAR, new Wrench (0, 0, 1, 0, 0, 0));
   addConstraint (BILATERAL|ROTARY, new Wrench (0, 0, 0, 1, 0, 0));
   addConstraint (BILATERAL|ROTARY, new Wrench (0, 0, 0, 0, 1, 0));
   addConstraint (ROTARY, new Wrench (0, 0, 0, 0, 0, 1));

   addCoordinate (-Math.PI, Math.PI, 0, getConstraint(5));
}
```

Six constraints are specified, with the sixth being a unilateral constraint that enforces the limits on the single coordinate describing the rotation angle. Each constraint and coordinate has an integer index giving the location in its list, in the order it was added. This index can be used to later retrieve the `RigidBodyConstraint` or `CoordinateInfo` object for the constraint or coordinate, using the methods `getConstraint(idx)` or `getCoordinateInfo(idx)`.

> Because `initializeConstraints()` is called in the superclass constructor, member attributes for the custom coupling will not yet be initialized when it is first called. Therefore, the method should not depend on the initial values of non-static member variables. `initializeConstraints()` can also be called later to rebuild the constraints if some defining setting is changed.

coordinatesToTCD()

This method has the signature

```
   public void coordinatesToTCD (RigidTransform3d TCD, VectorNd coords)
```

and is called when needed by the system. If coordinates are supported, then the transform $\mathbf{T}_{CD}$ should be set from the coordinate values supplied in `coords`, and returned in the argument `TCD`. Otherwise, this method should do nothing.

TCDToCoordinates()

This method has the signature

```
   public void TCDToCoordinates (VectorNd coords, RigidTransform3d TCD)
```

and is called when needed by the system. It is the inverse of `coordinatesToTCD()`: if coordinates are supported, then their values should be set from the joint transform $\mathbf{T}_{CD}$ supplied by `TCD` and returned in `coords`. Otherwise, this method should do nothing.

When calling this method, it is assumed that `TCD` is "legal" with respect to the joint's constraints (as defined by `projectToConstraints()`, described next). If this is not the case, then `projectToConstraints()` should be called instead.

One issue that can arise is when a coordinate represents an angle ϕ that has a range greater than 2π. In that case, a common strategy is to compute a nominal value for ϕ, and then add or subtract 2π from it until the resulting value is as close as possible to the *current* value for the angular coordinate. This allows the angle to wrap through its entire range. To implement this, one can use the method

```
   double nearestAngle (double phi)
```

in the coordinate's `CoordinateInfo` object, which finds the angle equivalent to `phi` that is nearest to the current coordinate value.

> Coordinate values computed by this method should *not* be clipped to their ranges.

projectToConstraints()

This method has the signature

```
public void projectToConstraints (
  RigidTransform3d TGD, RigidTransform3d TCD, VectorNd coords)
```

and is called when needed by the system. It is responsible for projecting the joint transform $\mathbf{T}_{CD}$ (supplied by `TCD`) onto the nearest transform $\mathbf{T}_{GD}$ that is valid for the *bilateral* constraints, and returning this in `TGD`. If coordinates are supported and `coords` is non-`null`, then the coordinate values corresponding to $\mathbf{T}_{GD}$ should also be computed and returned in `coords`. The easiest way to do this is to simply call `TCDToCoordinates(TGD,coords)`, although in some cases it may be computationally cheaper to compute both the coordinates and the projection at the same time.

Optionally, the coupling may also extend the projection to include unilateral constraints that are *not* associated with coordinate limits. In particular, this should be done for constraints for which is it desired to have the constraint error included in $\mathbf{T}_{err}$ and the corresponding argument `errC` that is passed to `updateConstraints()`.

updateConstraints()

This method has the signature

```
public void updateConstraints (
  RigidTransform3d TGD, RigidTransform3d TCD, Twist errC,
  Twist velGD, boolean updateEngaged)
```

and is usually called once per simulation time step. It is responsible for:

- Updating the values of all non-constant constraint wrenches, along with their derivatives;
- If `updateEngaged` is `true`, updating the engaged and distance attributes for all unilateral constraints not associated with a coordinate limit.

The method supplies several arguments:

- `TGD`, containing the idealized joint transform $\mathbf{T}_{GD}$ from frame G to D produced by calling `projectToConstraints()`.
- `TCD`, containing the joint transform $\mathbf{T}_{CD}$ from frame C to D and supplied for legacy reasons.
- `errC`, representing the (hopefully small) error transform $\mathbf{T}_{err}$ from frame C to G as a spatial twist vector.
- `velGD`, giving the spatial velocity $\hat{\mathbf{v}}_{GD}$ of frame G with respect to D, as seen in G; this is needed to compute wrench derivatives.
- `updateEngaged`, which requests the updating of unilateral engaged and distance attributes as describe above.

If the coupling supports coordinates, their values will be updated before the method is called so as to correspond to $\mathbf{T}_{GD}$. If needed, a coordinate's value may be obtained from the value attribute of its `CoordinateInfo` object, which may in turn be obtained using `getCoordinateInfo(idx)`. Likewise, `ConstraintInfo` objects for each constraint may be obtaining using `getConstraint(idx)`.

Constraint wrenches correspond to $\mathbf{G}_k$ and $\mathbf{N}_l$ in Section 4.10.1. These, along with their derivatives $\dot{\mathbf{G}}_k$ and $\dot{\mathbf{N}}_l$, are described by the wrenchG and dotWrenchG attributes of each constraint's RigidBodyConstraint object, and may be managed by a variety of methods:

```
Wrench getWrenchG()    // return the reference to wrenchG
void setWrenchG (
   double fx, double fy, double fx, double mx, double my, double mx)
void setWrenchG (Vector3d f, Vector3d m) // either f or m may be null
void setWrenchG (Wrench wr)
void negateWrenchG()
void zeroWrenchG()

Wrench getDotWrenchG() // return the reference to dotWrenchG
void setDotWrenchG (
   double fx, double fy, double fx, double mx, double my, double mx)
void setDotWrenchG (Vector3d f, Vector3d m) // either f or m may be null
void setDotWrenchG (Wrench wr)
void negateDotWrenchG()
void zeroDotWrenchG()
```

dotWrenchG is used in computing the time derivative terms **g** and **n** that appear in (3.8) and (1.6). While these improve the computational accuracy of the simulation, their effect is often small, and so in practice one may be able to omit computing dotWrenchG and instead leave its value as 0.

Wrench information must also be computed for unilateral constraints which implement coordinate limits. While it is not necessary to compute the distance and engaged attributes for these constraints (this is done automatically), it *is* necessary to ensure that the wrench's magnitude is compatible with the coordinate's speed. More precisely, if the coordinate is given by ϕ, then the limit wrench $\mathbf{N}_l$ must have a magnitude such that

$$\dot{\phi} = \mathbf{N}_l \hat{\mathbf{v}}_{GD}. \tag{4.19}$$

As mentioned above, if `updateEngaged` is `true`, the engaged and distance attributes for unilateral constraints not associated with coordinate limits must be updated. These correspond to E_l and d_l in Section 4.10.1, and are contained in the constraint's RigidBodyConstraint object and may be queried using the methods

```
double getDistance()
void setDistance (double d)

int getEngaged()
void setEngaged (int engaged)
```

It is up to `updateConstraints()` to compute the distance, with a negative value denoting penetration into the inadmissible region. If `projectToConstraints()` is implemented so as to account for the constraint, then $\mathbf{T}_{GD}$ will be projected out of the inadmissible region and the distance will be implicitly present $\mathbf{T}_{err}$ and so can be recovered by taking the dot product of the constraint wrench and `velGD`:

```
RigidBodyConstraint cons = getConstraint (3); // assume constraint index is 3
...
double dist = cons.getWrench().dot (velGD);
```

Otherwise, if the constraint is not accounted for in `projectToConstraints()`, the distance must be obtained by other means.

To update engaged, one may use the general convenience method

```
void updateEngaged (
   RigidBodyConstraint cons, double dist,
   double dmin, double dmax, Twist velGD)
```

which sets engaged according to the rules of Section 4.10.2, for an inadmissible region corresponding to `dist` < `dmin` or `dist` > `dmax`. The upper or lower bounds may be removed by setting `dmin` to `-inf` or `max` to `inf`, respectively.

4.10.5 Example: a simple custom joint

An simple model illustrating custom joint creation is provided by

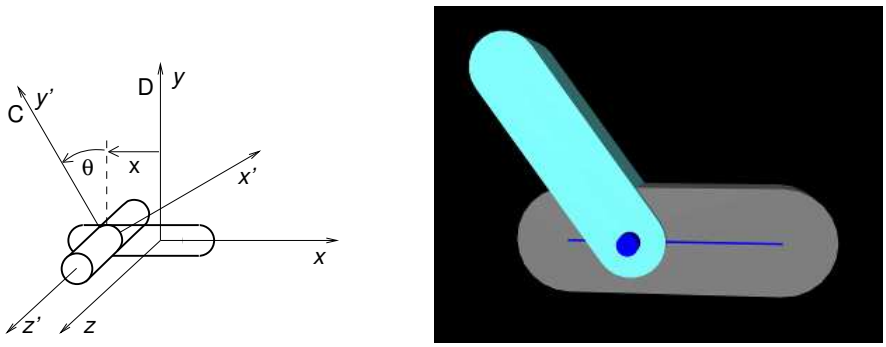


Figure 4.20: Coordinate frames for the `CustomJoint` (left) and the associated `CustomJointDemo` (right).

`artisynth.demos.tutorial.CustomJointDemo`

This implements a joint class defined by `CustomJoint` (also in the package `artisynth.demos.tutorial`), which is actually just a simple implementation of SlottedHingeJoint (Section 3.4.4). Certain details are omitted, such as exporting coordinate values and ranges as properties, and other things are simplified, such as the rendering code. One may consult the source code for `SlottedHingeJoint` to obtain a more complete example.

This section will focus on the implementation of the joint coupling, which is created as an inner class of `CustomJoint` called `CustomCoupling` and which (like all couplings) extends RigidBodyCoupling. The joint itself creates an instance of the coupling in its default constructor, exactly as described in Section 4.10.3.

The coupling allows two DOFs (Figure 4.20, left): translation along the x axis of D (described by the coordinate x), and rotation about the z axis of D (described by the coordinate θ), with $\mathbf{T}_{CD}$ related to the coordinates by (3.24). It implements `initializeConstraints()` as follows:

```
public void initializeConstraints () {
   addConstraint (BILATERAL|LINEAR);
   addConstraint (BILATERAL|LINEAR, new Wrench(0, 0, 1, 0, 0, 0));
   addConstraint (BILATERAL|ROTARY, new Wrench(0, 0, 0, 1, 0, 0));
   addConstraint (BILATERAL|ROTARY, new Wrench(0, 0, 0, 0, 1, 0));
   addConstraint (LINEAR);
   addConstraint (ROTARY, new Wrench(0, 0, 0, 0, 0, 1));

   addCoordinate (-1, 1, 0, getConstraint(4)); // x
   addCoordinate (-2*Math.PI, 2*Math.PI, 0, getConstraint(5)); // theta
}
```

Six constraints are added using `addConstraint()`: two linear bilaterals to restrict translation along the y and z axes of D, two rotary bilaterals to restrict rotation about the x and y axes of D, and two unilaterals to enforce limits on x and θ. Four of the constraints are constant in frame G, and so are initialized with a wrench value. The other two are not constant in G and so will need to be updated in `updateConstraints()`. The coordinates for x and θ are added at the end, using `addCoordinate()`, with default joint limits and a reference to the constraint that will enforce the limit.

The implementations for `coordinatesToTCD()` and `TCDToCoordinates()` simply use (3.24) to compute $\mathbf{T}_{CD}$ from the coordinates, or vice versa:

```
public void coordinatesToTCD (RigidTransform3d TCD, VectorNd coords) {
   double x = coords.get (X_IDX);
   double theta = coords.get (THETA_IDX);
   TCD.setIdentity();
   TCD.p.x = x;
   double c = Math.cos (theta);
   double s = Math.sin (theta);
   TCD.R.m00 = c;
   TCD.R.m11 = c;
   TCD.R.m01 = -s;
   TCD.R.m10 = s;
}
```

```
public void TCDToCoordinates (VectorNd coords, RigidTransform3d TGD) {
   coords.set (X_IDX, TGD.p.x);
   double theta = Math.atan2 (TGD.R.m10, TGD.R.m00);
   coords.set (THETA_IDX, getCoordinateInfo(THETA_IDX).nearestAngle (theta));
}
```

`X_IDX` and `THETA_IDX` are constants defining the coordinate indices for x and θ. In `TCDToCoordinates()`, note the use of the `CoordinateInfo` method `nearestAngle()`, as discussed in Section 4.10.4.

Projecting $\mathbf{T}_{CD}$ onto the error-free $\mathbf{T}_{GD}$ is done by `projectToConstraints()`, implemented as follows:

```
public void projectToConstraints (
   RigidTransform3d TGD, RigidTransform3d TCD, VectorNd coords) {
   TGD.R.set (TCD.R);
   TGD.R.rotateZDirection (Vector3d.Z_UNIT);
   TGD.p.x = TCD.p.x;
   TGD.p.y = 0;
   TGD.p.z = 0;
   if (coords != null) {
      TCDToCoordinates (coords, TGD);
   }
}
```

The translational projection is easy - the y and z components of the translation vector `p` are simply zeroed out. To project the rotation `R`, we use its `rotateZDirection()` method, which applies the shortest rotation aligning its z axis with $(0,0,1)$. The residual rotation will be a rotation in the x-y plane. If `coords` is non-`null` and needs to be computed, we simply call `TCDToCoordinates()`.

Lastly, the implementation for `projectToConstraints()` is as follows:

```
public void updateConstraints (
   RigidTransform3d TGD, RigidTransform3d TCD, Twist errC,
   Twist velGD, boolean updateEngaged) {
   RigidBodyConstraint cons = getConstraint(0); // constraint along y
   double s = TGD.R.m10;  // sin (theta)
   double c = TGD.R.m00;  // cos (theta)
   // constraint wrench along y is constant in D but needs to be
   // transformed to G
   cons.setWrenchG (s, c, 0, 0, 0, 0);
   // derivative term:
   double dotTheta = velGD.w.z;
   cons.setDotWrenchG (c*dotTheta, -s*dotTheta, 0, 0, 0, 0);

   // update x limit constraint if necessary
   cons = getConstraint(4);
   if (cons.getEngaged() != 0) {
      // constraint wrench along x, transformed to G, is (-c, s, 0)
      cons.setWrenchG (c, -s, 0, 0, 0, 0);
      cons.setDotWrenchG (-s*dotTheta, -c*dotTheta, 0, 0, 0, 0);
   }
   // theta limit constraint is constant; no need to do anything
}
```

Only constraints 0 and 4 need to have their wrenches updated, since the rest are constant, and we obtain their constraint objects using `getConstraint(idx)`. Constraint 0 restricts motion along the y axis in D, and while this is constant in D, it is *not* constant in G, which is where the wrench must be situated. The y axis of D as seen in G is the given by the second row of the rotation matrix of $\mathbf{T}_{GD}$, which from (3.24) we see is $(s,c,0)^T$, where $s \equiv \sin(\theta)$ and $c \equiv \cos(\theta)$. We obtain s and c directly from `TGD`, since this has been projected to lie on the constraint surface; alternatively, we could compute them from θ. To obtain the wrench derivative, we note that $\dot{s} = c\dot{\theta}$ and $\dot{c} = -c\dot{\theta}$, and that $\dot{\theta}$ is simply the z component of the angular velocity of G with respect to D, or `velGD.w.z`. The wrench and its derivative are set using the constraint's `setWrenchG()` and `setDotWrenchG()` methods.

The other non-constant constraint is the limit constraint for the x coordinate, which is the x axis of D as seen in G. This is updated similarly, although we only need to do so if the limit constraint is engaged. Since all unilateral constraints are coordinate limits, there is no need to update their distance or engaged attributes as this is done automatically by the system.

Chapter 5

Simulation Control

This section describes different devices which an application may use to control the simulation. These include *control panels* to allow for the interactive adjustment of properties, as well as *agents* which are applied every time step. Agents include *controllers* and *input probes* to supply and modify input parameters at the beginning of each time step, and *monitors* and *output probes* to observe and record simulation results at the end of each time step.

5.1 Control Panels

A *control panel* is an editing panel that allows for the interactive adjustment of component properties.

It is always possible to adjust component properties through the GUI by selecting one or more components and then choosing Edit properties ... in the right-click context menu. However, it may be tedious to repeatedly select the required components, and the resulting panels present the user with *all* properties common to the selection. A control panel allows an application to provide a customized editing panel for selected properties.

5.1.1 General principles

Control panels are implemented by the ControlPanel model component. They can be set up within a model's `build()` method by creating an instance of `ControlPanel`, populating it with widgets for editing the desired properties, and then adding it to the root model using the latter's addControlPanel() method.

One of the most commonly used means of adding widgets to a control panel is the method addWidget(comp,propertyPath), which creates a widget for a property specified by `propertyPath` with respect to the component `comp`. Property paths are discussed in the Section 1.4.2, and can consist solely of a property name, or, for properties located in descendant components, a component path followed by a colon '`:`' and the property name.

Other flavors of `addWidget()` also exist, as described in the API documentation for ControlPanel. In addition to property widgets, any type of `Swing` or `awt` component can be added using the method addWidget(awtcomp).

Control panels can also be created interactively using the GUI; see the section "Control Panels" in the ArtiSynth User Interface Guide.

5.1.2 Example: Creating a simple control panel

An application model showing a control panel is defined in

```
artisynth.demos.tutorial.SimpleMuscleWithPanel
```

This model simply extends `SimpleMuscle` (Section 4.4.2) to provide a control panel for adjusting gravity, the mass and color of the box, and the muscle excitation. The class definition, excluding `include` statements, is shown below:

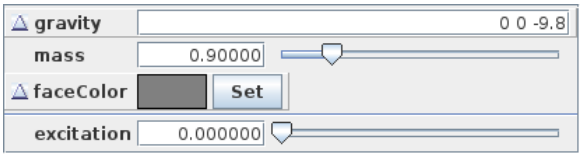


Figure 5.1: Control panel created by the model `SimpleMuscleWithPanel`.

```
1  public class SimpleMuscleWithPanel extends SimpleMuscle {
2     ControlPanel panel;
3
4     public void build (String[] args) throws IOException {
5
6        super.build (args);
7
8        // add control panel for gravity, rigid body mass and color, and excitation
9        panel = new ControlPanel("controls");
10       panel.addWidget (mech, "gravity");
11       panel.addWidget (mech, "rigidBodies/box:mass");
12       panel.addWidget (mech, "rigidBodies/box:renderProps.faceColor");
13       panel.addWidget (new JSeparator());
14       panel.addWidget (muscle, "excitation");
15
16       addControlPanel (panel);
17    }
18 }
```

The `build()` method calls `super.build()` to create the model used by `SimpleMuscle`. It then proceeds to create a `ControlPanel`, populate it with widgets, and add it to the root model (lines 8-15). The panel is given the name `"controls"` in the constructor (line 8); this is its component name and is also used as the title for the panel's window frame. A control panel does not need to be named, but if it is, then that name must be unique among the control panels.

Lines 9-11 create widgets for three properties located relative to the `MechModel` referenced by `mech`. The first is the `MechModel`'s `gravity`. The second is the `mass` of the box, which is a component located relative to `mech` by the path name (Section 1.1.3) `"rigidBodies/box"`. The third is the box's face color, which is the sub-property `faceColor` of the box's `renderProps` property.

Line 12 adds a `JSeparator` to the panel, using the `addWidget()` method that accepts general components, and line 13 adds a widget to control the `excitation` property for `muscle`.

> It should be noted that there are different ways to specify target properties in `addWidget()`. First, component paths may contain numbers instead of names, and so the box's mass property could be specified using `"rigidBodies/0:mass"` instead of `"rigidBodies/box:mass"` since the box's number is 0. Second, if a reference to a sub-component is available, one can specify properties directly with respect to that, instead of indicating the sub-component in the property path. For example, if the box was referenced by a variable `body`, then one could use the construction
>
> ```
> panel.addWidget (body, "mass");
> ```
>
> in place of
>
> ```
> panel.addWidget (mech, "rigidBodies/box:mass");
> ```

To run this example in ArtiSynth, select All demos > tutorial > SimpleMuscleWithPanel from the Models menu. The properties shown in the panel can be adjusted interactively by the user, while the model is either stationary or running.

5.2 Custom properties

Because of the usefulness of properties in creating control panels and probes (Sections 5.1) and Section 5.4), model developers may wish to add their own properties, either to the root model, or to a custom component.

This section provides only a brief summary of how to define properties. Full details are available in the "Properties" section of the Maspack Reference Manual.

5.2.1 Adding properties to a component

As mentioned in Section 1.4, properties are class-specific, and are exported by a class through code contained in the class's definition. Often, it is convenient to add properties to the `RootModel` subclass that defines the application model. In more advanced applications, developers may want to add properties to a custom component.

The property definition steps are:

Declare the property list:

The class exporting the properties creates its own static instance of a PropertyList, using a declaration like

```
static PropertyList myProps = new PropertyList (MyClass.class, MyParent.class );

@Override
public PropertyList getAllPropertyInfo() {
   return myProps;
}
```

where `MyClass` and `MyParent` specify the class types of the exporting class and its parent class. The `PropertyList` declaration creates a new property list, with a copy of all the properties contained in the parent class. If one does *not* want the parent class properties, or if the parent class does not have properties, then one would use the constructor PropertyList(MyClass.class) instead. If the parent class is an ArtiSynth model component (including the `RootModel`), then it will always have its own properties. The declaration of the method `getAllPropertyInfo()` exposes the property list to other classes.

Add properties to the list:

Properties can then be added to the property list, by calling the `PropertyList`'s add() method:

```
PropertyDesc add (String name, String description, Object defaultValue);
```

where `name` contains the name of the property, `description` is a comment describing the property, and `defaultValue` is an object containing the property's default value. This is done inside a static code block:

```
static {
   myProps.add ("stiffness", "spring stiffness", /*defaultValue=*/1);
   myProps.add ("damping", "spring damping", /*defaultValue=*/0);
}
```

Variations on the `add()` method exist for adding *read-only* or *inheritable* properties, or for setting various property options. Other methods allow the property list to be edited.

Declare property accessor functions:

For each property `propXXX` added to the property list, accessor methods of the form

```
void setPropXXX (TypeX value) {
   ...
}

TypeX getPropXXX() {
   TypeX value = ...
   return value;
}
```

must be declared, where `TypeX` is the value associated with the property.

> It is possible to specify different names for the accessor functions in the string argument `name` supplied to the `add()` method. Read-only properties only require a *get* accessor.

5.2.2 Example: a visibility property

An model illustrating the exporting of properties is defined in

```
artisynth.demos.tutorial.SimpleMuscleWithProperties
```

This model extends `SimpleMuscleWithPanel` (Section 4.4.2) to provide a custom property `boxVisible` that is added to the control panel. The class definition, excluding `include` statements, is shown below:

```
1  public class SimpleMuscleWithProperties extends SimpleMuscleWithPanel {
2
3     // internal property list; inherits properties from SimpleMuscleWithPanel
4     static PropertyList myProps =
5        new PropertyList (
6           SimpleMuscleWithProperties.class, SimpleMuscleWithPanel.class);
7
8     // override getAllPropertyInfo() to return property list for this class
9     public PropertyList getAllPropertyInfo() {
10       return myProps;
11    }
12
13    // add new properties to the list
14    static {
15       myProps.add ("boxVisible", "box is visible", false);
16    }
17
18    // declare property accessors
19    public boolean getBoxVisible() {
20       return box.getRenderProps().isVisible();
21    }
22
23    public void setBoxVisible (boolean visible) {
24       RenderProps.setVisible (box, visible);
25    }
26
27    public void build (String[] args) throws IOException {
28
29       super.build (args);
30
31       panel.addWidget (this, "boxVisible");
32       panel.pack();
33    }
34 }
```

First, a property list is created for the application class `SimpleMuscleWithProperties.class` which contains a copy of all the properties from the parent class `SimpleMuscleWithPanel.class` (lines 4-6). This property list is made visible by overriding `getAllPropertyInfo()` (lines 9-11). The `boxVisible` property itself is then added to the property list (line 15), and accessor functions for it are declared (lines 19-25).

The `build()` method calls `super.build()` to perform all the model creation required by the super class, and then adds an additional widget for the `boxVisible` property to the control panel.

To run this example in ArtiSynth, select All demos > tutorial > SimpleMuscleWithProperties from the Models menu. The control panel will now contain an additional widget for the property `boxVisible` as shown in Figure 5.2. Toggling this property will make the box visible or invisible in the viewer.

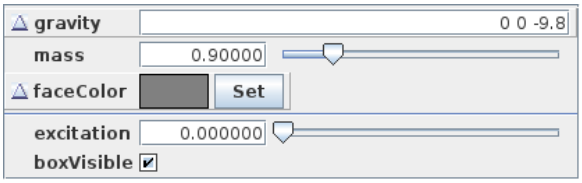


Figure 5.2: Control panel created by the model `SimpleMuscleWithProperties`, showing the newly defined property `boxVisible`.

5.3 Controllers and monitors

Application models can define custom *controllers* and *monitors* to control input values and monitor output values as a simulation progresses. Controllers are called every time step immediately before the `advance()` method, and monitors are called immediately after (Section 1.1.4). An example of controller usage is provided by ArtiSynth's inverse modeling feature, which uses an internal controller to estimate the actuation signals required to follow a specified motion trajectory.

More precise details about controllers and monitors and how they interact with model advancement are given in the ArtiSynth Reference Manual.

5.3.1 Implementation

Applications may declare whatever controllers or monitors they require and then add them to the root model using the methods addController() and addMonitor(). They can be any type of ModelComponent that implements the Controller or Monitor interfaces. For convenience, most applications simply subclass the default implementations ControllerBase or MonitorBase and then override the necessary methods.

The primary methods associated with both controllers and monitors are:

```
public void initialize (double t0);

public void apply (double t0, double t1);

public boolean isActive();
```

`apply(t0, t1)` is the "business" method and is called once per time step, with `t0` and `t1` indicating the start and end times t_0 and t_1 associated with the step. `initialize(t0)` is called whenever an application model's state is set (or reset) at a particular time t_0. This occurs when a simulation is first started or after it is reset (with $t_0 = 0$), and also when the state is reset at a waypoint or during adaptive stepping.

`isActive()` controls whether a controller or monitor is active; if `isActive()` returns `false` then the `apply()` method will not be called. The default implementations ControllerBase and MonitorBase, via their superclass ModelAgentBase, also provide a `setActive()` method to control this setting, and export it as the property active. This allows controller and monitor activity to be controlled at run time.

> To enable or disable a controller or monitor at run time, locate it in the navigation panel (under the RootModel's controllers or monitors list), chose Edit properties ... from the right-click context menu, and set the active property as desired.

Controllers and monitors may be associated with a particular model (among the list of models owned by the root model). This model may be set or queried using

```
void setModel (Model m);

Model getModel();
```

If associated with a model, `apply()` will be called immediately before (for controllers) or after (for monitors) the model's `advance()` method. If not associated with a model, then `apply()` will be called before or after the advance of *all* the models owned by the root model.

Controllers and monitors may also contain *state*, in which case they should implement the relevant methods from the HasState interface.

Typical actions for a controller include setting input forces or excitation values on components, or specifying the motion trajectory of parametric components (Section 3.1.3). Typical actions for a monitor include observing or recording the motion profiles or constraint forces that arise from the simulation.

When setting the position and/or velocity of a dynamic component that has been set to be parametric (Section 3.1.3), a controller should not set its position or velocity directly, but should instead set its *target position* and/or *target velocity*, since this allows the solver to properly interpolate the position and velocity during the time step. The methods to set or query target positions and velocities for Point-based components are

```
setTargetPosition (Point3d pos);
Point3d getTargetPosition ();       // read-only

setTargetVelocity (Vector3d vel);
Vector3d getTargetVelocity ();      // read-only
```

while for Frame-based components they are

```
setTargetPosition (Point3d pos);
setTargetOrientation (AxisAngle axisAng);
setTargetPose (RigidTransform3d TFW);
Point3d getTargetPosition ();       // read-only
AxisAngle getTargetOrientation ();  // read-only
RigidTransform3d getTargetPose();   // read-only

setTargetVelocity (Twist vel);
Twist getTargetVelocity ();         // read-only
```

5.3.2 Example: A controller to move a point

A model showing an application-defined controller is defined in

`artisynth.demos.tutorial.SimpleMuscleWithController`

This simply extends `SimpleMuscle` (Section 4.4.2) and adds a controller which moves the fixed particle `p1` along a circular path. The complete class definition is shown below:

```
1  package artisynth.demos.tutorial;
2
3  import java.io.IOException;
4  import maspack.matrix.*;
5
6  import artisynth.core.modelbase.*;
7  import artisynth.core.mechmodels.*;
8  import artisynth.core.gui.*;
9
10 public class SimpleMuscleWithController extends SimpleMuscleWithPanel
11 {
12    private class PointMover extends ControllerBase {
13
14       Point myPnt;      // point to be moved
15       Point3d myPos0;   // initial point position
16
17       public PointMover (Point pnt) {
18          myPnt = pnt;
19          myPos0 = new Point3d (pnt.getPosition());
20       }
21
22       public void apply (double t0, double t1) {
23          double ang = Math.PI*t1/2;            // angle associated with time t1
24          Point3d pos = new Point3d (myPos0);
```

```
25          pos.x += 0.5*Math.sin (ang);          // compute position for t1 ...
26          pos.z += 0.5*(1-Math.cos (ang));
27          myPnt.setTargetPosition (pos);        // ... and the set point's target
28       }
29    }
30
31    public void build (String[] args) throws IOException {
32       super.build (args);
33
34       addController (new PointMover (p1));
35       // increase model bounding box for the viewer
36       mech.setBounds (-1, 0, -1, 1, 0, 1);
37    }
38
39 }
```

A controller called `PointMover` is defined by extending `ControllerBase` and overriding the `apply()` method. It stores the point to be moved in `myPnt`, and the initial position in `myPos0`. The `apply()` method computes a target position for the point that starts at `myPos0` and then moves in a circle in the $x-z$ plane with an angular velocity of $\pi/2$ rad/sec (lines 22-28).

The `build()` method calls `super.build()` to create the model used by `SimpleMuscle`, and then creates an instance of `PointMover` to move particle `p1` and adds it to the root model (line 34). The viewer bounds are updated to make the circular motion more visible (line 36).

To run this example in ArtiSynth, select All demos > tutorial > SimpleMuscleWithController from the Models menu. When the model is run, the fixed particle `p1` will trace out a circular path in the $x-z$ plane.

5.4 Probes

In addition to controllers and monitors, applications can also attach streams of data, known as *probes*, to input and output values associated with the simulation. Probes derive from the same base class ModelAgentBase as controllers and monitors, but differ in that

1. They are associated with an explicit time interval during which they are applied;
2. They can have an attached file for supplying input data or recording output data;
3. They are displayable in the ArtiSynth *timeline* panel.

A probe is applied (by calling its `apply()` method) only for time steps that fall within its time interval. This interval can be set and queried using the following methods:

```
setStartTime (double t0);
setStopTime (double t1);
setInterval (double t0, double t1);

double getStartTime();
double getStopTime();
```

The probe's attached file can be set and queried using:

```
setAttachedFileName (String fileName);
String getAttachedFileName ();
```

where `fileName` is a string giving the file's path name.

Details about the timeline display can be found in the section "The Timeline" in the ArtiSynth User Interface Guide.

There are two types of probe: *input probes*, which are applied at the beginning of each simulation step before the controllers, and *output probes*, which are applied at the end of the step after the monitors.

While applications are free to construct any type of probe by subclassing either InputProbe or OutputProbe, most applications utilize either NumericInputProbe or NumericOutputProbe, which explicitly implement streams of numeric data which are connected to properties of various model components. The remainder of this section will focus on numeric probes.

As with controllers and monitors, probes also implement a `isActive()` method that indicates whether or not the probe is active. Probes that are not active are not invoked. Probes also provide a `setActive()` method to control this setting, and export it as the property active. This allows probe activity to be controlled at run time.

> To enable or disable a probe at run time, locate it in the navigation panel (under the RootModel's inputProbes or outputProbes list), chose Edit properties ... from the right-click context menu, and set the active property as desired.
>
> Probes can also be enabled or disabled in the timeline, by either selecting the probe and invoking activate or deactivate from the right-click context menu, or by clicking the track mute button (which activates or deactivates all probes on that track).

5.4.1 Numeric probe structure

Numeric probes are associated with:

- *A vector of temporally-interpolated numeric data*;
- *One or more properties* to which the probe is bound and which are either set by the numeric data (input probes), or used to set the numeric data (output probes).

The numeric data is implemented internally by a NumericList, which stores the data as a series of vector-valued knot points at prescribed times t_k and then interpolates the data for an arbitrary time t using an interpolation scheme provided by Interpolation.

Some of the numeric probe methods associated with the interpolated data include:

```
int getVsize();                       // returns the size of the data vector
setInterpolationOrder (Order order); // sets the interpolation scheme
Order getInterpolationOrder();        // returns the interpolation scheme

VectorNd getData (double t);          // interpolates data for time t
NumericList getNumericList();         // returns the underlying NumericList
```

Interpolation schemes are described by the enumerated type `Interpolation.Order` and presently include:

Step

Values at time t are set to the values of the closest knot point k such that $t_k \leq t$.

Linear

Values at time t are set by linear interpolation of the knot points $(k, k+1)$ such that $t_k \leq t \leq t_{k+1}$.

Parabolic

Values at time t are set by quadratic interpolation of the knots $(k-1, k, k+1)$ such that $t_k \leq t \leq t_{k+1}$.

Cubic

Values at time t are set by cubic Catmull interpolation of the knots $(k-1, \ldots, k+2)$ such that $t_k \leq t \leq t_{k+1}$.

Each property bound to a numeric probe must have a value that can be mapped onto a scalar or vector value. Such properties are know as *numeric properties*, and whether or not a value is numeric can be tested using NumericConverter.isNumeric(value).

By default, the total number of scalar and vector values associated with all the properties should equal the size of the interpolated vector (as returned by getVsize()). However, it is possible to establish more complex mappings between the property values and the interpolated vector. These mappings are beyond the scope of this document, but are discussed in the sections "General input probes" and "General output probes" of the ArtiSynth User Interface Guide.

5.4.2 Creating probes in code

This section discusses how to create numeric probes in code. They can also be created and added to a model graphically, as described in the section "Adding and Editing Numeric Probes" in the ArtiSynth User Interface Guide.

Numeric probes have a number of constructors and methods that make it relatively easy to create instances of them in code. For NumericInputProbe, there is the constructor

```
NumericInputProbe (ModelComponent c, String propPath, String filePath);
```

which creates a `NumericInputProbe`, binds it to a property located relative to the component `c` by `propPath`, and then attaches it to the file indicated by `filePath` and loads data from this file (see Section 5.4.4). The probe's start and stop times are specified in the file, and its vector size is set to match the size of the scalar or vector value associated with the property.

To create a probe attached to multiple properties, one may use the constructor

```
NumericInputProbe (ModelComponent c, String propPaths[], String filePath);
```

which binds the probe to multiple properties specified relative to `c` by `propPaths`. The probe's vector size is set to the sum of the sizes of the scalar or vector values associated with these properties.

For NumericOutputProbe, one may use the constructor

```
NumericOutputProbe (ModelComponent c, String propPath, String filePath, double ↩
 sample);
```

which creates a `NumericOutputProbe`, binds it to the property `propPath` located relative to `c`, and then attaches it to the file indicated by `filePath`. The argument `sample` indicates the *sample time* associated with the probe, in seconds; a value of 0.01 means that data will be added to the probe every 0.01 seconds. If `sample` is specified as -1, then the sample time will default to the maximum step size associated with the root model.

To create an output probe attached to multiple properties, one may use the constructor

```
NumericOutputProbe (
   ModelComponent c, String propPaths[], String filePath, double sample);
```

> As the simulation proceeds, an output probe will accumulate data, but this data will not be saved to any attached file until the probe's `save()` method is called. This can be requested in the GUI for all probes by clicking on the Save button in the timeline toolbar, or for specific probes by selecting them in the navigation panel (or the timeline) and then choosing Save data in the right-click context menu.

Output probes created with the above constructors have a default interval of [0, 1]. A different interval may be set using `setInterval()`, `setStartTime()`, or `setStopTime()`.

5.4.3 Example: probes connected to SimpleMuscle

A model showing a simple application of probes is defined in

```
artisynth.demos.tutorial.SimpleMuscleWithProbes
```

This extends `SimpleMuscle` (Section 4.4.2) to add an input probe to move particle `p1` along a defined path, along with an output probe to record the velocity of the frame marker. The complete class definition is shown below:

```
1 package artisynth.demos.tutorial;
2
3 import java.io.IOException;
4 import maspack.matrix.*;
5 import maspack.util.PathFinder;
6
7 import artisynth.core.modelbase.*;
```

```
8  import artisynth.core.mechmodels.*;
9  import artisynth.core.probes.*;
10
11 public class SimpleMuscleWithProbes extends SimpleMuscleWithPanel
12 {
13    public void createInputProbe() throws IOException {
14       NumericInputProbe p1probe =
15          new NumericInputProbe (
16             mech, "particles/p1:targetPosition",
17             PathFinder.getSourceRelativePath (this, "simpleMuscleP1Pos.txt"));
18       p1probe.setName("Particle Position");
19       addInputProbe (p1probe);
20    }
21
22    public void createOutputProbe() throws IOException {
23       NumericOutputProbe mkrProbe =
24          new NumericOutputProbe (
25             mech, "frameMarkers/0:velocity",
26             PathFinder.getSourceRelativePath (this, "simpleMuscleMkrVel.txt"),
27             0.01);
28       mkrProbe.setName("FrameMarker Velocity");
29       mkrProbe.setDefaultDisplayRange (-4, 4);
30       mkrProbe.setStopTime (10);
31       addOutputProbe (mkrProbe);
32    }
33
34    public void build (String[] args) throws IOException {
35       super.build (args);
36
37       createInputProbe ();
38       createOutputProbe ();
39       mech.setBounds (-1, 0, -1, 1, 0, 1);
40    }
41
42 }
```

The input and output probes are added using the custom methods `createInputProbe()` and `createOutputProbe()`. At line 14, `createInputProbe()` creates a new input probe bound to the `targetPosition` property for the component `particles/p1` located relative to the `MechModel` `mech`. The same constructor attaches the probe to the file `simpleMuscleP1Pos.txt`, which is read to load the probe data. The format of this and other probe data files is described in Section 5.4.4. The method PathFinder.getSourceRelativePath() is used to locate the file relative to the source directory for the application model (see Section 2.6). The probe is then given the name `"Particle Position"` (line 18) and added to the root model (line 19).

Similarly, `createOutputProbe()` creates a new output probe which is bound to the `velocity` property for the component `particles/0` located relative to `mech`, is attached to the file `simpleMuscleMkrVel.txt` located in the application model source directory, and is assigned a sample time of 0.01 seconds. This probe is then named `"FrameMarker Velocity"` and added to the root model.

The `build()` method calls `super.build()` to create everything required for `SimpleMuscle`, calls `createInputProbe()` and `createOutputProbe()` to add the probes, and adjusts the `MechModel` viewer bounds to make the resulting probe motion more visible.

To run this example in ArtiSynth, select All demos > tutorial > SimpleMuscleWithProbes from the Models menu. After the model is loaded, the input and output probes should appear on the timeline (Figure 5.3). Expanding the probes should display their numeric contents, with the knot points for the input probe clearly visible. Running the model will cause particle `p1` to trace the trajectory specified by the input probe, while the velocity of the marker is recorded in the output probe. Figure 5.4 shows an expanded view of both probes after the simulation has run for about six seconds.

5.4.4 Data file format

The data files associated with numeric probes are ASCII files containing two lines of header information followed by a set of knot points, one per line, defining the numeric data. The time value (relative to the probe's start time) for each knot point can be specified explicitly at the start of the each line, in which case the file takes the following format:

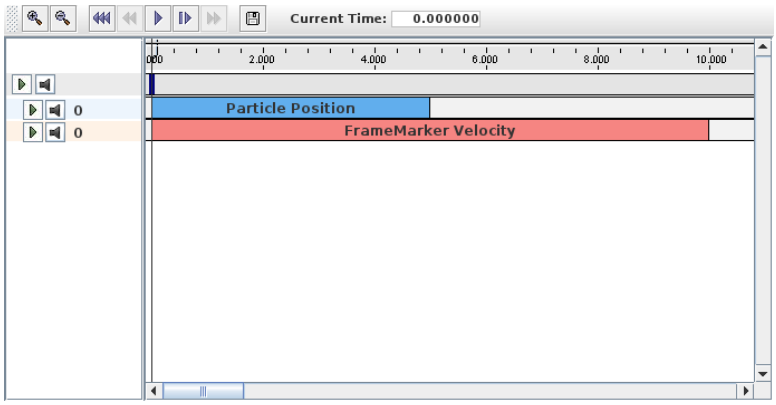


Figure 5.3: Timeline view of the probes created by SimpleMuscleWithProbes.

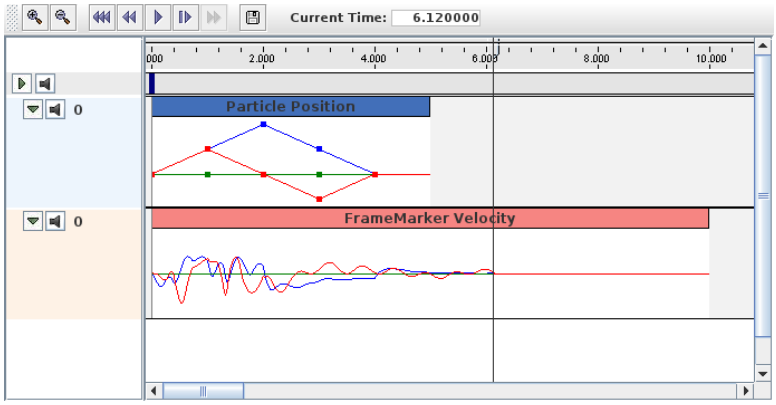


Figure 5.4: Expanded view of the probes after `SimpleMuscleWithProbes` has run for about 6 seconds, showing the data accumulated in the output probe "`FrameMarker Velocity`".

```
startTime stopTime scale
interpolation vsize explicit
t0 val00 val01 val02 ...
t1 val10 val11 val12 ...
t0 val20 val21 val22 ...
...
```

Knot point information begins on line 3, with each line being a sequence of numbers giving the knot's time followed by n values, where n is the vector size of the probe (i.e., the value returned by `getVsize()`).

Alternatively, time values can be implicitly specified starting at 0 (relative to the probe's start time) and incrementing by a uniform `timeStep`, in which case the file assumes a second format:

```
startTime stopTime scale
interpolation vsize timeStep
val00 val01 val02 ...
val10 val11 val12 ...
val20 val21 val22 ...
...
```

For both formats, `startTime`, `startTime`, and `scale` are numbers giving the probe's start and stop time in seconds and `scale` gives the scale factor (which is typically 1.0). `interpolation` is a word describing how the data should be interpolated between knot points and is the string value of `Interpolation.Order` as described in Section 5.4.1 (and which is typically `Linear`, `Parabolic`, or `Cubic`). `vsize` is an integer giving the probe's vector size.

The last entry on the second line is either a number specifying a (uniform) time step for the knot points, in which case the file assumes the second format, or the keyword `explicit`, in which case the file assumes the first format.

As an example, the file used to specify data for the input probe in the example of Section 5.4.3 looks like the following:

```
0 4.0 1.0
Linear 3 explicit
0.0   0.0 0.0 0.0
1.0   0.5 0.0 0.5
2.0   0.0 0.0 1.0
3.0 -0.5 0.0 0.5
4.0   0.0 0.0 0.0
```

Since the data is uniformly spaced beginning at 0, it would also be possible to specify this using the second file format:

```
0 4.0 1.0
Linear 3 1.0
 0.0 0.0 0.0
 0.5 0.0 0.5
 0.0 0.0 1.0
-0.5 0.0 0.5
 0.0 0.0 0.0
```

5.4.5 Adding probe data in-line

It is also possible to specify input probe data directly in code, instead of reading it from a file. For this, one would use the constructor

```
    NumericInputProbe (ModelComponent c, String propPath, double t0, double t1);
```

which creates a `NumericInputProbe` with the specified property and with start and stop times indicated by `t0` and `t1`. Data can then be added to this probe using the method

```
    addData (double[] data, double timeStep);
```

where `data` is an array of knot point data. This contains the same knot point information as provided by a file (Section 5.4.4), arranged in row-major order. Times values for the knots are either implicitly specified, starting at 0 (relative to the probe's start time) and increasing uniformly by the amount specified by `timeStep`, or are explicitly specified at the beginning of each knot if `timeStep` is set to the built-in constant `NumericInputProbe.EXPLICIT_TIME`. The size of the `data` array should then be either $n * m$ (implicit time values) or $(n+1) * m$ (explicit time values), where n is the probe's vector size and m is the number of knots.

As an example, the data for the input probe in Section 5.4.3 could have been specified using the following code:

```
      NumericInputProbe p1probe =
         new NumericInputProbe (
            mech, "particles/p1:targetPosition", 0, 5);
      p1probe.addData (
         new double[] {
            0.0,  0.0, 0.0, 0.0,
            1.0,  0.5, 0.0, 0.5,
            2.0,  0.0, 0.0, 1.0,
            3.0, -0.5, 0.0, 0.5,
            4.0,  0.0, 0.0, 0.0 },
            NumericInputProbe.EXPLICIT_TIME);
```

When specifying data in code, the interpolation defaults to `Linear` unless explicitly specified using `setInterpolationOrder()`, as in, for example:

```
    probe.setInterpolationOrder (Order.Cubic);
```

5.4.6 Numeric monitor probes

In some cases, it may be useful for an application to deploy an output probe in which the data, instead of being collected from various component properties, is generated by a function within the probe itself. This ability is provided by a NumericMonitorProbe, which generates data using its generateData(vec,t,trel) method. This evaluates a vector-valued function of time at either the absolute time `t` or the probe-relative time `trel` and stores the result in the vector `vec`, whose size equals the vector size of the probe (as returned by getVsize()). The probe-relative time `trel` is determined by

$$\text{trel} = (\text{t} - \text{tstart})/\text{scale} \tag{5.1}$$

where `tstart` and `scale` are the probe's start time and scale factors as returned by getStartTime() and getScale().

As described further below, applications have several ways to control how a `NumericMonitorProbe` creates data:

- Provide the probe with a DataFunction using the setDataFunction(func) method;
- Override the generateData(vec,t,trel) method;
- Override the apply(t) method.

The application is free to generate data in any desired way, and so in this sense a `NumericMonitorProbe` can be used similarly to a `Monitor`, with one of the main differences being that the data generated by a `NumericMonitorProbe` can be automatically displayed in the ArtiSynth GUI or written to a file.

The DataFunction interface declares an `eval()` method,

```
void eval (VectorNd vec, double t, double trel)
```

that for `NumericMonitorProbe`s evaluates a vector-valued function of time, where the arguments take the same role as for the monitor's generateData() method. Applications can declare an appropriate `DataFunction` and set or query it within the probe using the methods

```
void setDataFunction (DataFunction func);

DataFunction getDataFunction();
```

The default implementation `generateData()` checks to see if a data function has been specified, and if so, uses that to generate the probe data. Otherwise, if the probe's data function is `null`, the data is simply set to zero.

To create a `NumericMonitorProbe` using a supplied `DataFunction`, an application will create a generic probe instance, using one of its constructors such as

```
NumericMonitorProbe (vsize, fileName, startTime, stopTime, interval);
```

and then define and instantiate a `DataFunction` and pass it to the probe using `setDataFunction()`. It is not necessary to supply a file name (i.e., `fileName` can be `null`), but if one is provided, then the probe's data can be saved to that file.

A complete example of this is defined in

```
artisynth.demos.tutorial.SinCosMonitorProbe
```

the listing for which is:

```
1  package artisynth.demos.tutorial;
2
3  import maspack.matrix.*;
4  import maspack.util.Clonable;
5
6  import artisynth.core.workspace.RootModel;
7  import artisynth.core.probes.NumericMonitorProbe;
8  import artisynth.core.probes.DataFunction;
9
10 /**
11  * Simple demo using a NumericMonitorProbe to generate sine and cosine waves.
```

```
 */
public class SinCosMonitorProbe extends RootModel {

   // Define the DataFunction that generates a sine and a cosine wave
   class SinCosFunction implements DataFunction, Clonable {

      public void eval (VectorNd vec, double t, double trel) {
         // vec should have size == 2, one for each wave
         vec.set (0, Math.sin (t));
         vec.set (1, Math.cos (t));
      }

      public Object clone() throws CloneNotSupportedException {
         return (SinCosFunction)super.clone();
      }
   }

   public void build (String[] args) {

      // Create a NumericMonitorProbe with size 2, file name "sinCos.dat", start
      // time 0, stop time 10, and a sample interval of 0.01 seconds:
      NumericMonitorProbe sinCosProbe =
         new NumericMonitorProbe (/*vsize=*/2, "sinCos.dat", 0, 10, 0.01);

      // then set the data function:
      sinCosProbe.setDataFunction (new SinCosFunction());
      addOutputProbe (sinCosProbe);
   }
}
```

In this example, the `DataFunction` is implemented using the class `SinCosFunction`, which also implements Clonable and the associated `clone()` method. This means that the resulting probe will also be duplicatable within the GUI. Alternatively, one could implement `SinCosFunction` by extending DataFunctionBase, which implements `Clonable` by default. Probes containing `DataFunction`s which are *not* `Clonable` will not be duplicatable.

When the example is run, the resulting probe output is shown in the timeline image of Figure 5.5.

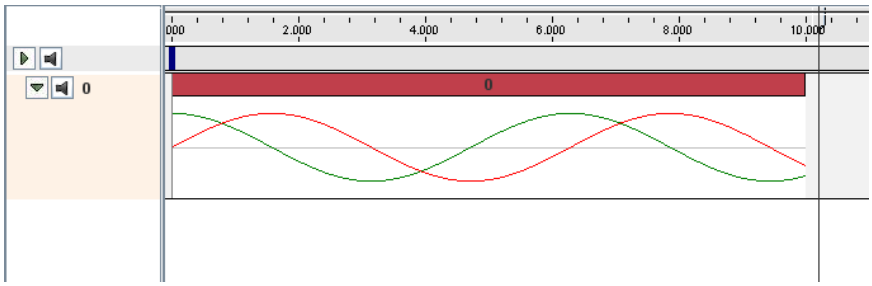


Figure 5.5: Output from a `NumericMonitorProbe` which generates sine and cosine waves.

As an alternative to supplying a `DataFunction` to a generic `NumericMonitorProbe`, an application can instead subclass `NumericMonitorProbe` and override either its generateData(vec,t,trel) or apply(t) methods. As an example of the former, one could create a subclass as follows:

```
   class SinCosProbe extends NumericMonitorProbe {

      public SinCosProbe (
         String fileName, double startTime, double stopTime, double interval) {
         super (2, fileName, startTime, stopTime, interval);
      }

      public void generateData (VectorNd vec, double t, double trel) {
         vec.set (0, Math.sin (t));
         vec.set (1, Math.cos (t));
```

```
      }
   }
```

Note that when subclassing, one must also create constructor(s) for that subclass. Also, NumericMonitorProbes which don't have a DataFunction set are considered to be clonable by default, which means that the clone() method may also need to be overridden if cloning requires any special handling.

5.4.7 Numeric control probes

In other cases, it may be useful for an application to deploy an input probe which takes numeric data, and instead of using it to modify various component properties, instead calls an internal method to directly modify the simulation in any way desired. This ability is provided by a NumericControlProbe, which applies its numeric data using its applyData(vec,t,trel) method. This receives the numeric input data via the vector vec and uses it to modify the simulation for either the absolute time t or probe-relative time trel. The size of vec equals the vector size of the probe (as returned by getVsize()), and the probe-relative time trel is determined as described in Section 5.4.6.

A NumericControlProbe is the Controller equivalent of a NumericMonitorProbe, as described in Section 5.4.6. Applications have several ways to control how they apply their data:

- Provide the probe with a DataFunction using the setDataFunction(func) method;
- Override the applyData(vec,t,trel) method;
- Override the apply(t) method.

The application is free to apply data in any desired way, and so in this sense a NumericControlProbe can be used similarly to a Controller, with one of the main differences being that the numeric data used can be automatically displayed in the ArtiSynth GUI or read from a file.

The DataFunction interface declares an eval() method,

```
void eval (VectorNd vec, double t, double trel)
```

that for NumericControlProbes applies the numeric data, where the arguments take the same role as for the monitor's applyData() method. Applications can declare an appropriate DataFunction and set or query it within the probe using the methods

```
void setDataFunction (DataFunction func);

DataFunction getDataFunction();
```

The default implementation applyData() checks to see if a data function has been specified, and if so, uses that to apply the probe data. Otherwise, if the probe's data function is null, the data is simply ignored and the probe does nothing.

To create a NumericControlProbe using a supplied DataFunction, an application will create a generic probe instance, using one of its constructors such as

```
NumericControlProbe (vsize, data, startTime, stopTime, timeStep);

NumericControlProbe (fileName);
```

and then define and instantiate a DataFunction and pass it to the probe using setDataFunction(). The latter constructor creates the probe and reads in both the data and timing information from the specified file.

A complete example of this is defined in

```
artisynth.demos.tutorial.SpinControlProbe
```

the listing for which is:

```java
package artisynth.demos.tutorial;

import maspack.matrix.RigidTransform3d;
import maspack.matrix.VectorNd;
import maspack.util.Clonable;
import maspack.interpolation.Interpolation;

import artisynth.core.mechmodels.MechModel;
import artisynth.core.mechmodels.RigidBody;
import artisynth.core.mechmodels.Frame;
import artisynth.core.workspace.RootModel;
import artisynth.core.probes.NumericControlProbe;
import artisynth.core.probes.DataFunction;

/**
 * Simple demo using a NumericControlProbe to spin a Frame about the z
 * axis.
 */
public class SpinControlProbe extends RootModel {

   // Define the DataFunction that spins the body
   class SpinFunction implements DataFunction, Clonable {

      Frame myFrame;
      RigidTransform3d myTFW0; // initial frame to world transform

      SpinFunction (Frame frame) {
         myFrame = frame;
         myTFW0 = new RigidTransform3d (frame.getPose());
      }

      public void eval (VectorNd vec, double t, double trel) {
         // vec should have size == 1, giving the current spin angle
         double ang = Math.toRadians(vec.get(0));
         RigidTransform3d TFW = new RigidTransform3d();
         TFW.R.mulRpy (ang, 0, 0);
         myFrame.setPose (TFW);
      }

      public Object clone() throws CloneNotSupportedException {
         return super.clone();
      }
   }

   public void build (String[] args) {

      MechModel mech = new MechModel ("mech");
      addModel (mech);

      // Create a parametrically controlled rigid body to spin:
      RigidBody body = RigidBody.createBox ("box", 1.0, 1.0, 0.5, 1000.0);
      mech.addRigidBody (body);
      body.setDynamic (false);

      // Create a NumericControlProbe with size 1, initial spin data
      // with time step 2.0, start time 0, and stop time 8.
      NumericControlProbe spinProbe =
         new NumericControlProbe (
            /*vsize=*/1,
            new double[] { 0.0, 90.0, 0.0, -90.0, 0.0 },
            2.0, 0.0, 8.0);
      // set cubic interpolation for a smoother result
      spinProbe.setInterpolationOrder (Interpolation.Order.Cubic);
      // then set the data function:
```

```
65         spinProbe.setDataFunction (new SpinFunction(body));
66         addInputProbe (spinProbe);
67      }
68   }
```

This example creates a simple box and then uses a `NumericControlProbe` to spin it about the z axis, using a `DataFunction` implementation called `SpinFunction`. A clone method is also implemented to ensure that the probe will be duplicatable in the GUI, as described in Section 5.4.6. A single channel of data is used to control the orientation angle of the box about z, as shown in Figure 5.6.

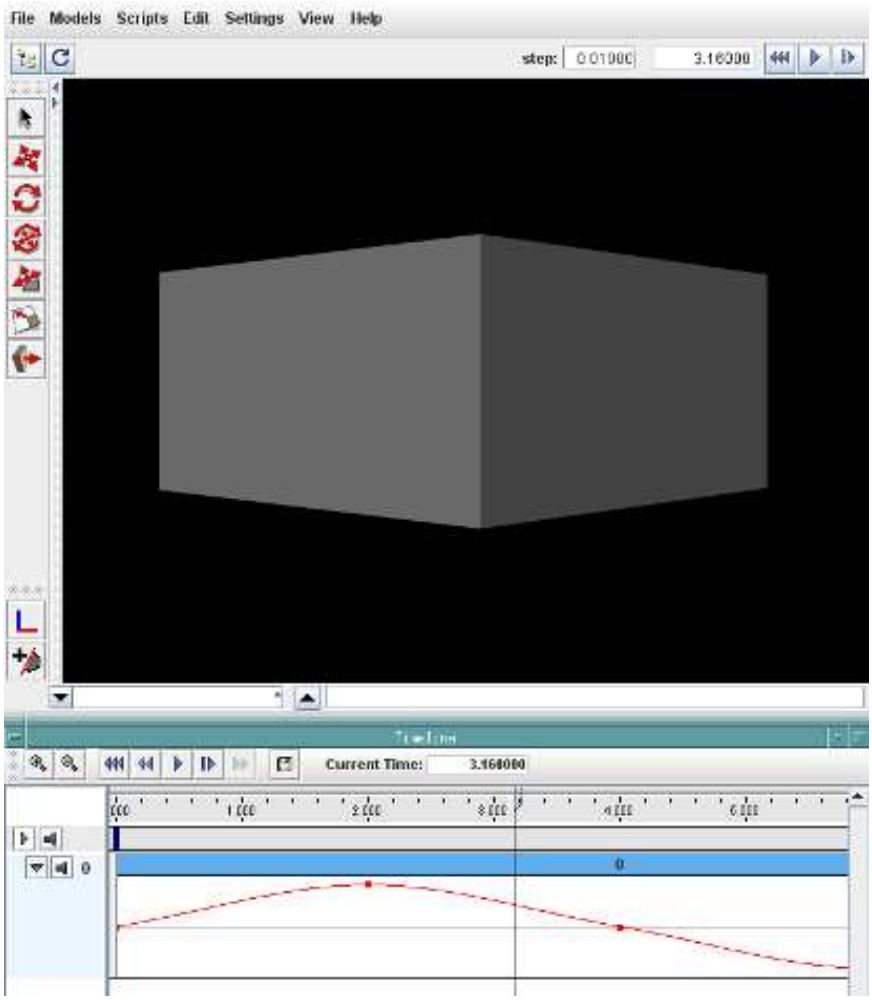


Figure 5.6: Screen shot of the `SpinControlDemo`, showing the numeric data in the timeline.

Alternatively, an application can subclass `NumericControlProbe` and override either its applyData(vec,t,trel) or apply(t) methods, as described for `NumericMonitorProbes` (Section 5.4.6).

5.5 Application-Defined Menu Items

Application models can define custom *menu items* that appear under the `Application` menu in the main ArtiSynth menu bar.

This can be done by implementing the interface HasMenuItems in either the `RootModel` or any of its top-level components (e.g., models, controllers, probes, etc.). The interface contains a single method

```
   public boolean getMenuItems(List<Object> items);
```

which, if the component has menu items to add, should append them to `items` and return `true`.

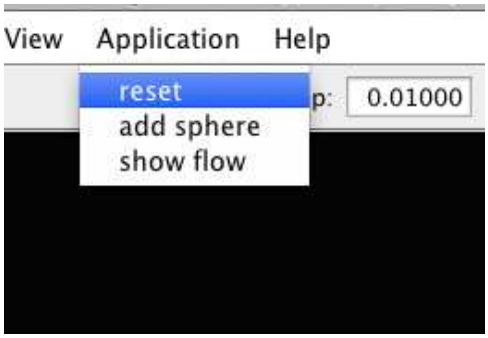


Figure 5.7: Application-defined menu items appearing under the ArtiSynth menu bar.

The `RootModel` and all models derived from ModelBase implement `HasMenuItems` by default, but with `getMenuItems()` returning `false`. Models wishing to add menu items should override this default declaration. Other component types, such as controllers, need to explicitly implement `HasMenuItems`.

> Note: the Application menu will only appear if `getMenuItems()` returns `true` for either the `RootModel` or one or more of its top-level components.

getMenuItems() will be called each time the Application menu is selected, so the menu itself is created on demand and can be varied to suite the current system state. In general, it should return items that are capable of being displayed inside a Swing `JMenu`; other items will be ignored. The most typical item is a Swing `JMenuItem`. The convenience method createMenuItem(listener,text,toolTip) can be used to quickly create menu items, as in the following code segment:

```
public boolean getMenuItems(List<Object> items) {
   items.add (GuiUtils.createMenuItem (this, "reset", ""));
   items.add (GuiUtils.createMenuItem (this, "add sphere", ""));
   items.add (GuiUtils.createMenuItem (this, "show flow", ""));
   return true;
}
```

This creates three menu items, each with `this` specified as an `ActionListener` and no tool-tip text, and appends them to `items`. They will then appear under the Application menu as shown in Figure 5.7.

To actually execute the menu commands, the items returned by `getMenuItems()` need to be associated with an `ActionListener` (defined in `java.awt.event`), which supplies the method `actionPerformed()` which is called when the menu item is selected. Typically the `ActionListener` is the component implementing `HasMenuItems`, as was assumed in the example declaration of `getMenuItems()` shown above. `RootModel` and other models derived from `ModelBase` implement `ActionListener` by default, with an empty declaration of `actionPerformed()` that should be overridden as required. A declaration of `actionPerformed()` capable of handling the menu example above might look like this:

```
public void actionPerformed (ActionEvent event) {
   String cmd = event.getActionCommand();
   if (cmd.equals ("reset")) {
      resetModel();
   }
   else if (cmd.equals ("add sphere")) {
      addSphere();
   }
   else if (cmd.equals ("show flow")) {
      showFlow();
   }
}
```

Chapter 6

Finite Element Models

This chapter details how to construct three-dimensional finite element models, and how to couple them with the other simulation components described in previous sections (e.g. particles and rigid bodies). Finite element *muscles*, which have additional properties that allow them to contract given activation signals, are discussed in Section 6.9. An example FEM model of the masseter, coupled to a rigid jaw and maxilla, is shown in Figure 6.1.

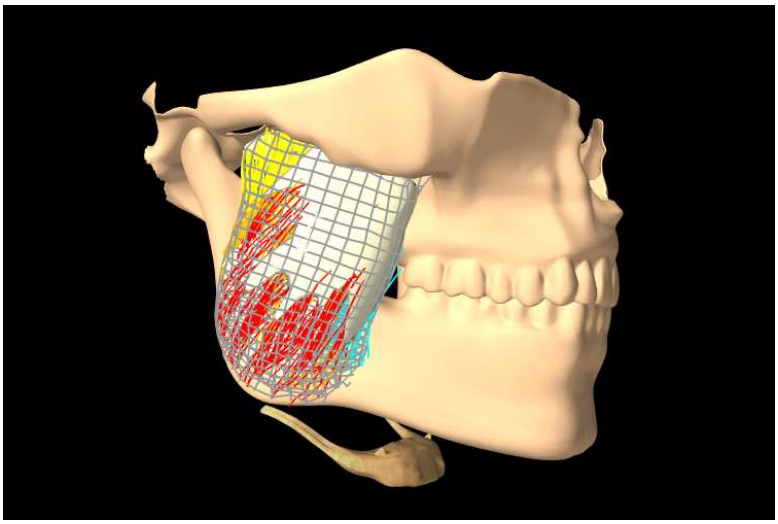


Figure 6.1: Finite element model of the masseter, coupled to the jaw and maxilla.

6.1 Overview

The finite element method (FEM) is a numerical technique used for solving a system of partial differential equations (PDEs) over some domain. The general approach is to divide the domain into a set of building blocks, referred to as *elements*. These partition the space, and form local domains over which the system of equations can be locally approximated. The corners of these elements, the *nodes*, become control points in a discretized system. The solution is then assumed to be smoothly interpolated across the elements based on values determined at the nodes. Using this discretization, the differential system is converted into an algebraic one, which is often linearized and solved iteratively.

In ArtiSynth, the PDEs considered are the governing equations of continuum mechanics: the conservation of mass, momentum, and energy. To complete the system, a *constitutive equation* is required that describes the stress-strain response of the material. This constitutive equation is what distinguishes between material types. The domain is the three-dimensional space that the model occupies. This must be divided into small elements which accurately represent the geometry. Within each element, the PDEs are sampled at a set of points, referred to as *integration points*, and terms are numerically integrated to form an algebraic system to solve.

The purpose of the rest of this chapter is to describe the construction and use of finite elements models within ArtiSynth. It does not further discuss the mathematical framework or theory. For an in-depth coverage of the nonlinear finite element method, as applied to continuum mechanics, the reader is referred to the textbook by Bonet and Wood [3].

6.1.1 FemModel3d

The basic type of finite element model is implemented in the class FemModel3d. This class controls some properties that are used by the model as a whole. The key ones that affect simulation dynamics are:

Property	Description
`density`	The density of the model
`material`	An object that describes the material's *constitutive law* (i.e. its stress-strain relationship).
`particleDamping`	Proportional damping associated with the particle-like motion of the FEM nodes.
`stiffnessDamping`	Proportional damping associated with the system's stiffness term.

These properties can be set and retrieved using the methods

```
setDensity (double density);     // sets the density
double getDensity ();            // gets the density

setMaterial (FemMaterial mat);   // sets the FEM's material
FemMaterial getMaterial ();      // gets the FEM's material

setParticleDamping (double d);   // sets the particle (mass) damping coefficient
double getParticleDamping ();    // gets the particle (mass) damping coefficient

setStiffnessDamping (double d); // sets the stiffness damping coefficient
double getStiffnessDamping ();  // gets the stiffness damping coefficient
```

Keep in mind that ArtiSynth is essentially "unitless" (Section 4.2), so it is the responsibility of the developer to ensure that all properties are specified in a compatible way.

The density of the model is used to compute the mass distribution throughout the volume. Note that we use a *diagonally lumped mass matrix* (DLMM) formulation, so the mass is assumed to be concentrated at the location of the discretized FEM nodes. To allow for a spatially-varying density, densities can be explicitly set for individual elements, or masses can be explicitly set for individual nodes.

The FEM's material property is a delegate object used to compute stress and stiffness within individual elements. It handles the *constitutive* component of the model, as described in more detail in Section 6.1.3. In addition to the main material defined for the model, it is also possible set a material on a per-element basis, and to define additional materials which augment the behavior of the main materials (Section 6.8).

The two damping parameters are related to *Rayleigh damping*, which is used to dissipate energy within finite element models. There are two proportional damping terms: one related to the system's mass, and one related to stiffness. The resulting damping force applied is

$$\mathbf{f}_d = -(d_M\mathbf{M} + d_K\mathbf{K})\mathbf{v}, \tag{6.1}$$

where d_M is the value of `particleDamping`, d_K is the value of `stiffnessDamping`, $\mathbf{M}$ is the FEM model's lumped mass matrix, $\mathbf{K}$ is the FEM's stiffness matrix, and $\mathbf{v}$ is the concatenated vector of FEM node velocities. Since the lumped mass matrix is diagonal, the mass-related component of damping can be applied separately to each FEM node. Thus, the mass component reduces to the same system as Equation (3.3), which is why it is referred to as "particle damping".

6.1.2 Component Structure

Each FemModel3d contains several lists of sub-components:

`nodes`
: The particle-like dynamic components of the model. These lie at the corners of the elements and carry all the mass (due to DLMM formulation).

`elements`
: The volumetric model elements. These define the 3D sub-units over which the system is numerically integrated.

`shellElements`
: The shell elements. These define additional 2D sub-units over which the system is numerically integrated.

`meshes`

The geometry in the model. This includes the surface mesh, and any other embedded geometries.

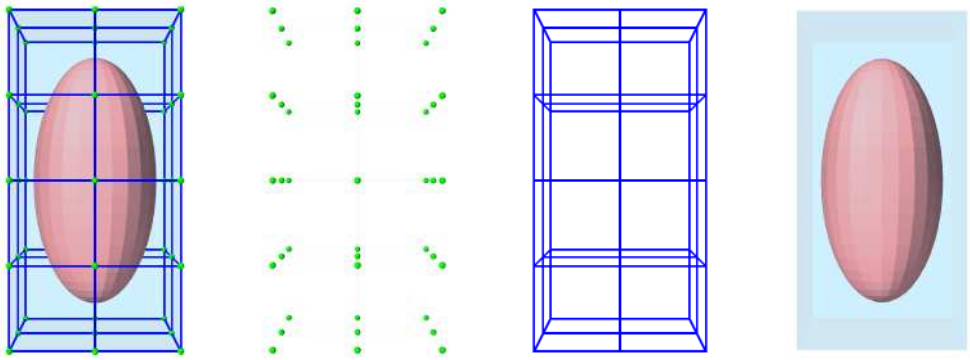
`materials`

Optional additional materials which can be added to the model to augment the behavior of the model's material property. This is described in more detail in Section 6.8.

`fields`

Optional *field* components which can be used to interpolate application-defined quantities over the FEM model's domain. Fields are described in detail in Section 6.10.

The `nodes`, `elements` and `meshes` components are illustrated in Figure 6.2.



(a) FEM model (b) Nodes (c) Elements (d) Geometry

Figure 6.2: Sub-components of FemModel3d.

6.1.2.1 Nodes

The set of nodes belong to a finite element model can be obtained by the method

```
PointList<FemNode3d> getNodes();  // returns list of FEM nodes
```

Nodes are implemented in the class FemNode3d, which is a subclass of Particle (Section 3.1). They are the main dynamic components of the finite element model. The key properties affecting simulation dynamics are:

Property	Description
`restPosition`	The initial position of the node.
`position`	The current position of the node.
`velocity`	The current velocity of the node.
`mass`	The mass of the node.
`dynamic`	Whether the node is considered dynamic or parametric (e.g. boundary condition).

Each of these properties has corresponding `getXxx()` and `setXxx(...)` functions to access and modify them.

The `restPosition` property defines the node's position in the FEM model's "natural" or "undeformed" state. Rest positions are used to compute an initial configuration for the model, from which strains are determined. A node's rest position can be updated in code using the method: FemNode3d.setRestPosition(Point3d).

> If any node's rest positions are changed, the current values for stress and stiffness will become invalid. They can be manually updated using the method FemModel3d.updateStressAndStiffness() for the parent model. Otherwise, stress and stiffness will be automatically updated at the beginning of the next time step.

The properties `position` and `velocity` give the node's current 3D state. These are common to all point-like particles, as is the `mass` property. Here, however, `mass` represents the lumped mass of the immediately surrounding material. Its

value is initialized by equally dividing mass contributions from each adjacent element, given their densities. For a finer control of spatially-varying density, node masses can be set manually after FEM creation.

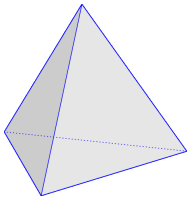
The FEM node's `dynamic` property specifies whether or not the node is considered when computing the dynamics of the system. If not, it is treated as being parametrically controlled. This has implications when setting boundary conditions (Section 6.1.4).

6.1.2.2 Elements

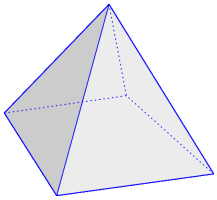
Elements are the 3D volumetric spatial building blocks of the domain. Within each element, the displacement (or strain) field is interpolated from displacements at nodes:

$$\mathbf{u}(\mathbf{x}) = \sum_{i=1}^{N} \phi_i(\mathbf{x})\mathbf{u}_i, \tag{6.2}$$

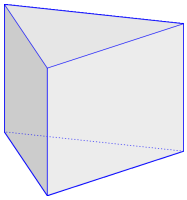
where $\mathbf{u}_i$ is the displacement of the ith node that is adjacent to the element, and $\phi_i(\cdot)$ is referred to as the *shape function* (or *basis function*) associated with that node. Elements are classified by their shape, number of nodes, and shape function order (Table 6.1). ArtiSynth supports the following element types:



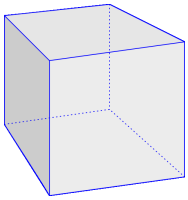
TetElement, QuadtetElement



PyramidElement, QuadpyramidElement



WedgeElement, QuadwedgeElement



HexElement, QuadhexElement

Table 6.1: Supported element types

Element Type	# Nodes	Order	# Integration Points
TetElement	4	linear	1
PyramidElement	5	linear	5
WedgeElement	6	linear	6
HexElement	8	linear	8
QuadtetElement	10	quadratic	4
QuadpyramidElement	13	quadratic	5
QuadwedgeElement	15	quadratic	9
QuadhexElement	20	quadratic	14

The base class for all of these is FemElement3d. A numerical integration is performed within each element to create the (tangent) stiffness matrix. This integration is performed by evaluating the stress and stiffness at a set of *integration points* within each element, and applying numerical quadrature. The list of elements in a model can be obtained with the method

```
RenderableComponentList<FemElement3d> getElements();
```

All objects of type FemElement3d have the following properties:

Property	Description
`density`	Density of the element
`material`	An object that describes the *constitutive law* within the element (i.e. its stress-strain relationship).

If left unspecified, the element's `density` is inherited from the containing `FemModel3d` object. When set, the mass of the element is computed and divided amongst all its nodes, updating the lumped mass matrix.

Each element's' `material` property is also inherited by default from the containing `FemModel3d`. Specifying a material here allows for spatially-varying material properties across the model. Materials will be discussed further in Section 6.1.3.

6.1.2.3 Shell elements

Shell elements are additional 2D spatial building blocks which can be added to a model. They are typically used to model structures which are too thin to be easily represented by 3D volumetric elements, or to provide additional internal stiffness within a set of volumetric elements.

ArtiSynth presently supports the following shell element types, with the number of nodes, shape function order, and integration point count described in Table 6.2:

ShellTriElement ShellQuadElement

Table 6.2: Supported shell element types

Element Type	# Nodes	Order	# Integration Points
ShellTriElement	3	linear	9 (3 if membrane)
ShellQuadElement	4	linear	8 (4 if membrane)

The base class for all shell elements is ShellElement3d, which contains the same density and material properties as FemElement3d, as well as the additional property defaultThickness, whose use will be described below.

The list of shell elements in a model can be obtained with the method

```
RenderableComponentList<ShellElement3d> getShellElements();
```

Both the volumetric elements (`FemElement3d`) and the shell elements (`ShellElement3d`) derive from the base class FemElement3dBase. To obtain *all* the elements in an FEM model, both shell and volumetric, one may use the method

```
ArrayList<FemElement3dBase> getAllElements();
```

Each shell element can actually be instantiated in two forms:

- As a *regular* shell element, which has a bending stiffness;
- As a *membrane* element, which does not have bending stiffness.

Regular shell elements are implemented using the same *extensible director* formulation used by FEBio [9], and more specifically the front/back node formulation [10]. Each node associated with a (regular) shell element is assigned a *director*, which is a 3D vector providing a normal direction and virtual thickness at that node (Figure 6.3). This virtual thickness allows us to continue to use 3D materials to provide the constitutive laws that determine the shell's stress/strain response, including its bending behavior. It also allows us to continue to use the element's density to determine its mass.

Director information is automatically assigned to a FemNode3d whenever one or more regular shell elements is connected to it. This information includes both the current value of the director, its *rest* value, and its velocity, with the difference between the first two determining the element's bending strain. These quantities can be queried using the methods

```
Vector3d getDirector ();         // return the current director value
void getDirector (Vector3d dir);

Vector3d getRestDirector ();     // return the rest director value
void getRestDirector (Vector3d dir);

Vector3d getDirectorVel ();      // return the director velocity

boolean hasDirector();           // does this node have a director?
```

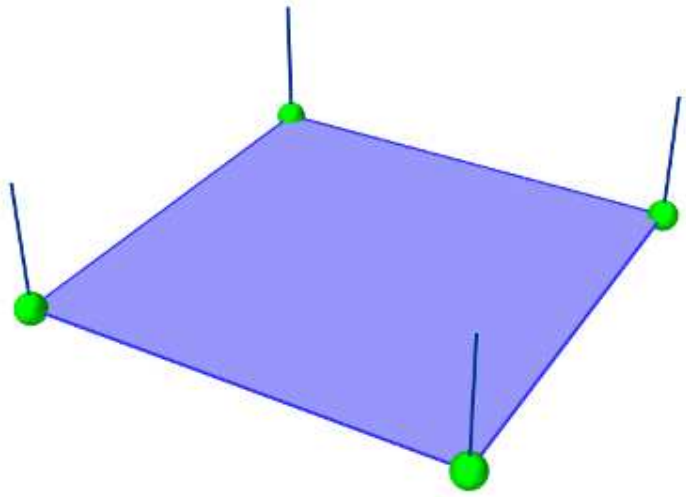


Figure 6.3: `ShellQuadElement` with the directors (dark blue lines) visible at the nodes.

For nodes which are not connected to regular shell elements, and therefore do not have director information assigned, these methods all return a zero-valued vector.

If not otherwise specified, the current and rest director values are computed automatically from the surrounding (regular) shell elements, with their value **d** being computed from

$$\mathbf{d} = \sum t_i \, \mathbf{n}_i$$

where t_i is the value of the defaultThickness property and $\mathbf{n}_i$ is the surface normal of the i-th surrounding regular shell element. However, if necessary, it is also possible to explicitly assign these values, using the methods

```
setDirector (Vector3d dir);     // set the current director

setRestDirector (Vector3d dir); // set the rest director
```

> ArtiSynth FEM nodes can currently support only one director, which is shared by all regular shell elements associated with that node. This effectively means that all such elements must belong to the same "surface", and that two intersecting surfaces cannot share the same nodes.

As indicated above, shell elements can also be instantiated as membrane elements, which do not exhibit bending stiffness and therefore do not require director information. The regular/membrane distinction is specified in the element's constructor. For example, `ShellTriElement` and `ShellQuadElement` each have constructors with the signatures:

```
ShellTriElement (FemNode3d n0, FemNode3d n1, FemNode3d n2,
  double thickness, boolean membrane);

ShellQuadElement (FemNode3d n0, FemNode3d n1, FemNode3d n2, FemNode3d n3,
  double thickness, boolean membrane);
```

The `thickness` argument specifies the defaultThickness property, while `membrane` determines whether or not the element is a membrane element.

While membrane elements do not require explicit director information stored at the nodes, they do make use of an *inferred* director that is parallel to the element's surface normal, and has a constant length equal to the element's defaultThickness property. This gives the element a virtual volume, which (as with regular elements) is used to determine 3D strains and to compute the element's mass from it's density.

6.1.2.4 Meshes

The geometry associated with a finite element model consists of a collection of meshes (e.g. PolygonalMesh, PolylineMesh, PointMesh) that move along with the model in a way that maintains the shape function interpolation

equation (6.2) at each vertex location. These geometries can be used for visualizations, or for physical interactions like collisions. However, they have no physical properties themselves. FEM geometries will be discussed in more detail in Section 6.3. The list of meshes can be obtained with the method

```
MeshComponentList<FemMeshComp> getMeshComps();
```

6.1.3 Materials

The stress-strain relationship within each element is defined by a "material" delegate object, implemented by a subclass of FemMaterial. This material object is responsible for implementing the functions

```
void computeStressAndTangent (...)
```

which computes the stress tensor and (optionally) the tangent stiffness matrix at each integration point, based on the current local deformation at that point.

The default material type is LinearMaterial, where stress is related to strain through:

$$\sigma(\mathbf{x}) = D\,\varepsilon(\mathbf{x}), \tag{6.3}$$

$$\text{where} \quad D = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix}, \quad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)},$$

σ is the standard 6×1 stress vector, ε is the strain vector, E is the Young's Modulus, and ν is Poisson's ratio. This linear material uses a corotational formulation, so rotations are removed per element before computing the strain [13]. To enable or disable this corotational formulation, use LinearMaterial.setCorotated(boolean).

All material models, including linear and nonlinear, are available in the package `artisynth.core.materials`. A list of common materials is provided in Table 6.3. Those that are subclasses of IncompressibleMaterial allow for incompressibility.

Table 6.3: Commonly used FEM materials

Material	Parameters	
LinearMaterial	E	Young's modulus
	ν	Poisson's ratio
	corotated	corotational formulation
StVenantKirchoffMaterial	E	Young's modulus
	ν	Poisson's ratio
NeoHookeanMaterial	E	Young's modulus
	ν	Poisson's ratio
IncompNeoHookeanMaterial	G	shear modulus
	κ	bulk modulus
MooneyRivlinMaterial	$C_{10}, C_{01}, C_{20}, C_{02}$	distortional parameters
	κ	bulk modulus
OgdenMaterial	$\mu_1, \ldots, \mu_6$	material parameters
	$\alpha_1, \ldots, \alpha_6$	
	κ	bulk modulus

6.1.4 Boundary conditions

Boundary conditions can be implemented in one of several ways:

1. Explicitly setting FEM node positions/velocities
2. Attaching FEM nodes to other dynamic components

3. Enabling collisions

To enforce an explicit (Dirichlet) boundary condition for a set of nodes, their `dynamic` property must be set to `false`. This notifies ArtiSynth that the state of these nodes (both position and velocity) will be controlled parametrically. By disabling dynamics, a fixed boundary condition is applied. For a moving boundary, positions and velocities of the boundary nodes must be explicitly set every timestep. This can be accomplished with either a Controller (see Section 5.3) or an InputProbe (see Section 5.4). Note that both the position *and* velocity of the nodes should be explicitly set for consistency.

Another type of supported boundary condition is to attach FEM nodes to other components, including particles, springs, rigid bodies, and locations within other FEM elements. Here, the node is still considered dynamic, but its motion is coupled to that of the attached component through a constraint mechanism. Attachments will be discussed further in Section 6.4.

Finally, the boundary of an FEM can be constrained by enabling collisions with other components. This will be covered in Section 6.11.

6.2 FEM model creation

Creating a finite element model in ArtiSynth typically follows the pattern:

```
// Create and add main MechModel
MechModel mech = new MechModel("mech");
addModel(mech);

// Create FEM
FemModel3d fem = new FemModel3d("fem");

/* ... Setup FEM structure and properties ... */

// Add FEM to model
mech.addModel(fem);
```

The main code block for the FEM setup should do the following:

- Build the node/element structure
- Set physical properties
  - density
  - damping
  - material
- Set boundary conditions
- Set render properties

Building the FEM structure can be done with the use of factory methods for simple shapes, by loading external files, or by writing code to manually assemble the nodes and elements.

6.2.1 Factory methods

For simple shapes such as beams and ellipsoids, there are factory methods to automatically build the node and element structure. These methods are found in the FemFactory class. Some common methods are

```
FemFactory.createGrid(...)          // basic beam
FemFactory.createCylinder(...)      // cylinder
FemFactory.createTube(...)          // hollowed cylinder
FemFactory.createEllipsoid(...)     // ellipsoid
FemFactory.createTorus(...)         // torus
```

The inputs specify the dimensions, resolution, and potentially the type of element to use. The following code creates a basic beam made up of hexahedral elements:

```
// Create FEM
FemModel3d beam = new FemModel3d("beam");

// Build FEM structure
double[] size = {1.0, 0.25, 0.25};  // widths
int[] res = {8, 2, 2};              // resolution (# elements)

FemFactory.createGrid(beam, FemElementType.Hex,
   size[0], size[1], size[2],
   res[0], res[1], res[2]);

/* ... Set FEM properties ... */

// Add FEM to model
mech.addModel(beam);
```

6.2.2 Loading external FEM meshes

For more complex geometries, volumetric meshes can be loaded from external files. A list of supported file types is provided in Table 6.4. To load a geometry, an appropriate file reader must be created. Readers capable of reading FEM models implement the interface FemReader, which has the method

```
readFem( FemModel3d fem )   // populates the FEM based on file contents
```

Additionally, many `FemReader` classes have static methods to handle the loading of files for convenience.

Table 6.4: Supported FEM geometry files

Format	File extensions	Reader	Writer
ANSYS	.node, .elem	AnsysReader	AnsysWriter
TetGen	.node, .ele	TetGenReader	TetGenWriter
Abaqus	.inp	AbaqusReader	AbaqusWriter
VTK (ASCII)	.vtk	VtkAsciiReader	–

The following code snippet demonstrates how to load a model using the AnsysReader.

```
// Create FEM
FemModel3d tongue = new FemModel3d("tongue");

// Read FEM from file
try {
   // Get files relative to THIS class
   String nodeFileName = PathFinder.getSourceRelativePath(this,
                         "data/tongue.node");
   String elemFileName = PathFinder.getSourceRelativePath(this,
                         "data/tongue.elem");

   AnsysReader.read(tongue, nodeFileName, elemFileName);

} catch (IOException ioe) {
   // Wrap error, fail to create model
   throw new RuntimeException("Failed to read model", ioe);
}

// Add to model
mech.addModel(tongue);
```

The method PathFinder.getSourceRelativePath() is used to find a path within the ArtiSynth source tree that is relative to the current model's source file (Section 2.6). Note the try-catch block. Most of these readers throw an `IOException` if the read fails.

6.2.3 Generating from surfaces

There are two ways an FEM model can be generated from a surface: by using a FEM mesh generator, and by extruding a surface along its normal direction.

ArtiSynth has the ability to interface directly with the TetGen library (http://tetgen.org) to create a tetrahedral volumetric mesh given a closed and manifold surface. The main Java class for calling TetGen directly is TetgenTessellator. The tessellator has several advanced options, allowing for the computation of convex hulls, and for adding points to a volumetric mesh. For simply creating an FEM from a surface, there is a convenience routine within FemFactory that handles both mesh generation and constructing a `FemModel3d`:

```
// Create an FEM from a manifold mesh with a given quality
FemFactory.createFromMesh( PolygonalMesh mesh, double quality );
```

If `quality` > 0, then points will be added in an attempt to bound the maximum radius-edge ratio (see the `-q` switch for TetGen). According to the TetGen documentation, the algorithm *usually* succeeds for a quality ratio of 1.2.

It's also possible to create thin layer of elements by extruding a surface along its normal direction.

```
// Create an FEM by extruding a surface
FemFactory.createExtrusion(
      FemModel3d model, int nLayers, double layerThickness, double zOffset,
      PolygonalMesh surface);
```

For example, to create a two-layer slice of elements centered about a surface of a tendon mesh, one might use

```
// Load the tendon surface mesh
PolygonalMesh tendonSurface = new PolygonalMesh("tendon.obj");

// Create the tendon
FemModel3d tendon = new FemModel3d("tendon");
int layers = 2;              // 2 layers
double thickness = 0.0005;   // 0.5 mm layer thickness
double offset = thickness;   // center the layers about the surface

// Create the extrusion
FemFactory.createExtrusion( tendon, layers, thickness, offset, tendonSurface );
```

For this type of extrusion, triangular faces become wedge elements, and quadrilateral faces become hexahedral elements.

> Note: for extrusions, no care is taken to ensure element quality; if the surface has a high curvature relative to the total extrusion thickness, then some elements will be inverted.

6.2.4 Building elements in code

A finite element model's structure can also be manually constructed in code. FemModel3d has the methods:

```
addNode ( FemNode3d );        // add a node to the model
addElement ( FemElement3d )   // add an element to the model
```

For an element to successfully be added, all its nodes must already have been added to the model. Nodes can be constructed from a 3D location, and elements from an array of nodes. A convenience routine is available in FemElement3d that automatically creates the appropriate element type given the number of nodes (Table 6.1):

```
// Creates an element using the supplied nodes
FemElement3d FemElement3d.createElement( FemNode3d[] nodes );
```

Be aware of node orderings when supplying nodes. For linear elements, ArtiSynth uses a clockwise convention with respect to the outward normal for the first face, followed by the opposite node(s). To determine the correct ordering for a particular element, check the coordinates returned by the function FemElement3dBase.getNodeCoords(). This returns the concatenated coordinate list for an "ideal" element of the given type.

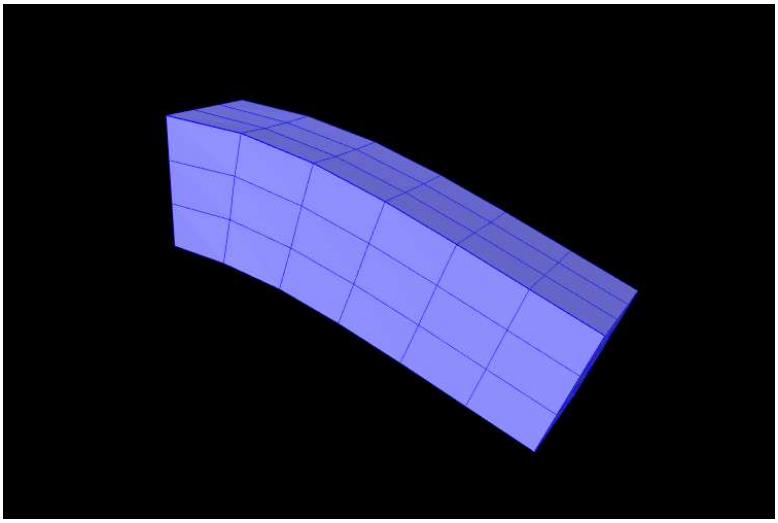


Figure 6.4: FemBeam model loaded into ArtiSynth.

6.2.5 Example: a simple beam model

A complete application model that implements a simple FEM beam is given below.

```
package artisynth.demos.tutorial;

import java.awt.Color;
import java.io.IOException;

import maspack.render.RenderProps;
import artisynth.core.femmodels.FemFactory;
import artisynth.core.femmodels.FemModel.SurfaceRender;
import artisynth.core.femmodels.FemModel3d;
import artisynth.core.femmodels.FemNode3d;
import artisynth.core.materials.LinearMaterial;
import artisynth.core.mechmodels.MechModel;
import artisynth.core.workspace.RootModel;

public class FemBeam extends RootModel {

   // Models and dimensions
   FemModel3d fem;
   MechModel mech;
   double length = 1;
   double density = 10;
   double width = 0.3;
   double EPS = 1e-15;

   public void build (String[] args) throws IOException {

      // Create and add MechModel
      mech = new MechModel ("mech");
      addModel(mech);

      // Create and add FemModel
      fem = new FemModel3d ("fem");
      mech.add (fem);

      // Build hex beam using factory method
      FemFactory.createHexGrid (
         fem, length, width, width, /*nx=*/6, /*ny=*/3, /*nz=*/3);

      // Set FEM properties
      fem.setDensity (density);
```

```
41        fem.setParticleDamping (0.1);
42        fem.setMaterial (new LinearMaterial (4000, 0.33));
43
44        // Fix left-hand nodes for boundary condition
45        for (FemNode3d n : fem.getNodes()) {
46           if (n.getPosition().x <= -length/2+EPS) {
47              n.setDynamic (false);
48           }
49        }
50
51        // Set rendering properties
52        setRenderProps (fem);
53
54     }
55
56     // sets the FEM's render properties
57     protected void setRenderProps (FemModel3d fem) {
58        fem.setSurfaceRendering (SurfaceRender.Shaded);
59        RenderProps.setLineColor (fem, Color.BLUE);
60        RenderProps.setFaceColor (fem, new Color (0.5f, 0.5f, 1f));
61     }
62
63  }
```

This example can be found in `artisynth.demos.tutorial.FemBeam`. The `build()` method first creates a `MechModel` and `FemModel3d`. A FEM beam is created using a factory method on line 36. This beam is centered at the origin, so its length extends from `-length/2` to `length/2`. The density, damping and material properties are then assigned.

On lines 45–49, a fixed boundary condition is set to the left-hand side of the beam by setting the corresponding nodes to be non-dynamic. Due to numerical precision, a small `EPSILON` buffer is required to ensure all left-hand boundary nodes are identified (line 46).

Rendering properties are then assigned to the FEM model on line 52. These will be discussed further in Section 6.12.

6.3 FEM Geometry

Associated with each FEM model is a list of geometry with the heading `meshes`. This geometry can be used for either display purposes, or for interactions such as collisions. A geometry itself has no physical properties; its motion is entirely governed by the FEM model that contains it.

All FEM geometries are of type FemMeshComp, which stores a reference to a mesh object (Section 2.5), as well as attachment information that links vertices of the mesh to points within the FEM. The attachments enforce the shape function interpolation in Equation (6.2) to hold at each mesh vertex, with constant shape function coefficients.

6.3.1 Surface meshes

By default, every `FemModel3d` has an auto-generated geometry representing the "surface mesh". The surface mesh consists of all un-shared element faces (i.e. the faces of individual elements that are exposed to the world), and its vertices correspond to the nodes that make up those faces. As the FEM nodes move, so do the mesh vertices due to the attachment framework.

The surface mesh can be obtained using one of the following functions in `FemModel3d`:

```
FemMeshComp getSurfaceMeshComp ();     // returns the FemMeshComp surface component
PolygonalMesh getSurfaceMesh ();  // returns the underlying polygonal surface mesh
```

The first returns the surface complete with attachment information. The latter method directly returns the `PolygonalMesh` that is controlled by the FEM.

It is possible to manually set the surface mesh:

```
setSurfaceMesh ( PolygonalMesh surface );  // manually set surface mesh
```

However, doing so is normally not necessary. It is always possible to add additional mesh geometries to a finite element model, and the visibility settings can be changed so that the default surface mesh is not rendered.

6.3.2 Embedding geometry within an FEM

Any geometry of type MeshBase can be added to a `FemModel3d`. To do so, first position the mesh so that its vertices are in the desired locations inside the FEM, then call one of the `FemModel3d` methods:

```
FemMeshComp addMesh ( MeshBase mesh );                // creates and returns ↩
    FemMeshComp
FemMeshComp addMesh ( String name, MeshBase mesh );
```

The latter is a convenience routine that also gives the newly embedded `FemMeshComp` a name.

6.3.3 Example: a beam with an embedded sphere

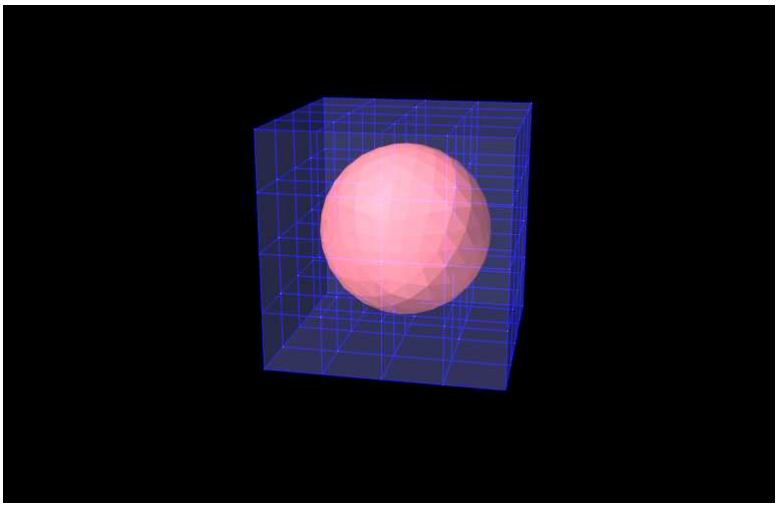


Figure 6.5: FemEmbeddedSphere model loaded into ArtiSynth.

A complete model demonstrating embedding a mesh is given below.

```
1  package artisynth.demos.tutorial;
2
3  import java.awt.Color;
4  import java.io.IOException;
5
6  import maspack.geometry.*;
7  import maspack.render.RenderProps;
8  import artisynth.core.mechmodels.Collidable.Collidability;
9  import artisynth.core.femmodels.*;
10 import artisynth.core.femmodels.FemModel.SurfaceRender;
11 import artisynth.core.materials.LinearMaterial;
12 import artisynth.core.mechmodels.MechModel;
13 import artisynth.core.workspace.RootModel;
14
15 public class FemEmbeddedSphere extends RootModel {
16
17    // Internal components
18    protected MechModel mech;
19    protected FemModel3d fem;
20    protected FemMeshComp sphere;
21
22    @Override
23    public void build(String[] args) throws IOException {
24       super.build(args);
```

```
25
26        mech = new MechModel("mech");
27        addModel(mech);
28
29        fem = new FemModel3d("fem");
30        mech.addModel(fem);
31
32        // Build hex beam and set properties
33        double[] size = {0.4, 0.4, 0.4};
34        int[] res = {4, 4, 4};
35        FemFactory.createHexGrid (fem,
36           size[0], size[1], size[2], res[0], res[1], res[2]);
37        fem.setParticleDamping(2);
38        fem.setDensity(10);
39        fem.setMaterial(new LinearMaterial(4000, 0.33));
40
41        // Add an embedded sphere mesh
42        PolygonalMesh sphereSurface = MeshFactory.createOctahedralSphere(0.15, 3);
43        sphere = fem.addMesh("sphere", sphereSurface);
44        sphere.setCollidable (Collidability.EXTERNAL);
45
46        // Boundary condition: fixed LHS
47        for (FemNode3d node : fem.getNodes()) {
48           if (node.getPosition().x == -0.2) {
49              node.setDynamic(false);
50           }
51        }
52
53        // Set rendering properties
54        setFemRenderProps (fem);
55        setMeshRenderProps (sphere);
56     }
57
58     // FEM render properties
59     protected void setFemRenderProps ( FemModel3d fem ) {
60        fem.setSurfaceRendering (SurfaceRender.Shaded);
61        RenderProps.setLineColor (fem, Color.BLUE);
62        RenderProps.setFaceColor (fem, new Color (0.5f, 0.5f, 1f));
63        RenderProps.setAlpha(fem, 0.2);    // transparent
64     }
65
66     // FemMeshComp render properties
67     protected void setMeshRenderProps ( FemMeshComp mesh ) {
68        mesh.setSurfaceRendering( SurfaceRender.Shaded );
69        RenderProps.setFaceColor (mesh, new Color (1f, 0.5f, 0.5f));
70        RenderProps.setAlpha (mesh, 1.0);    // opaque
71     }
72
73  }
```

This example can be found in `artisynth.demos.tutorial.FemEmbeddedSphere`. The model is very similar to `FemBeam`. A `MechModel` and `FemModel3d` are created and added. At line 41, a `PolygonalMesh` of a sphere is created using a factory method. The sphere is already centered inside the beam, so it does not need to be repositioned. At Line 42, the sphere is embedded inside model `fem`, creating a `FemMeshComp` with the name "sphere". The full model is shown in Figure 6.5.

6.4 Connecting FEM models to other components

To couple FEM models to other dynamic components, the "attachment" mechanism described in Section 1.2 is used. This involves creating and adding to the model attachment components, which are instances of DynamicAttachment, as described in Section 3.6. Common point-based attachment classes are listed in Table 6.5.

Table 6.5: Point-based attachments

Attachment	Description
PointParticleAttachment	Attaches one "point" to one "particle"
PointFrameAttachment	Attaches one "point" to one "frame"
PointFem3dAttachment	Attaches one "point" to a linear combination of FEM nodes

FEM models are connected to other model components by attaching their nodes to various components. This can be done by creating an attachment object of the appropriate type, and then adding it to the `MechModel` using

```
addAttachment (DynamicAttachment attach); // adds an attachment constraint
```

There are also convenience routines inside `MechModel` that will create the appropriate attachments automatically (see Section 3.6.1).

All attachments described in this section are based around FEM *nodes*. However, it is also possible to attach frame-based components (such as rigid bodies) directly to an FEM, as described in Section 6.6.

6.4.1 Connecting nodes to rigid bodies or particles

Since FemNode3d is a subclass of Particle, the same methods described in Section 3.6.1 for attaching particles to other particles and frames are available. For example, we can attach an FEM node to a rigid body using a either a statement of the form

```
mech.addAttachment (new PointFrameAttachment(body, node));
```

or the following equivalent statement which does the same thing:

```
mech.attachPoint (node, body);
```

Both of these create a `PointFrameAttachment` between a rigid body (called `body`) and an FEM node (called `node`) and then adds it to the `MechModel mech`.

One can also attach the nodes of one FEM model to the nodes of another using statements like

```
mech.addAttachment (new PointParticle (node1, node2));
```

or

```
mech.attachPoint (node2, node1);
```

which attaches `node2` to `node1`.

6.4.2 Example: connecting a beam to a block

The following model demonstrates attaching an FEM beam to a rigid block.

```
1  package artisynth.demos.tutorial;
2
3  import java.io.IOException;
4
5  import maspack.matrix.RigidTransform3d;
6  import artisynth.core.femmodels.FemNode3d;
7  import artisynth.core.mechmodels.PointFrameAttachment;
8  import artisynth.core.mechmodels.RigidBody;
9
10 public class FemBeamWithBlock extends FemBeam {
11
12    public void build (String[] args) throws IOException {
13
14       // Build simple FemBeam
15       super.build (args);
```

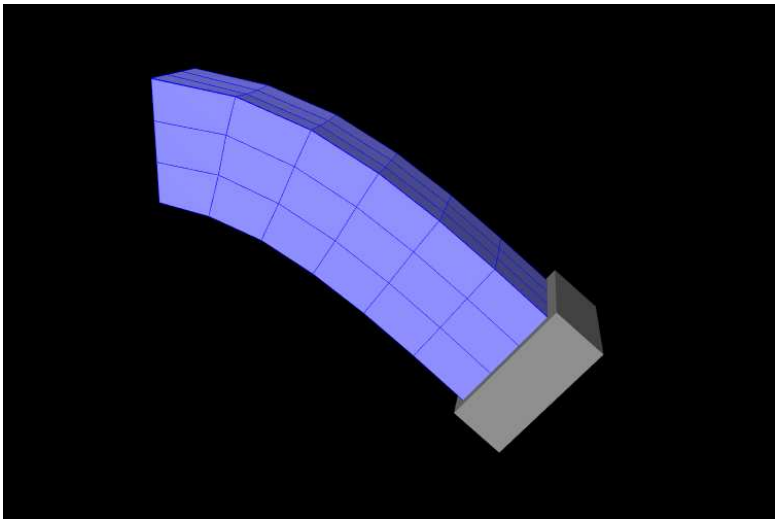


Figure 6.6: FemBeamWithBlock model loaded into artisynth.

```
16
17        // Create a rigid block and move to the side of FEM
18        RigidBody block = RigidBody.createBox (
19           "block", width/2, 1.2*width, 1.2*width, 2*density);
20        mech.addRigidBody (block);
21        block.setPose (new RigidTransform3d (length/2+width/4, 0, 0));
22
23        // Attach right-side nodes to rigid block
24        for (FemNode3d node : fem.getNodes()) {
25           if (node.getPosition().x >= length/2-EPS) {
26              mech.addAttachment (new PointFrameAttachment (block, node));
27           }
28        }
29     }
30
31  }
```

This model extends the `FemBeam` example of Section 6.2.5. The `build()` method then creates and adds a `RigidBody` block (lines 18–20). On line 21, the block is repositioned to the side of the beam to prepare for the attachment. On lines 24–28, all right-most nodes of the beam are then set to be attached to the block using a `PointFrameAttachment`. In this case, the attachments are explicitly created. They could also have been attached using

```
mech.attachPoint (node, block);  // attach node to rigid block
```

6.4.3 Connecting nodes directly to elements

Typically, nodes do not align in a way that makes it possible to connect them to other FEM models and/or points based on simple point-to-node attachments. Instead, we use a different mechanism that allows us to attach a point to an arbitrary location within an FEM model. This is done using an attachment component of type PointFem3dAttachment, which implements an attachment where the position $\mathbf{p}$ and velocity $\mathbf{u}$ of the attached point is determined by a weighted sum of the positions $\mathbf{x}_k$ and velocities $\mathbf{u}_k$ of selected fem nodes:

$$\mathbf{p} = \sum \alpha_k \mathbf{x}_k \tag{6.4}$$

Any force $\mathbf{f}$ acting on the attached point is then propagated back to the nodes, according to the relation

$$\mathbf{f}_k = \alpha_k \mathbf{f} \tag{6.5}$$

where $\mathbf{f}_k$ is the force acting on node k due to $\mathbf{f}$. This relation can be derived based on the conservation of energy. If $\mathbf{p}$ is embedded within a single element, then the $\mathbf{x}_k$ are simply the element nodes and the α_i are corresponding shape function values; this is known as an *element-based* attachment. On the other hand, as described below, it is sometimes

desirable to form an attachment using a more general set of nodes that extends beyond a single element; this is known as a *nodal-based* attachment (Section 6.4.7).

An element-based attachment can be created using a code fragment of the form

```
PointFem3dAttachment ax = new PointFem3dAttachment(pnt);
ax.setFromElement (pnt.getPosition(), elem);
mech.addAttachment (ax);
```

First, a `PointFem3dAttachment` is created for the point `pnt`. Next, `setFromElement()` is used to determine the nodal weights within the element `elem` for the specified position (which in this case is simply the point's current position). To do this, it computes the "natural coordinates" coordinates of the position within the element. For this to be guaranteed to work, the position should be on or inside the element. If natural coordinates cannot be found, the method will return `false` and the nearest estimates coordinates will be used instead. However, it is sometimes possible to find natural coordinates outside a given element as long as the shape functions are well-defined. Finally, the attachment is added to the model.

More conveniently, the exact same functionality is provided by the `attachPoint()` method in `MechModel`:

```
mech.attachPoint (pnt, elem);
```

This creates an attachment identical to that created by the previous code fragment.

Often, one does not want to have to determine the element to which a point should be attached. In that case, one can call

```
PointFem3dAttachment ax = new PointFem3dAttachment(pnt);
ax.setFromFem (pnt.getPosition(), fem);
mech.addAttachment (ax);
```

or, equivalently,

```
mech.attachPoint (pnt, fem);
```

This will find the nearest element to the node in question and use that to create the attachment. If the node is outside the FEM model, then it will be attached to the nearest point on the FEM's surface.

6.4.4 Example: connecting two FEMs together

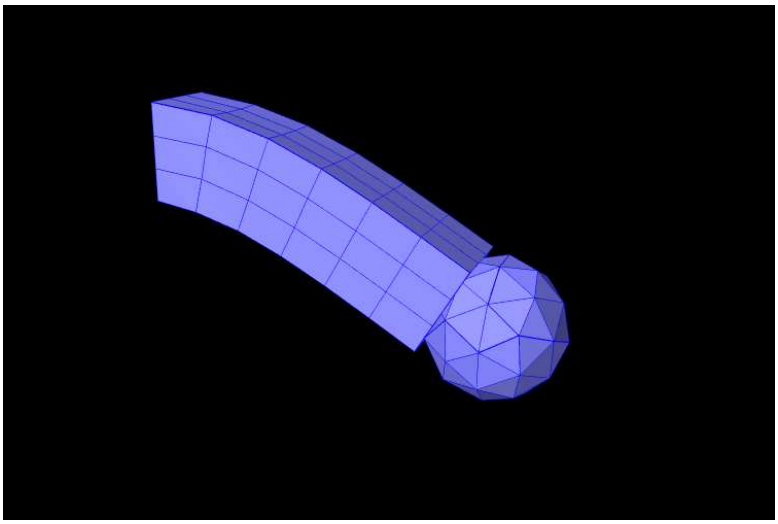


Figure 6.7: FemBeamWithFemSphere model loaded into ArtiSynth.

The following model demonstrates how to attach two FEM models together:

```
1 package artisynth.demos.tutorial;
2
3 import java.io.IOException;
```

```
4
5  import maspack.matrix.RigidTransform3d;
6  import artisynth.core.femmodels.*;
7  import artisynth.core.materials.LinearMaterial;
8  import maspack.util.PathFinder;
9
10 public class FemBeamWithFemSphere extends FemBeam {
11
12    public void build (String[] args) throws IOException {
13
14       // Build simple FemBeam
15       super.build (args);
16
17       // Create a FEM sphere
18       FemModel3d femSphere = new FemModel3d("sphere");
19       mech.addModel(femSphere);
20       // Read from TetGen file
21       TetGenReader.read(femSphere,
22          PathFinder.getSourceRelativePath(FemModel3d.class, "meshes/sphere2.1.node"),
23          PathFinder.getSourceRelativePath(FemModel3d.class, "meshes/sphere2.1.ele"));
24       femSphere.scaleDistance(0.22);
25       // FEM properties
26       femSphere.setDensity(10);
27       femSphere.setParticleDamping(2);
28       femSphere.setMaterial(new LinearMaterial(4000, 0.33));
29
30       // Reposition FEM to side of beam
31       femSphere.transformGeometry( new RigidTransform3d(length/2+width/2, 0, 0) );
32
33       // Attach sphere nodes that are inside beam
34       for (FemNode3d node : femSphere.getNodes()) {
35          // Find element containing node (if exists)
36          FemElement3d elem = fem.findContainingElement(node.getPosition());
37          // Add attachment if node is inside "fem"
38          if (elem != null) {
39             mech.attachPoint(node, elem);
40          }
41       }
42
43       // Set render properties
44       setRenderProps(femSphere);
45
46    }
47
48 }
```

This example can be found in `artisynth.demos.tutorial.FemBeamWithFemSphere`. The model extends `FemBeam`, adding a finite element sphere and coupling them together. The sphere is created and added on lines 18–28. It is read from TetGen-generated files using the TetGenReader class. The model is then scaled to match the dimensions of the current model, and transformed to the right side of the beam. To create attachments, the code first checks for any nodes that belong to the sphere that fall inside the beam using the FemModel3d.findContainingElement(Point3d) method (line 36), which returns the containing element if the point is inside the model, or `null` if the point is outside. Internally, this spatial search uses a bounding volume hierarchy for efficiency (see BVTree and BVFeatureQuery). If the point is contained within the beam, then `mech.attachPoint()` is used to attach it to the nodes of the element (line 39).

6.4.5 Finding which nodes to attach

While it is straightforward to connect nodes to rigid bodies or other FEM nodes or elements, it is often necessary to determine *which* nodes to attach. This was evident in the example of Section 6.4.4, which attached nodes of an FEM sphere that were found to be *inside* an FEM beam.

As in that example, finding the nodes to attach can often be done using geometric queries. For example, we often select nodes based on how close they are to the other body we wish to attach to.

Various prolixity queries are available for this task. To find the distance of a point to a polygonal mesh, we can use the following PolygonalMesh methods,

```
double distanceToPoint (Point3d pnt)

int isInside (Point3d pnt) // for closed meshes only
```

where the latter method returns 1 if the point is inside and 0 otherwise. For checking the distance of an FEM node, `pnt` can be obtained from `node.getPosition()` (or possibly `node.getRestPosition()`). For example, to find all nodes within a distance `tol` of the surface of a rigid body, we could use the code fragment:

```
RigidBody body;
FemModel3d fem;
...
double tol = 0.001;
PolygonalMesh surface = body.getSurfaceMesh();
ArrayList<FemNode3d> nearNodes = new ArrayList<FemNode3d>();
for (FemNode3d n : fem.getNodes()) {
   if (surface.distanceToPoint (n.getPosition()) < tol) {
      nearNodes.add (n);
   }
}
```

If we want to check only nodes that lie on the FEM surface, then we can filter them using the `FemModel3d` method isSurfaceNode():

```
for (FemNode3d n : fem.getNodes()) {
   if (fem.isSurfaceNode (n) &&
       surface.distanceToPoint (n.getPosition()) < tol) {
      nearNodes.add (n);
   }
}
```

> Most of the mesh-based query methods work only for *triangular* meshes. The `PolygonalMesh` method isTriangular() can be used to determine if the mesh is triangular. If it is not, it can made triangular by calling triangulate(), although in general this should be done during model construction *before* the mesh-based component has been added to the model.

For connecting an FEM model to another FEM model, `FemModel3d` provides a number of query methods:

```
/* nearest element queries */

// find the nearest element (shell or volumetric) to pnt:
FemElement3dBase findNearestElement (Point3d near, Point3d pnt)
FemElement3dBase findNearestElement (
   Point3d near, Point3d pnt, ElementFilter filter)

// find the nearest surface element (shell or volumetric) to pnt:
FemElement3dBase findNearestSurfaceElement(Point3d near, Point3d pnt)

// find the nearest volumetric element to pnt:
FemElement3d findNearestVolumetricElement (Point3d near, Point3d pnt)

// find the nearest shell element to pnt:
ShellElement3d findNearestShellElement (Point3d near, Point3d pnt)

// find the volumetric element (if any) containing pnt:
FemElement3d findContainingElement(Point3d pnt)

/* nearest node queries */

// find the nearest node to pnt that is within maxDist:
FemNode3d findNearestNode (Point3d pnt, double maxDist)
```

```
// find all nodes that are within maxDist of pnt:
ArrayList<FemNode3d> findNearestNodes (Point3d pnt, double maxDist)
```

All the above queries are based on the FEM model's *current* nodal positions. The method findNearestElement(near,pnt,filter) allows the application to specify a FemModel.ElementFilter to restrict the elements that are searched.

The argument `near` that appears in some of the queries is an optional argument which, if not `null`, returns the location of the corresponding nearest point on the element. The distance from `pnt` to the element can then be found using

```
near.distance (pnt);
```

If the resulting distance is 0, then the point is on or inside the element. Otherwise, the point is outside the element, and if no element filters were used in the query, outside the FEM model itself.

> Typically, it is preferred attach a point to an element only if it lies on or inside an element. However, it is possible to attach points outside an element as long as the system is able to determine appropriate element "coordinates" for that point (which it may not be able to do if the point is far away). In addition, the motion behavior of an exterior attached point may sometimes appear counterintuitive.

The `FemModel3d` element and node queries can be used in a variety of ways.

findNearestNodes() can be used to find all nodes within a certain distance of a point, as part of the process of making nodal-based attachments (Section 6.4.7).

findNearestNode() is used in the FemMuscleBeam example (Section 6.9.4) to determine if a desired muscle point is near enough to a node to use that node directly, or if a marker should be created.

As another example, suppose we wish to connect the surface nodes of an FEM model `femA` to the surface elements of another model `femB` if they lie within a prescribed distance `tol` of the surface of `femB`. Then we could use the following code:

```
MechModel mech;
FemModel3d femA;
FemModel3d femB;
...
double tol = 0.001;
Point3d near = new Point3d();
for (FemNode3d n : femA.getNodes()) {
   if (femA.isSurfaceNode(n)) {
      FemElement3dBase elem =
         femB.findNearestSurfaceElement (near, n.getPosition());
      if (elem != null && near.distance(n.getPosition()) <= tol) {
         // attach if within distance
         mech.attachPoint (n, elem);
      }
   }
}
```

6.4.6 Example: two bodies connected by an FEM "spring"

The `LumbarFrameSpring` example in Section 3.5.4 uses a frame spring to connect two simplified lumbar vertebrae. However, it is also possible to use an FEM model in place of a frame spring, possibly providing a more realistic model of the intervertebral disc. A simple model which does this is defined in

```
artisynth.demos.tutorial.LumbarFEMDisk
```

The initial source code is similar to that for `LumbarFrameSpring`, but differs in the section where the FEM disk replaces the `FrameSpring`:

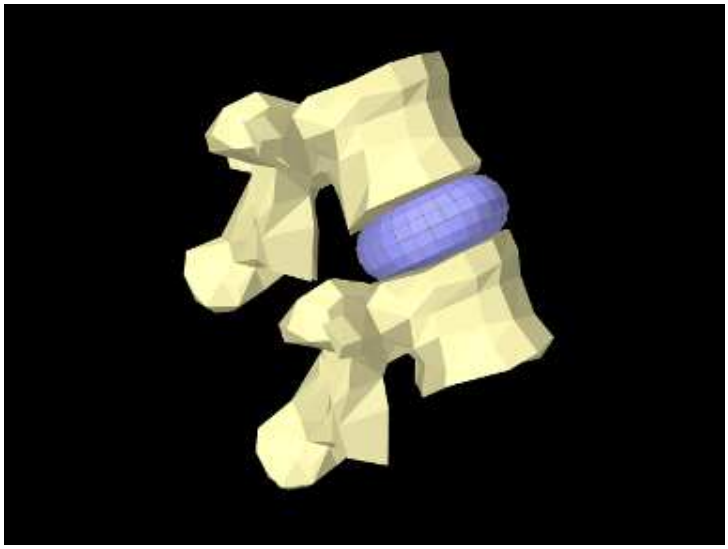


Figure 6.8: `LumbarFEMDisk` loaded into ArtiSynth, showing a simplified FEM model connecting two vertebrae.

```
56        // create a torus shaped FEM model for the disk
57        FemModel3d fem = new FemModel3d();
58        fem.setDensity (1500);
59        fem.setMaterial (new LinearMaterial (20000, 0.4));
60        FemFactory.createHexTorus (fem, 0.011, 0.003, 0.008, 16, 30, 2);
61        mech.addModel (fem);
62
63        // position it betweem the disks
64        double DTOR = Math.PI/180.0;
65        fem.transformGeometry (
66           new RigidTransform3d (-0.012, 0.0, 0.040, 0, -DTOR*25, DTOR*90));
67
68        // find and attach nearest nodes to either the top or bottom mesh
69        double tol = 0.001;
70        for (FemNode3d n : fem.getNodes()) {
71           // top vertebra
72           double d = lumbar1.getSurfaceMesh().distanceToPoint(n.getPosition());
73           if (d < tol) {
74              mech.attachPoint (n, lumbar1);
75           }
76           // bottom vertebra
77           d = lumbar2.getSurfaceMesh().distanceToPoint(n.getPosition());
78           if (d < tol) {
79              mech.attachPoint (n, lumbar2);
80           }
81        }
82        // set render properties ...
83        fem.setSurfaceRendering (SurfaceRender.Shaded);
84        RenderProps.setFaceColor (fem, new Color (153/255f, 153/255f, 1f));
85        RenderProps.setFaceColor (mech, new Color (238, 232, 170)); // bone color
86     }
87  }
```

The simplified FEM model representing the "disk" is created at lines 57-61, using a torus-shaped model created by `FemFactory`. It is then repositioning using `transformGeometry()` (Section 4.8) to place it between the vertebrae (line 64-66). After the FEM model is positioned, we find which nodes are within a distance `tol` of each vertebral surface and attach them to the appropriate body (lines 69-81).

To run this example in ArtiSynth, select All demos > tutorial > LumbarFEMDisk from the Models menu. The model should load and initially appear as in Figure 6.8. The behavior is best seem by running the model and using the pull controller to exert forces on the upper vertebra.

6.4.7 Nodal-based attachments

The example of Section 6.4.4 uses element-based attachments to connect the nodes of one FEM to elements of another. As mentioned above, element-based attachments assume that the attached point is associated with a specific FEM model element. While this often gives good results, there are situations where it may be desirable to distribute the connection more broadly among a larger set of nodes.

In particular, this is sometimes the case when connecting FEM models to point-to-point springs. The end-point of such a spring may end up exerting a large force on the FEM, and then if the number of nodes to which the end-point is attached are too small, the resulting forces on these nodes (Equation 6.5) may end up being too large. In other words, it may be desirable to distribute the spring's force more evenly throughout the FEM model.

To handle such situations, it is possible to create a *nodal-based* attachment in which the nodes and weights are explicitly specified. This involves explicitly creating a `PointFem3dAttachment` for the point or particle to be attached and the specifying the nodes and weights directly,

```
PointFem3dAttachment ax = new PointFem3dAttachment (part);
ax.setFromNodes (nodes, weights);
mech.addAttachment (ax);
```

where `nodes` and `weights` are arrays of `FemNode` and `double`, respectively. It is up to the application to determine these.

PointFem3dAttachment provides several methods for explicitly specifying nodes and weights. The signatures for these include:

```
void setFromNodes (FemNode[] nodes, double[] weights)
void setFromNodes (Collection<FemNode> nodes, VectorNd weights)
boolean setFromNodes (Point3d pos, FemNode[] nodes)
boolean setFromNodes (Point3d pos, Collection<FemNode> nodes)
```

The last two methods determine the weights automatically, using an inverse-distance-based calculation in which each weight α_k is initially computed as

$$\alpha_k = \frac{d_{\max}}{d_k + d_{max}} \tag{6.6}$$

where d_k is the distance from node k to `pos` and $d_{\max}$ is the maximum distance. The weights are then adjusted to ensure that they sum to one and that the weighted sum of the nodes equals `pos`. In some cases, the specified nodes may not provide enough support for the last condition to be met, in which case the methods return `false`.

6.4.8 Example: element vs. nodal-based attachments

The model demonstrating the difference between element and nodal-based attachments is defined in

```
artisynth.demos.tutorial.PointFemAttachment
```

It creates two FEM models, each with a single point-to-point spring attached to a particle at their center. The model at the top (`fem1` in the code below) is connected to the particle using an element-based attachment, while the lower model (`fem2` in the code) is connected using a nodal-based attachment with a larger number of nodes. Figure 6.9 shows the result after the model is run until stable. The element-based attachment results in significantly higher deformation in the immediate vicinity around the attachment, while for the nodal-based attachment, the deformation is distributed much more evenly through the model.

The build method and some of the auxiliary code for this model is shown below. Code for the other auxiliary methods, including `addFem()`, `addParticle()`, `addSpring()`, and `setAttachedNodesWhite()`, can be found in the actual source file.

```
1   // Filter to select only elements for which the nodes are entirely on the
2   // positive side of the x-z plane.
3   private class MyFilter extends ElementFilter {
4      public boolean elementIsValid (FemElement e) {
5         for (FemNode n : e.getNodes()) {
6            if (n.getPosition().y < 0) {
```

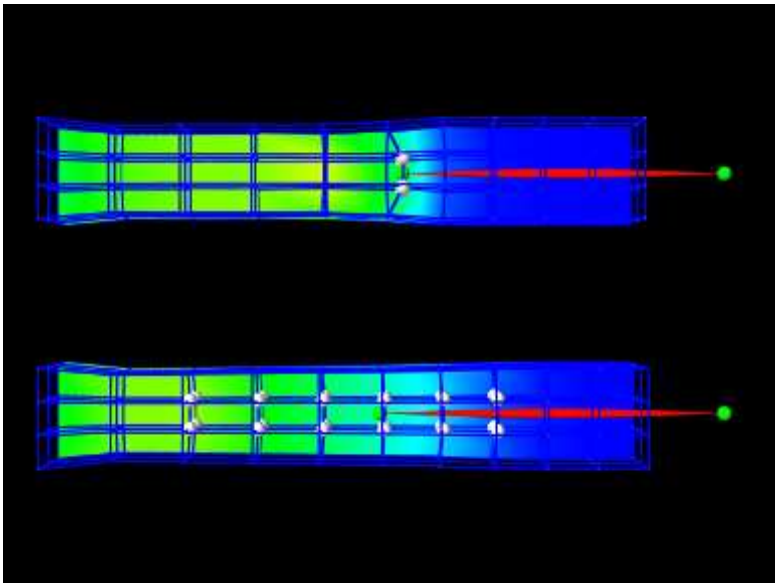


Figure 6.9: PointFemAttachment loaded into ArtiSynth and run until stable. The top and bottom models are connected to their springs using element and nodal-based attachments, respectively. The nodes associated with each attachment are rendered as white spheres.

```
                return false;
             }
          }
          return true;
       }
    }

    // Collect and return all the nodes of an FEM model associated with a
    // set of elements specified by an array of element numbers
    private HashSet<FemNode3d> collectNodes (FemModel3d fem, int[] elemNums) {
       HashSet<FemNode3d> nodes = new HashSet<FemNode3d>();
       for (int i=0; i<elemNums.length; i++) {
          FemElement3d e = fem.getElements().getByNumber (elemNums[i]);
          for (FemNode3d n : e.getNodes()) {
             nodes.add (n);
          }
       }
       return nodes;
    }

    public void build (String[] args) {
       MechModel mech = new MechModel ("mech");
       addModel (mech);
       mech.setGravity (0, 0, 0); // turn off gravity

       // create and add two FEM beam models centered at the specified locations
       FemModel3d fem1 = addFem (mech,  0.0, 0.0,  0.25);
       FemModel3d fem2 = addFem (mech,  0.0, 0.0, -0.25);

       // reconstruct the FEM surface meshes so that they show only elements on
       // the positive side of the x-y plane. Also, set surface rendering to
       // show strain values.
       fem1.createSurfaceMesh (new MyFilter());
       fem1.setSurfaceRendering (SurfaceRender.Strain);
       fem2.createSurfaceMesh (new MyFilter());
       fem2.setSurfaceRendering (SurfaceRender.Strain);

       // create and add the particles for the point-to-point springs
       // that will apply forces to each FEM.
```

```
46        Particle p1 = addParticle (mech,  0.9, 0.0,  0.25);
47        Particle p2 = addParticle (mech,  0.9, 0.0, -0.25);
48        Particle m1 = addParticle (mech,  0.0, 0.0,  0.25);
49        Particle m2 = addParticle (mech,  0.0, 0.0, -0.25);
50
51        // attach spring end-point to fem1 using an element-based marker
52        mech.attachPoint (m1, fem1);
53
54        // attach spring end-point to fem2 using a larger number of nodes, formed
55        // from the node set for elements 22, 31, 40, 49, and 58. This is done by
56        // explicitly creating the attachment and then setting it to use the
57        // specified nodes
58        HashSet<FemNode3d> nodes =
59           collectNodes (fem2, new int[] { 22, 31, 40, 49, 58 });
60
61        PointFem3dAttachment ax = new PointFem3dAttachment (m2);
62        ax.setFromNodes (m2.getPosition(), nodes);
63        mech.addAttachment (ax);
64
65        // finally, create the springs
66        addSpring (mech, /*stiffness=*/10000, p1, m1);
67        addSpring (mech, /*stiffness=*/10000, p2, m2);
68
69        // set the attachments nodes for m1 and m2 to render as white spheres
70        setAttachedNodesWhite (m1);
71        setAttachedNodesWhite (m2);
72        // set render properties for m1
73        RenderProps.setSphericalPoints (m1, 0.015, Color.GREEN);
74     }
```

The `build()` method begins by creating a `MechModel` and then adding to it two FEM beams (created using the auxiliary method `addFem()`. Rendering of each FEM model's surface is then set up to show strain values (`setSurfaceRendering()`, lines 41 and 43). The surface meshes themselves are also redefined to exclude the frontmost elements, allowing the strain values to be displayed closer model centers. This redefinition is done using calls to `createSurfaceMesh()` (lines 40, 41) with a custom `ElementFilter` defined at lines 3-12.

Next, the end-point particles for the axial springs are created (using the auxiliary method `addParticle()`, lines 46-49), and particle `m1` is attached to `fem1` using `mech.attachPoint()` (line 52), which creates an element-based attachment at the point's current location. Point `m2` is then attached to `fem2` using a nodal-based attachment. The nodes for these are collected as the union of all nodes for a specified set of elements (lines 58-59, and the method `collectNodes()` defined at lines 16-25). These are then used to create a nodal-based attachment (lines 61-63), where the weights are determined automatically using the method associated with equation (6.6).

Finally, the springs are created (auxiliary method `addSpring()`, lines 66-67), the nodes associated for each attachment are set to render as white spheres (`setAttachedNodesWhites()`, lines 70-71), and the particles are set to render as green spheres.

To run this example in ArtiSynth, select All demos > tutorial > PointFemAttachment from the Models menu. Running the model will cause it to settle into the state shown in Figure 6.9. Selecting and dragging one of the spring anchor points at the right will cause the spring tension to vary and further illustrate the difference between the element and nodal-based attachments.

6.5 FEM markers

Just as there are `FrameMarkers` to act as anchor points on a frame or rigid body (Section 3.2.1), there are also `FemMarkers` that can mark a point inside a finite element. They are frequently used to provide anchor points for attaching springs and forces to a point inside an element, but can also be used for graphical purposes.

FEM markers are implemented by the class FemMarker, which is a subclass of Point. They are essentially massless points that contain their own attachment component, so when creating and adding a marker there is no need to create a separate attachment component.

Within the component hierarchy, FEM markers are typically stored in the `markers` list of their associated FEM model. They can be created and added using a code fragment of the form

```
FemMarker mkr = new FemMarker (1, 0, 0);
mkr.setFromFem (fem); // attach to the nearest fem element
fem.addMarker (mkr);  // add to fem
```

This creates a marker at the location $(1,0,0)$ (in world coordinates), calls `setFromFem()` to attach it to the nearest element in the FEM model (which is either the containing element or the nearest element on the model's surface), and then adds it to the `markers` list.

If the marker's attachment has not already been set when `addMarker()` is called, then `addMarker()` will call `setFromFem()` automatically. Therefore the above code fragment is equivalent to the following:

```
FemMarker mkr = new FemMarker (1, 0, 0);
fem.addMarker (mkr);
```

Alternatively, one may want to explicitly specify the nodes associated with the attachment, as described in Section 6.4.7:

```
FemMarker mkr = new FemMarker (1, 0, 0);
mkr.setFromNodes (nodes, weights);
fem.addMarker (mkr);
```

There are a variety of methods available to set the attachment, mirroring those available in the underlying base class `PointFem3dAttachment`:

```
void setFromFem (FemModel3d fem)
boolean setFromElement (FemElement3d elem)
void setFromNodes (FemNode[] nodes, double[] weights)
void setFromNodes (Collection<FemNode> nodes, VectorNd weights)
boolean setFromNodes (FemNode[] nodes)
boolean setFromNodes (Collection<FemNode> nodes)
```

The last two methods compute nodal weights automatically, as described in Section 6.4.7, based on the marker's currently assigned position. If the supplied nodes do not provide sufficient support, then the methods return `false`.

Another set of convenience methods are supplied by `FemModel3d`, which combine these with the `addMarker()` call:

```
void addMarker (FemMarker mkr, FemElement3d elem)
void addMarker (FemMarker mkr, FemNode[] nodes, double[] weights)
void addMarker (FemMarker mkr, Collection<FemNode> nodes, VectorNd weights)
boolean addMarker (FemMarker mkr, FemNode[] nodes)
boolean addMarker (FemMarker mkr, Collection<FemNode> nodes)
```

For example, one can do

```
FemMarker mkr = new FemMarker (1, 0, 0);
fem.addMarker (mkr, nodes, weights);
```

Markers are often used to track movement within an FEM model. For that, one can examine their positions and velocities, as with any other particles, using the methods

```
Point3d getPosition();     // returns the current position
Vector3d getVelocity();     // returns the current velocity
```

The return values from these methods should not be modified. Alternatively, when a 3D force $\mathbf{f}$ is applied to the marker, it is distributed to the attached nodes according to the nodal weights, as described in Equation (6.5).

6.5.1 Example: attaching an FEM beam to a muscle

A complete application model that employs a fem marker as an anchor for a spring is given below.

```
1 package artisynth.demos.tutorial;
2
3 import java.awt.Color;
4 import java.io.IOException;
```

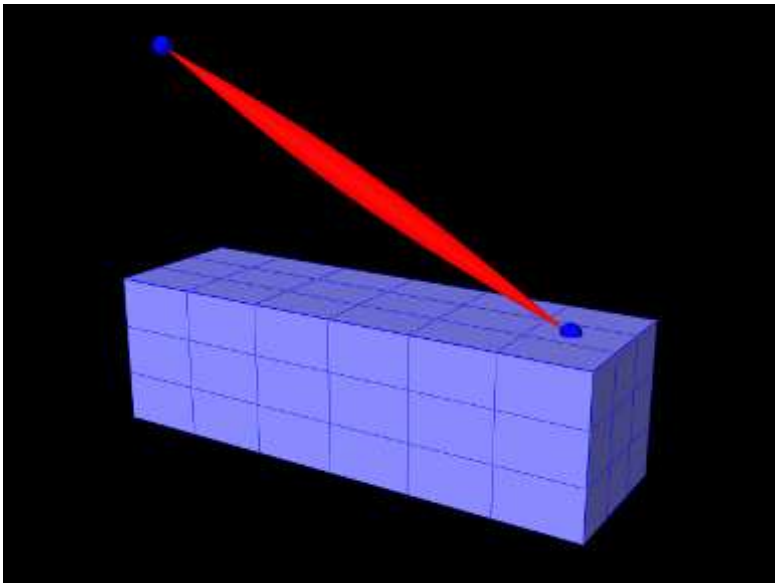


Figure 6.10: FemBeamWithMuscle model loaded into ArtiSynth.

```
5
6  import maspack.render.RenderProps;
7  import maspack.render.Renderer;
8  import artisynth.core.femmodels.FemMarker;
9  import artisynth.core.femmodels.FemModel3d;
10 import artisynth.core.materials.SimpleAxialMuscle;
11 import artisynth.core.mechmodels.Muscle;
12 import artisynth.core.mechmodels.Particle;
13 import artisynth.core.mechmodels.Point;
14
15 public class FemBeamWithMuscle extends FemBeam {
16
17    // Creates a point-to-point muscle
18    protected Muscle createMuscle () {
19       Muscle mus = new Muscle (/*name=*/null, /*restLength=*/0);
20       mus.setMaterial (
21          new SimpleAxialMuscle (/*stiffness=*/20, /*damping=*/10, /*maxf=*/10));
22       RenderProps.setLineStyle (mus, Renderer.LineStyle.SPINDLE);
23       RenderProps.setLineColor (mus, Color.RED);
24       RenderProps.setLineRadius (mus, 0.03);
25       return mus;
26    }
27
28    // Creates a FEM Marker
29    protected FemMarker createMarker (
30       FemModel3d fem, double x, double y, double z) {
31       FemMarker mkr = new FemMarker (/*name=*/null, x, y, z);
32       RenderProps.setSphericalPoints (mkr, 0.02, Color.BLUE);
33       fem.addMarker (mkr);
34       return mkr;
35    }
36
37    public void build (String[] args) throws IOException {
38
39       // Create simple FEM beam
40       super.build (args);
41
42       // Add a particle fixed in space
43       Particle p1 = new Particle (/*mass=*/0, -length/2, 0, 2*width);
44       mech.addParticle (p1);
45       p1.setDynamic (false);
```

```
46        RenderProps.setSphericalPoints (p1, 0.02, Color.BLUE);
47
48        // Add a marker at the end of the model
49        FemMarker mkr = createMarker (fem, length/2-0.1, 0, width/2);
50
51        // Create a muscle between the point an marker
52        Muscle muscle = createMuscle();
53        muscle.setPoints (p1, mkr);
54        mech.addAxialSpring (muscle);
55     }
56
57  }
```

This example can be found in `artisynth.demos.tutorial.FemBeamWithMuscle`. This model extends the `FemBeam` example, adding a `FemMarker` for the spring to attach to. The method `createMarker(...)` on lines 29–35 is used to create and add a marker to the FEM. Since the element is initially set to `null`, when it is added to the FEM, the model searches for the containing or nearest element. The loaded model is shown in Figure 6.10.

6.6 Frame attachments

It is also possible to attach frame components, including rigid bodies, directly to FEM models, using the attachment component FrameFem3dAttachment. Analogously to PointFem3dAttachment, the attachment is implemented by connecting the frame to a set of FEM nodes, and attachments can be either element-based or nodal-based. The frame's origin is computed in the same way as for point attachments, using a weighted sum of node positions (Equation 6.4), while the orientation is computed using a polar decomposition on a deformation gradient determined from either element shape functions (for element-based attachments) or a Procrustes type analysis using nodal rest positions (for nodal-based attachments).

An element-based attachment can be created using either a code fragment of the form

```
FrameFem3dAttachment ax = new FrameFem3dAttachment(frame);
ax.setFromElement (frame.getPose(), elem);
mech.addAttachment (ax);
```

or, equivalently, the `attachFrame()` method in `MechModel`:

```
mech.attachFrame (frame, elem);
```

This attaches the frame `frame` to the nodes of the FEM element `elem`. As with `PointFem3dAttachment`, if the frame's origin is not inside the element, it may not be possible to accurately compute the internal nodal weights, in which case `setFromElement()` will return `false`.

In order to have the appropriate element located automatically, one can instead use

```
FrameFem3dAttachment ax = new FrameFem3dAttachment(frame);
ax.setFromFem (frame.getPose(), fem);
mech.addAttachment (ax);
```

or, equivalently,

```
mech.attachFrame (frame, fem);
```

As with point-to-FEM attachments, it may be desirable to create a nodal-based attachment in which the nodes and weights are not tied to a specific element. The reasons for this are generally the same as with nodal-based point attachments (Section 6.4.7): the need to distribute the forces and moments acting on the frame across a broader set of element nodes. Also, element-based frame attachments use element shape functions to determine the frame's orientation, which may produce slightly asymmetric results if the frame's origin is located particularly close to a specific node.

FrameFem3dAttachment provides several methods for explicitly specifying nodes and weights. The signatures for these include:

```
void setFromNodes (RigidTransform3d TFW, FemNode[] nodes, double[] weights)
void setFromNodes (RigidTransform3d TFW, Collection<FemNode> nodes,
                   VectorNd weights)
boolean setFromNodes (RigidTransform3d TFW, FemNode[] nodes)
boolean setFromNodes (RigidTransform3d TFW, Collection<FemNode> nodes)
```

Unlike their counterparts in `PointFem3dAttachment`, the first two methods also require the current desired pose of the frame `TFW` (in world coordinates). This is because while nodes and weights will unambiguously specify the frame's origin, they do not specify the desired orientation.

6.6.1 Example: attaching frames to an FEM beam

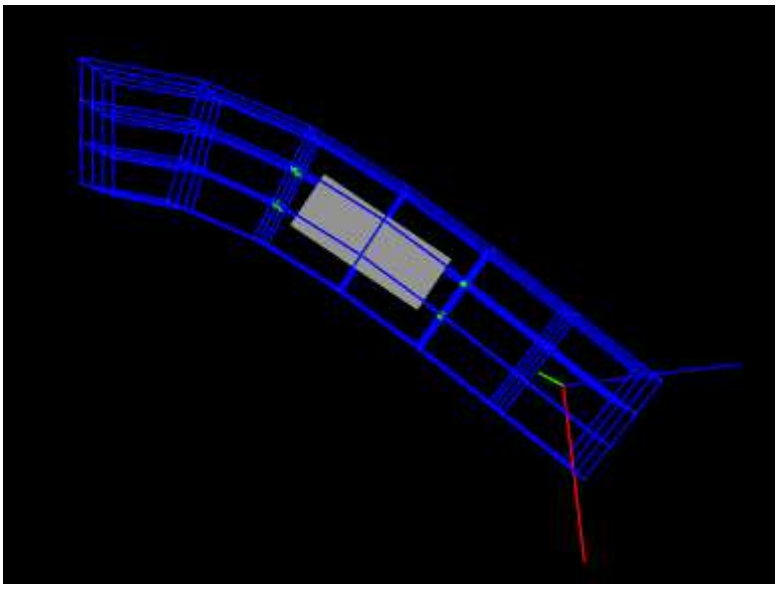


Figure 6.11: FrameFemAttachment loaded into ArtiSynth and run until stable.

A model illustrating how to connect frames to an FEM model is defined in

`artisynth.demos.tutorial.FrameFemAttachment`

It creates an FEM beam, along with a rigid body block and a massless coordinate frame, that are then attached to the beam using nodal and element-based attachments. The build method is shown below:

```
1   public void build (String[] args) {
2
3      MechModel mech = new MechModel ("mech");
4      addModel (mech);
5
6      // create and add FEM beam
7      FemModel3d fem = FemFactory.createHexGrid (null, 1.0, 0.2, 0.2, 6, 3, 3);
8      fem.setMaterial (new LinearMaterial (500000, 0.33));
9      RenderProps.setLineColor (fem, Color.BLUE);
10     RenderProps.setLineWidth (mech, 2);
11     mech.addModel (fem);
12     // fix leftmost nodes of the FEM
13     for (FemNode3d n : fem.getNodes()) {
14        if ((n.getPosition().x-(-0.5)) < 1e-8) {
15           n.setDynamic (false);
16        }
17     }
18
19     // create and add rigid body box
20     RigidBody box = RigidBody.createBox (
```

```
21          "box", 0.25, 0.1, 0.1, /*density=*/1000);
22       mech.add (box);
23
24       // create a basic frame and set its pose and axis length
25       Frame frame = new Frame();
26       frame.setPose (new RigidTransform3d (0.4, 0, 0, 0, Math.PI/4, 0));
27       frame.setAxisLength (0.3);
28       mech.addFrame (frame);
29
30       mech.attachFrame (frame, fem); // attach using element-based attachment
31
32       // attach the box to the FEM, using all the nodes of elements 31 and 32
33       HashSet<FemNode3d> nodes = collectNodes (fem, new int[] { 22, 31 });
34       FrameFem3dAttachment attachment = new FrameFem3dAttachment(box);
35       attachment.setFromNodes (box.getPose(), nodes);
36       mech.addAttachment (attachment);
37
38       // render the attachment nodes for the box as spheres
39       for (FemNode n : attachment.getNodes()) {
40          RenderProps.setSphericalPoints (n, 0.007, Color.GREEN);
41       }
42    }
```

Lines 3-22 create a `MechModel` and populate it with an FEM beam and a rigid body box. Next, a basic `Frame` is created, with a specified pose and an axis length of 0.3 (to allow it to be seen), and added to the `MechModel` (lines 25-28). It is then attached to the FEM beam using an element-based attachment (line 30). Meanwhile, the box is attached to using a nodal-based attachment, created from all the nodes associated with elements 22 and 31 (lines 33-36). Finally, all attachment nodes are set to be rendered as green spheres (lines 39-41).

To run this example in ArtiSynth, select All demos > tutorial > FrameFemAttachment from the Models menu. Running the model will cause it to settle into the state shown in Figure 6.11. Forces can interactively be applied to the attached block and frame using the pull tool, causing the FEM model to deform (see the section "Pull Manipulation" in the ArtiSynth User Interface Guide).

6.6.2 Adding joints to FEM models

The ability to connect frames to FEM models, as described in Section 6.6, makes it possible to interconnect different FEM models directly using joints, as described in Section 3.3. This is done internally by using `FrameFem3dAttachments` to connect frames C and D of the joint (Figure 3.7) to their respective FEM models.

As indicated in Section 3.3.3, most joints have a constructor of the form

```
JointType (bodyA, bodyB, TDW);
```

that creates a joint connecting `bodyA` to `bodyB`, with the initial pose of the D frame given (in world coordinates) by `TDW`. The same body and transform settings can be made on an existing joint using the method setBodies(bodyA, bodyB, TDW). For these constructors and methods, it is possible to specify FEM models for `bodyA` and/or `bodyB`. Internally, the joint then creates a `FrameFem3dAttachment` to connect frame C and/or D of the joint (See Figure 3.7) to the corresponding FEM model.

However, unlike joints involving rigid bodies or frames, there are no associated $\mathbf{T}_{CA}$ or $\mathbf{T}_{DB}$ transforms (since there is no fixed frame within an FEM to define such transforms). Methods or constructors which utilize $\mathbf{T}_{CA}$ or $\mathbf{T}_{DB}$ can therefore not be used with FEM models.

6.6.3 Example: two FEM beams connected by a joint

A model connecting two FEM beams by a joint is defined in

```
artisynth.demos.tutorial.JointedFemBeams
```

It creates two FEM beams and connects them via a special slotted-revolute joint. The build method is shown below:

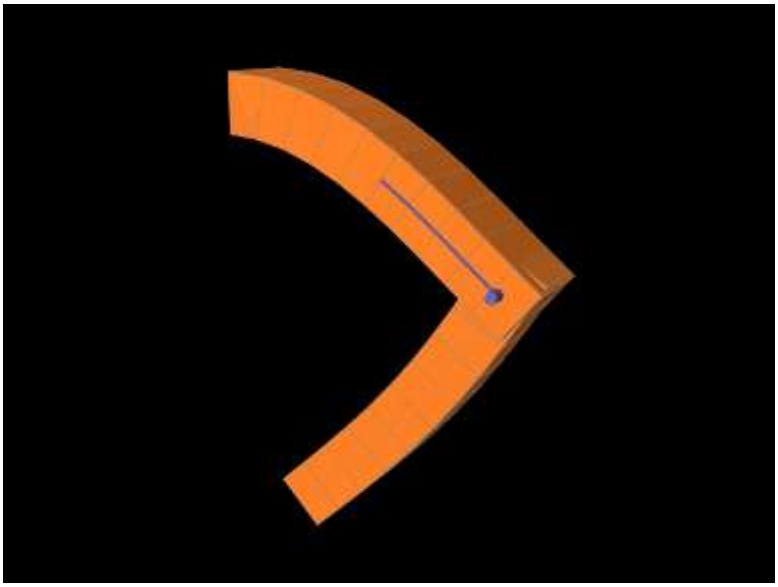


Figure 6.12: JointedFemBeams loaded into ArtiSynth and run until stable.

```
1   public void build (String[] args) {
2
3      MechModel mech = new MechModel ("mechMod");
4      addModel (mech);
5
6      double stiffness = 5000;
7      // create first fem beam and fix the leftmost nodes
8      FemModel3d fem1 = addFem (mech, 2.4, 0.6, 0.4, stiffness);
9      for (FemNode3d n : fem1.getNodes()) {
10        if (n.getPosition().x <= -1.2) {
11           n.setDynamic(false);
12        }
13     }
14     // create the second fem beam and shift it 1.5 to the right
15     FemModel3d fem2 = addFem (mech, 2.4, 0.4, 0.4, 0.1*stiffness);
16     fem2.transformGeometry (new RigidTransform3d (1.5, 0, 0));
17
18     // create a slotted revolute joint that connects the two fem beams
19     RigidTransform3d TDW = new RigidTransform3d(0.5, 0, 0, 0, 0, Math.PI/2);
20     SlottedRevoluteJoint joint = new SlottedRevoluteJoint (fem2, fem1, TDW);
21     mech.addBodyConnector (joint);
22
23     // set ranges and rendering properties for the joint
24     joint.setShaftLength (0.8);
25     joint.setMinX (-0.5);
26     joint.setMaxX (0.5);
27     joint.setSlotDepth (0.63);
28     joint.setSlotWidth (0.08);
29     RenderProps.setFaceColor (joint, myJointColor);
30  }
```

Lines 3-16 create a `MechModel` and populates it with two FEM beams, `fem1` and `fem2`, using an auxiliary method `addFem()` defined in the model source file. The leftmost nodes of `fem1` are set fixed. A `SlottedRevoluteJoint` is then created to interconnect `fem1` and `fem2` at a location specified by `TDW` (lines 19-21). Lines 24-29 set some parameters for the joint, along with various render properties.

To run this example in ArtiSynth, select All demos > tutorial > JointedFemBeams from the Models menu. Running the model will cause it drop and flex under gravity, as shown in 6.12. Forces can interactively be applied to the beams using the pull tool (see the section “Pull Manipulation” in the ArtiSynth User Interface Guide).

6.7 Incompressibility

FEM incompressibility within ArtiSynth is enforced by trying to ensure that the volume of an FEM remains locally constant. This, in turn, is accomplished by constraining nodal velocities so that the local volume change, or *divergence*, is zero (or close to zero). There are generally two ways to do this:

- *Hard incompressibility*, which sets up explicit constraints on the nodal velocities;
- *Soft incompressibility*, which uses a restoring pressure based on a potential field to try to keep the volume constant.

Both of these methods operate independently, and both can be used either separately or together. Generally speaking, hard incompressibility will result in incompressibility being more rigorously enforced, but at the cost of increased computation time and (sometimes) less stability. Soft incompressibility allows the application to control the restoring force used to enforce incompressibility, usually by adjusting the value of the *bulk modulus* material property. As the bulk modulus is increased, soft incompressibility starts to act more like 'hard' incompressibility, with an infinite bulk modulus corresponding to perfect incompressibility. However, very large bulk modulus values will generally produce stability problems.

> Incompressibility is not currently implemented for shell elements. Applying hard incompressibility to a shell element will have no effect on its behavior. If soft incompressibility is applied, by supplying the element with an incompressible material, then only the deviatoric component of that material will have any effect; the dilational component will generate no stress.

6.7.1 Volume regions and locking

Both hard and soft incompressibility can be applied to different regions of local volume. From larger to smaller, these regions are:

- *Nodal* - the local volume surrounding each node;
- *Element* - the volume of each element;
- *Full* - the volume at each integration point.

Element-based incompressibility is the standard method generally seen in the literature. However, it tends not to work well for tetrahedral meshes, because constraining the volume of each tet in a tetrahedral mesh tends to over constrain the system. This is because the number of tets in a large tetrahedral mesh is often $O(5n)$, where n is the number of nodes, and so putting a volume constraint on each element may result in $O(5n)$ constraints, which exceeds the $3n$ degrees of freedom (DOF) in the FEM. This overconstraining results in an artificially increased stiffness known as *locking*. Because of locking, for tetrahedrally based meshes it may be better to use nodal-based incompressibility, which creates a single volume constraint around each node, resulting in only n constraints, leaving $2n$ DOF to handle the remaining deformation. However, nodal-based incompressibility is computationally more costly than element-based and may not be as stable.

Generally, the best solution for incompressible problems is to use element-based incompressibility with a mesh consisting of hexahedra, or primarily hexahedra and a mix of other elements (the latter commonly being known as a *hex dominant* mesh). For hex-based meshes, the number of elements is roughly equal to the number of nodes, and so adding a volume constraint for each element imposes n constraints on the model, which (like nodal incompressibility) leaves $2n$ DOF to handle the remaining deformation.

Full incompressibility tries to control the volume at each integration point within each element, which almost always results in a large number of volumetric constraints and hence locking. It is therefore not commonly used and is provided mostly for debugging and diagnostic purposes.

6.7.2 Hard incompressibility

Hard incompressibility is controlled by the incompressible property of the FEM, which can be set to one of the following values of the enumerated type `FemModel.IncompMethod`:

OFF No hard incompressibility enforced.

ELEMENT Element-based hard incompressibility enforced (Section 6.7.1).

NODAL Nodal-based hard incompressibility enforced (Section 6.7.1).

AUTO Selects either `ELEMENT` or `NODAL`, with the former selected if the number of elements is less than or equal to the number of nodes.

ON Same as `AUTO`.

Hard incompressibility uses explicit constraints on the nodal velocities to enforce the incompressibility, which increases computational cost. Also, if the number of constraints is too large, *perturbed pivot* errors may be encountered by the solver. However, hard incompressibility can in principle handle situations where complete incompressibility is required. It is equivalent to the mixed u-P formulation used in commercial FEM codes (such as ANSYS), and the Lagrange multipliers computed for the constraints are pressure impulses.

Hard incompressibility can be applied in addition to soft incompressibility, in which case it will provide additional incompressibility enforcement on top of that provided by the latter. It can also be applied to linear materials, which are not themselves able to emulate true incompressible behavior (Section 6.7.4).

6.7.3 Soft incompressibility

Soft incompressibility enforces incompressibility using a restoring pressure that is controlled by a volume-based energy potential. It is only available for FEM materials that are subclasses of IncompressibleMaterial. The energy potential $U(J)$ is a function of the determinant J of the deformation gradient, and is scaled by the material's *bulk modulus* κ. The restoring pressure p is given by

$$p = \frac{\partial U}{\partial J}. \tag{6.7}$$

Different potentials can be selected by setting the bulkPotential property of the incompressible material, whose value is an instance of `IncompressibleMaterial.BulkPotential`. Currently there are two different potentials:

QUADRATIC The potential and associated pressure are given by

$$U(J) = \frac{1}{2}\kappa(J-1)^2, \quad p = \kappa(J-1). \tag{6.8}$$

LOGARITHMIC The potential and associated pressure are given by

$$U(J) = \frac{1}{2}\kappa(\ln J)^2, \quad p = \kappa\frac{\ln J}{J} \tag{6.9}$$

The default potential is `QUADRATIC`, which may provide slightly improved stability characteristics. However, we have not noticed significant differences between the two potentials in practice.

How soft incompressibility is applied within an FEM model is controlled by the FEM's softIncompMethod property, which can be set to one of the following values of the enumerated type `FemModel.IncompMethod`:

ELEMENT Element-based soft incompressibility enforced (Section 6.7.1).

NODAL Nodal-based soft incompressibility enforced (Section 6.7.1).

AUTO Selects either `ELEMENT` or `NODAL`, with the former selected if the number of elements is less than or equal to the number of nodes.

FULL Incompressibility enforced at each integration point (Section 6.7.1).

6.7.4 Incompressibility and linear materials

Within a linear material, incompressibility is controlled by Poisson's ratio ν, which for isotropic materials can assume a value in the range $[-1, 0.5]$. This specifies the amount of transverse contraction (or expansion) exhibited by the material as it compressed or extended along a particular direction. A value of 0 allows the material to be compressed or extended without any transverse contraction or expansion, while a value of 0.5 in theory indicates a perfectly incompressible material. However, setting $\nu = 0.5$ in practice causes a division by zero, so only values close to 0.5 (such as 0.49) can be used.

Moreover, the incompressibility only applies to small displacements, so that even with $\nu = 0.49$ it is still possible to squash a linear FEM completely flat if enough force is applied. If true incompressible behavior is desired with a linear material, then one must also use hard incompressibility (Section 6.7.2).

6.7.5 Using incompressibility in practice

As mentioned above, when modeling incompressible models, we have found that the best practice is to use, if possible, either a hex or hex-dominant mesh, along with element-based incompressibility.

Hard incompressibility allows the handling of full incompressibility but at the expense of greater computational cost and often less stability. When modeling biomechanical materials, it is often permissible to use only soft incompressibility, partly since biomechanical materials are rarely completely incompressible. When implementing soft incompressibility, it is common practice to set the bulk modulus to something like 100 times the other (deviatoric) stiffnesses of the material.

We have found stability behavior to be complex, and while hard incompressibility often results in less stable behavior, this is not always the case: in some situations the stronger enforcement afforded by hard incompressibility actually improves stability.

6.8 Varying and augmenting material behaviors

The default material used by all elements of an FEM model is supplied by the model's material property. However, it is often the case that one wishes to specify different material behaviors for different sets of elements within an FEM model. This may be particularly true when combining volumetric and shell elements.

There are several ways to vary material behavior within a model. These include:

- **Setting an explicit material for specific elements**, using their material property. An element's material is `null` by default, but if set to a material, it will override the default material supplied by the FEM model. While this method is quite straightforward, it does have one disadvantage: because material settings are *copied* by each element, subsequent interactive or online changes to the material require resetting the material in *all* the affected elements.

- **Binding one or more material parameters to a field**. Sometimes certain material parameters, such as stiffness quantities or direction information, may need to vary across the FEM domain. While sometimes this can be handled by setting material properties for specific elements, it may be more convenient to bind the varying properties to a *field*, which can specify varying values over a domain composed of either a regular grid or an FEM mesh. Only one material needs to be used, and any properties which are not bound can be adjusted interactively or online. Fields and their bindings are described in detail in the Section 6.10.

- **Adding augmenting material behaviors using MaterialBundles**. A material bundle may be specified either for all elements, or for a subset of them, and each provides one material (via its own material property) whose behavior is *added* to that of the indicated elements. This also provides an easy way to *combine* the behaviors of two of more materials in the same element. One also has the option of setting the material property of the certain elements to NullMaterial, so that *only* the augmenting material(s) are applied.

- **Adding muscle behaviors using MuscleBundles**. This is analogous to using `MaterialBundles`, except that `MuscleBundles` are restricted to using an instance of a MuscleMaterial, and include support for handling the excitation value, as well as the activation directions (which usually vary across the FEM domain). `MuscleBundles` are only present in the `FemMuscleModels` subclass of `FemModel3d`, and are described in detail in Section 6.9.

The remainder of this section will discuss `MaterialBundles`.

Adding a MaterialBundle to an FEM model is illustrated by the following code fragment:

```
FemMaterial extraMat;   // material to be added
FemModel3d fem;         // FEM model
...

MaterialBundle bun = new MaterialBundle ("mybundle", extraMat);
// add volumetric elements to the bundle
for (FemElement3d e : fem.getElements()) {
   if (/* e should be added to the bundle */) {
      bun.addElement (e);
   }
}
fem.addMaterialBundle (bun);
```

Once added, the stress computed by the bundle's material will be *added* to the stress computed by any other materials which are active for the bundle's elements.

When deciding what elements to add to a bundle, one is free to choose any means necessary. The example above inspects all the volumetric elements in the FEM. To instead inspect all the shell elements, or all volumetric and shell elements, one could use the code fragments such as the following:

```
// add shell elements to the bundle
for (ShellElement3d e : fem.getShellElements()) {
   if (/* e should be added to the bundle */) {
      bun.addElement (e);
   }
}

// add volumetric or shell elements to the bundle
for (FemElement3dBase e : fem.getAllElements()) {
   if (/* e should be added to the bundle */) {
      bun.addElement (e);
   }
}
```

Of course, if the elements are known through other means, then they can be added directly.

The element composition of a bundle can be controlled by the methods

```
void addElement (FemElement3dBase e)        // adds an element
boolean removeElement (FemElement3dBase e)  // removes an element
void clearElements()                        // removes all elements

int numElements()                           // gets the number of elements
FemElment3dBase getElement (int idx)        // gets the idx-th element
```

It is also possible to create a `MaterialBundle` whose material is added to *all* the FEM elements. This can done either by using one of the following constructors

```
MaterialBundle (String name, boolean useAllElements)
MaterialBundle (String name, FemMaterial mat, boolean useAllElements)
```

with `useAllElements` set to `true`, or by calling the method

```
void setUseAllElements (boolean enable)
```

When a bundle is set to use all the FEM elements, it clears its own element list, and one is not permitted to add elements using `addElement(elem)`.

After a bundle has been created, it is possible to get or set its material property using the methods

```
FemMateral getMaterial()                  // get the material
FemMateral setMaterial (FemMaterial mat)  // set the material
```

Again, because materials are copied internally, any modification to a material *after* it has been used as an input to `setMaterial()` will *not* be noticed by the bundle. Instead, one should modify the material *before* calling `setMaterial()`, or modify the *copied* material which can be obtained by calling `getMaterial()` or by storing the value returned by `setMaterial()`.

Finally, a `MuscleBundle` is a renderable component. Setting its elementWidgetSize property to a value greater than zero will cause the rendering of all its elements, using a solid widget representation of each at a scale controlled by elementWidgetSize: 0.5 for half size, 1.0 for full size, etc. The color of the widgets is controlled by the faceColor property of the bundle's renderProps. Being able to render the elements makes it easy to select the bundle and visualize which elements it contains.

6.8.1 Example: FEM sheet with a stiff spine

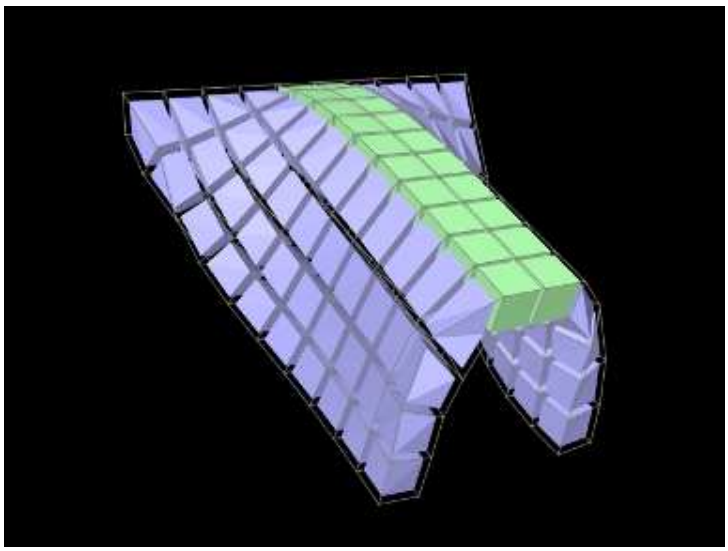


Figure 6.13: MaterialBundleDemo model after being run in ArtiSynth.

A simple model demonstrating the use of material bundles is defined in

```
artisynth.demos.tutorial.MaterialBundleDemo
```

It consists of a simple thin hexahedral sheet in which a material bundle is used to stiffen elements close to the x-axis, creating a stiff "spine". While the same effect could be achieved by simply setting a different material property for the "spine" elements, it does provide a good example of muscle bundle usage. The model's `build` method is given below:

```
1    public void build (String[] args) {
2       MechModel mech = new MechModel ("mech");
3       addModel (mech);
4
5       // create a fem model consisting of a thin sheet of hexes
6       FemModel3d fem = FemFactory.createHexGrid (null, 1.0, 1.0, 0.1, 10, 10, 1);
7       fem.setDensity (1000);
8       fem.setMaterial (new LinearMaterial (10000, 0.45));
9       mech.add (fem);
10      // fix the left-most nodes:
11      double EPS = 1e-8;
12      for (FemNode3d n : fem.getNodes()) {
13         if (n.getPosition().x <= -0.5+EPS) {
14            n.setDynamic (false);
15         }
16      }
17      // create a "spine" of stiffer elements using a MaterialBundle with a
18      // stiffer material
19      MaterialBundle bun =
```

```
20            new MaterialBundle ("spine", new NeoHookeanMaterial (5e6, 0.45), false);
21         for (FemElement3d e : fem.getElements()) {
22            // use element centroid to determine which elements are on the "spine"
23            Point3d pos = new Point3d();
24            e.computeCentroid (pos);
25            if (Math.abs(pos.y) <= 0.1+EPS) {
26               bun.addElement (e);
27            }
28         }
29         fem.addMaterialBundle (bun);
30
31         // add a control panel to control both the fem and bundle materials,
32         // as well as the fem and bundle widget sizes
33         ControlPanel panel = new ControlPanel();
34         panel.addWidget ("fem material", fem, "material");
35         panel.addWidget ("fem widget size", fem, "elementWidgetSize");
36         panel.addWidget ("bundle material", bun, "material");
37         panel.addWidget ("bundle widget size", bun, "elementWidgetSize");
38         addControlPanel (panel);
39
40         // set rendering properties, using element widgets
41         RenderProps.setFaceColor (fem, new Color (0.7f, 0.7f, 1.0f));
42         RenderProps.setFaceColor (bun, new Color (0.7f, 1.0f, 0.7f));
43         bun.setElementWidgetSize (0.9);
44         fem.setElementWidgetSize (0.8);
45      }
```

Lines 6-9 create a thin FEM hex sheet, centered on the origin, with size $1 \times 1 \times 0.1$ and $10 \times 10 \times 1$ elements along each axis. Its material is set to a linear material with a Young's modulus of 10000. The leftmost nodes are then fixed (lines 11-16).

Next, a `MuscleBundle` is added and used to apply an additional material to the elements near the x axis (lines 19-29). It is given a neo-hookean material with a much higher stiffness, and the "spine" elements for it are selected by finding those whose centroids have a y value within 0.1 of the x axis.

After the bundle is added, a control panel is created allowing interactive control of the materials for both the FEM model and the bundle, along with their elementWidgetSize properties.

Finally, some render properties are set (lines 41-44). The idea is to render the elements of both the FEM model and the bundle using element widgets (both of which will be visible since there is no surface rendering and the element widget sizes for both are greater than 0). The widget size for the bundle is made larger than that of the FEM model to ensure its widgets will cover those of the latter. Widget colors are controlled by the FEM and bundle face colors, set in lines 41-42.

The example can be run in ArtiSynth by selecting All demos > tutorial > MaterialBundleDemo from the Models menu. When it is run, the sheet will fall under gravity but be much stiffer along the spine (Figure 6.13). The control panel can be used to interactively adjust the material parameters for both the FEM and the bundle. This is one advantage of using bundles: the same material can be used to control multiple elements. The panel also allows interactive adjustment of the widget sizes, to illustrate what they look like and how they are rendered.

6.9 Muscle activated FEM models

Finite element muscle models are an extension to regular FEM models. As such, everything previously discussed for regular FEM models also applies to FEM muscles. Muscles have additional properties that allow them to contract when activated. There are two types of muscles supported:

Fibre-based: Point-to-point muscle fibres are embedded in the model.

Material-based: An auxiliary material is added to the constitutive law to embed muscle properties.

In this section, both types will be described.

6.9.1 FemMuscleModel

The main class for FEM-based muscles is FemMuscleModel, a subclass of FemModel3d. It differs from a basic FEM model in that it has the new property

Property	Description
`muscleMaterial`	An object that adds an activation-dependent 'muscle' term to the *constitutive law*.

This is a delegate object of type MuscleMaterial that computes activation-dependent stress and stiffness in the muscle. In addition to this property, `FemMuscleModel` adds two new lists of subcomponents:

`bundles`
: Groupings of muscle sub-units (fibres or elements) that can be activated.

`exciters`
: Components that control the activation of a set of bundles or other exciters.

6.9.1.1 Bundles

Muscle bundles allow for a muscle to be partitioned into separate groupings of fibres/elements, where each bundle can be activated independently. They are implemented in the class MuscleBundle. Bundles have three key properties:

Property	Description
`excitation`	Activation level of the muscle, $a \in [0,1]$.
`fibresActive`	Enable/disable "fibre-based" muscle components.
`muscleMaterial`	An object that adds an activation-dependent 'muscle' term to the *constitutive law*.

The `excitation` property controls the level of muscle activation, with zero being no muscle action, and one being fully activated. The `fibresActive` property is a boolean variable that controls whether or not to treat any contained fibres as point-to-point-like muscles ("fibre-based"). If false, the fibres are ignored. The third property, `muscleMaterial`, allows for a `MuscleMaterial` to be specified per bundle. By default, its value is inherited from `FemMuscleModel`.

Once a muscle bundle is created, muscle sub-units must be assigned to it. These are either point-to-point fibres, or material-based muscle element descriptors. The two types will be covered in Sections 6.9.2 and 6.9.3, respectively.

6.9.1.2 Exciters

Muscle exciters enable you to simultaneously activate a group of "excitation components". This includes: point-to-point muscles, muscle bundles, muscle fibres, material-based muscle elements, and other muscle exciters. Components that can be excited all implement the ExcitationComponent interface. To add or remove a component to the exciter, use

```
addTarget (ExcitationComponent ex);     // adds a component to the exciter
addTarget (ExcitationComponent ex,      // adds a component with a gain factor
    double gain);
removeTarget (ExcitationComponent ex); // removes a component
```

If a gain factor is specified, the activation is scaled by the gain for that component.

6.9.2 Fibre-based muscles

In fibre-based muscles, a set of point-to-point muscle fibres are added between FEM nodes or markers. Each fibre is assigned an AxialMuscleMaterial, just like for regular point-to-point muscles (Section 4.4.1). Note that these muscle materials typically have a "rest length" property, that will likely need to be adjusted for each fibre. Once the set of fibres are added to a `MuscleBundle`, they need to be enabled. This is done by setting the `fibresActive` property of the bundle to `true`.

Fibres are added to a `MuscleBundle` using one of the functions:

```
addFibre( Muscle muscle );              // adds a point-to-point fibre
Muscle addFibre( Point p0, Point p1,    // creates and adds a fibre
    AxialMuscleMaterial mat);
```

The latter returns the newly created `Muscle` fibre. The following code snippet demonstrates how to create a fibre-based `MuscleBundle` and add it to an FEM muscle.

```
// Create a muscle bundle
MuscleBundle bundle = new MuscleBundle("fibres");
Point3d[] fibrePoints = ...   //create a sequential list of points

// Add fibres
Point pPrev = fem.addMarker(fibrePoints[0]);     // create an FEM marker
for (int i=1; i<=fibrePoints.length; i++) {
   Point pNext = fem.addMarker(fibrePoint[i]);

   // Create fibre material
   double l0 = pNext.distance(pPrev);            // rest length
   AxialMuscleMaterial fibreMat =
         new BlemkerAxialMuscle(
               1.4*l0, l0, 3000, 0, 0);

   // Add a fibre between pPrev and pNext
   bundle.addFibre(pPrev, pNext, fibreMat);      // add fibre to bundle
   pPrev = pNext;
}

// Enable use of fibres (default is disabled)
bundle.setFibresActive(true);
fem.addMuscleBundle(bundle);                     // add the bundle to fem
```

In these fibre-based muscles, force is only exerted between the anchor points of the fibres; it is a discrete approximation. These models are typically more stable than material-based ones.

6.9.3 Material-based muscles

In material-based muscles, the constitutive law is augmented with additional terms to account for muscle-specific properties. This is a continuous representation within the model.

The basic building block for a material-based muscle bundle is a MuscleElementDesc. This object contains a reference to a `FemElement3d`, a `MuscleMaterial`, and either a single direction or set of directions that specify the direction of contraction. If a single direction is specified, then it is assumed the entire element contracts in the same direction. Otherwise, a direction can be specified for each *integration point* within the element. A `null` direction signals that there is no muscle at the corresponding point. This allows for a sub-element resolution for muscle definitions. The positions of integration points for a given element can be obtained with:

```
// loop through all integration points for a given element
for ( IntegrationPoint3d ipnt : elem.getIntegrationPoints() ) {
   Point3d curPos = new Point3d();
   Point3d restPos = new Point3d();
   ipnt.computePosition (curPos, elem);          // computes current position
   ipnt.computeRestPosition (restPos, elem);     // computes rest position
}
```

By default, the `MuscleMaterial` is inherited from the bundle's `material` property. Supported muscle materials include: GenericMuscle, BlemkerMuscle, and FullBlemkerMuscle. The Blemker-type materials are based on [2]. `BlemkerMuscle` only uses the muscle-specific terms (since a base material is provided the underlying FEM model), whereas `FullBlemkerMuscle` adds all terms described in the aforementioned paper.

Elements can be added to a muscle bundle using one of the methods:

```
// Adds a muscle element
addElement (MuscleElementDesc elem);
```

```
// Creates and adds a muscle element
MuscleElementDesc addElement (FemElement3d elem, Vector3d dir);
// Sets a direction per integration point
MuscleElementDesc addElement (FemElement3d elem, Vector3d[] dirs);
```

The following snippet demonstrates how to create and add a material-based muscle bundle:

```
// Create muscle bundle
MuscleBundle bundle = new MuscleBundle("embedded");

// Muscle material
MuscleMaterial muscleMat = new BlemkerMuscle(
      1.4, 1.0, 3000, 0, 0);
bundle.setMuscleMaterial(muscleMat);

// Muscle direction
Vector3d dir = Vector3d.X_UNIT;

// Add elements to bundle
for (FemElement3d elem : beam.getElements()) {
   bundle.addElement(elem, dir);
}

// Add bundle to model
beam.addMuscleBundle(bundle);
```

6.9.4 Example: comparison with two beam examples

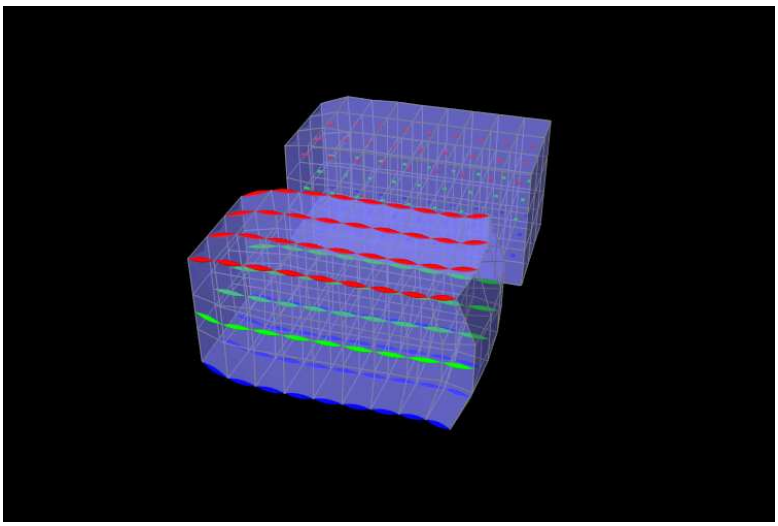


Figure 6.14: FemMuscleBeams model loaded into ArtiSynth.

An example comparing a fibre-based and a material-based muscle is shown in Figure 6.14. The code can be found in `artisynth.demos.tutorial.FemMuscleBeams`. There are two `FemMuscleModel` beams in the model: one fibre-based, and one material-based. Each has three muscle bundles: one at the top (red), one in the middle (green), and one at the bottom (blue). In the figure, both muscles are fully activated. Note the deformed shape of the beams. In the fibre-based one, since forces only act between point on the fibres, the muscle seems to bulge. In the material-based muscle, the entire continuous volume contracts, leading to a uniform deformation.

Material-based muscles are more realistic. However, this often comes at the cost of stability. The added terms to the constitutive law are highly nonlinear, which may cause numerical issues as elements become highly contracted or highly deformed. Fibre-based muscles are, in general, more stable. However, they can lead to bulging and other deformation artifacts due to their discrete nature.

6.10 Fields

In modeling applications, particularly those employing FEM methods, situations often arise where it is necessary to describe numeric quantities that vary over some spatial domain. For example, the stiffness parameters of an FEM material may vary at different points within the volumetric mesh. When modeling muscles, the activation direction vector will also typically vary over the mesh.

ArtiSynth provides *field* objects which can be used to represent such spatially varying quantities, together with mechanisms to attach these to properties within FEM materials. Field objects can provide either *scalar* or *vector* fields. Scalar fields implement the interface ScalarField and supply the method

```
double getValue (Point3d pos)
```

while vector fields implement VectorField and supply the method

```
T getValue (Point3d pos)
```

where `T` is any class implementing maspack.matrix.VectorObject. (Most of the fixed-size matrix objects in `maspack.matrix`, along with `MatrixNd` and `VectorNd`, implement `VectorObject`.) In both cases, the idea is to provide values at arbitrary points over some spatial domain.

> Both `ScalarField` and `VectorField` are subclassed from `Field`. The reason for separating scalar and vector fields is simply efficiency: having scalar fields work with the primitive type `double`, instead of the object `Double`, requires considerably less storage and somewhat less computational effort.

6.10.1 Field types

A field is typically implemented by specifying a finite set of values at discrete locations on an underlying spatial grid and then using different methods to determine values at arbitrary points.

At present, ArtiSynth provides the following different field types:

Grid fields Implemented by classes that extend either ScalarGridField or VectorGridField, grid fields specify their values at the vertices of a regular 3D grid. Values at an arbitrary position **p** are determined by finding the grid cell containing **p**, and then using trilinear interpolation of the surrounding nodal values.

Nodal fields Implemented by classes that extend either ScalarNodalField or VectorNodalField, nodal fields specify their values at the nodes of an FEM mesh. Values at an arbitrary position **p** are determined by finding the element containing **p**, and then finding the nodal coordinates (i.e., weights) of **p** within that element and interpolating the value accordingly. Values within a nodal field can also be queried directly either by node, or a set of node numbers and weights:

```
T getValue (FemNode3d node)

T getValue (int[] nodeNums, double[] weights)
```

Element fields Implemented by classes that extend either ScalarElementField or VectorElementField, element fields specify their values at the elements of an FEM mesh, with the value assumed to be constant over the element. Values at an arbitrary position **p** are determined by finding the element containing **p** and then returning the value for that element. Values within a nodal field can also be queried directly by element:

```
T getValue (FemElement3dBase elem)
```

Sub-element fields Implemented by classes that extend either ScalarSubElemField or VectorSubElemField, sub-element fields specify their values at specific points within each element of an FEM mesh, with the points corresponding to the element's integration points. Values at an arbitrary position **p** are determined by finding the element containing **p**, extrapolating the point values back to the nodes, and then using nodal interpolation. While this is computationally expensive (and it is also possible to create a nodal field that gives the same result), the purpose of sub-element fields is to provide precise values at element integration points, which can be queried directly using the element and point sub-index:

Field component	value	field type
modelbase.ScalarGridField	double	scalar grid
modelbase.VectorGridField	VectorObject	vector grid
femmodels.ScalarNodalField	double	scalar nodal
femmodels.VectorNodalField	VectorObject	vector nodal
femmodels.Vector3dNodalField	Vector3d	vector nodal
femmodels.VectorNdNodalField	VectorNd	vector nodal
femmodels.MatrixNdNodalField	MatrixNd	vector nodal
femmodels.ScalarElementField	double	scalar element
femmodels.VectorElementField	VectorObject	vector element
femmodels.Vector3dElementField	Vector3d	vector element
femmodels.VectorNdElementField	VectorNd	vector element
femmodels.MatrixNdElementField	MatrixNd	vector element
femmodels.ScalarSubElemField	double	scalar sub-element
femmodels.VectorSubElemField	VectorObject	vector sub-element
femmodels.Vector3dSubElemField	Vector3d	vector sub-element
femmodels.VectorNdSubElemField	VectorNd	vector sub-element
femmodels.MatrixNdSubElemField	MatrixNd	vector sub-element

Table 6.6: List of all the field components available in ArtiSynth, with their value and field types.

```
T getValue (FemElement3dBase elem, int subIdx)
```

Internally, field features, such as nodes or elements, are generally referred to by their *number*. This is because node and element numbers are guaranteed to be *persistent*, in the sense that the number will remain unchanged as long as the node or element is not removed from the FEM model. Element field implementations also need to distinguish between volumetric and shell elements, since these are each numbered independently.

Table 6.6 gives a list of all the currently implemented fields. Some of them, such as those prefaced with `Vector3d`, `VectorNd`, and `MatrixNd`, are specialized because they implement special features for their value type, or require specialized construction because the value type does not have a fixed size.

Each field component also has a *default value*, which is returned for any query for which the specified position lies outside the field's domain. For a grid field, this means lying outside the bounds of the grid. For an FEM-based field, this means lying outside the FEM mesh. The default value can be specified in the field's constructor. If not specified, it is usually set to 0.

> Having a default value also means that explicit values do not have to specified at all of the features (e.g., nodes or elements) used to define the field. If an explicit value is missing, the default value is used instead.

6.10.2 Adding fields to a model

Fields are added to an ArtiSynth model by first creating an instance of the desired field class, populating it with values (at vertices, nodes, or elements, as required by the field), and then adding it to the model. Grid fields can generally be added anywhere, whereas FEM-based fields should be added to the fields sublist of their associated FEM model (grid fields can be added there as well).

As a simple example, one can construct a scalar nodal field, associated with an FEM model named `fem`, like this:

```
FemModel3d fem;
// ... build the fem ...
ScalarNodalField field = new ScalarNodalField ("stiffness", fem, 0);
for (FemNode3d n : fem.getNodes()) {
   double value = ... // compute value for the node
   field.setValue (n, value);
}
fem.addField (field); // add the field to the fem
```

Each field type has several constructors available. The one used above takes a component name (`"stiffness"`), the FEM modal associated with the field, and a default value (0 in this case). When setting field values, it is often easiest to simply specify the node (although the node's number (i.e., `node.getNumber()`) can be specified as well. After being created and initialized, the field is added to the FEM model using its `addField()` method.

> As noted in the previous section, it is not necessary to specify values for all the features of a field (e.g., nodes for a nodal field, or elements for an element field). For features with undefined values, the default value will be used instead.

As another example, consider constructing a vector element field, with two-dimensional values characterized by `Vector2d`. An example of this is as follows:

```
VectorElementField<Vector2d> field =
   new VectorElementField<> ("uv", Vector2d.class, fem);
for (FemElement3d e : fem.getElements()) {
   Vector2d value = ... // compute value for the element
   field.setValue (e, value);
}
fem.addField (field); // add the field to the fem
```

General vector fields are declared using type parameterization, in which the class type of the vector must be appended to the declaration between angle brackets (`< >`). Also, the constructor also requires the class type to be specified as an argument. In the example above, the field is constructed with a name (`"uv"`), value class type (`Vector2d.class`), and the FEM model. Since no default value is specified, a value of 0 will be assumed.

Since it is widely used, the vector type `Vector3d` has its own predefined field classes, which don't require parameterization and may also contain special methods and features relevant for 3D vectors. A `Vector3d` nodal field can be declared and created as follows:

```
Vector3dNodalField field = new Vector3dNormalField ("normals", fem);
```

Since the vector type is explicitly "wired in", no type parameterization is required, and `Vector3d.class` does not need to be specified to the constructor.

The vector types `VectorNd` and `MatrixNd` also have special predefined classes, because these components do not have a predetermined size and so it is necessary to specify this size information when constructing the field. Some constructors for fields with these types look like:

```
MatrixNdElementField field =
   new MatrixNdElementField ("matrixField", 3, 6, fem);
VectorNdNodalField field =
   new VectorNdNodalField ("params", 7, fem);
```

Both of these contain size information after the name argument: 3×6 for the matrix field, and 7 for the vector field. Again, no default value is specified, so 0 will be assumed.

Lastly, we consider constructing a sub-element field. This is normally done when we need precisely computed information at each of the element's integration points, and we can't assume that nodal interpolation will give an accurate enough approximation. A general template for constructing a sub-element field might look like this:

```
ScalarSubElemField field = new ScalarSubElemField ("precise", fem, -1);
for (FemElement3d e : fem.getElements()) {
   IntegrationPoint3d[] ipnts = e.getAllIntegrationPoints();
   for (int k=0; k<ipnts.length; k++) {
      double value = ... // compute value at integration point k
      field.setValue (e, k, value);
   }
}
fem.addField (field); // add the field to the fem
```

Here, we create a scalar sub-element field named `"precise"` with a default value of -1. Then we iterate first through the FEM elements, and then through each element's integration points, computing values appropriate to each one. Since we are likely to need information from the integration points themselves to compute the value, we use `e.getAllIntegrationPoints()` to obtain a list of them.

> Assigning values to a sub-element field, it is important to use the element's `getAllIntegrationPoints()` method instead of `getIntegrationPoints()`, since the former returns a list of *all* integration points, including the special *warping* point which is contained in the element's center and is used by the solver under some circumstances. Likewise, the number of sub-element values required is given by `numAllIntegrationPoints()`.

When computing sub-element field values, if it is necessary to obtain either the rest or current (spatial) position for an integration point, these can be obtained as follows:

```
IntegrationPoint3d ipnt;
Point3d pos = new Point3d();
FemElement3d elem;

...

// compute spatial position:
ipnt.computePosition (pos, elem.getNodes());

// compute rest position:
ipnt.computeRestPosition (pos, elem.getNodes());
```

6.10.3 Binding to material properties

Once a field has been created and added to the model, one may *bind* it to certain properties of a `FemMaterial`. When this is done, then whenever those properties are queried internally by the solver at the integration points for each FEM element, the field value at that integration point is used instead of the regular property value.

> As long as a property XXX is bound to a field, its regular value will still appear (and can be set) through widgets in control or property panels, or via the `getXXX()` and `setXXX()` accessors. However, the regular value won't be used internally by the FEM simulation.

To bind a property to a field, it is necessary that

1. The type of the field matches the value of the property;
2. The property is itself *bindable*.

If the property has a `double` value, then it can be bound to any `ScalarField`. Otherwise, if the property has a value `T`, where `T` is an instance of `maspack.matrix.VectorObject`, then it can be bound to any `VectorField<T>`.

Bindable properties export two methods with the signature

```
setXXXField (T field, boolean useRestPos)

T getXXXField ()
```

where `XXX` is the property name and `T` is an appropriate field type. The second argument, `useRestPos`, is relevant only for grid fields, and indicates whether the grid should be queried using the integration point's *rest* position, or current *spatial* position. For FEM-based fields, this distinction is irrelevant, since the field is assumed to always move with the FEM.

For example, consider the YoungsModulus property of a NeoHookeanMaterial material. This has a `double` value, and is bindable, and so can be bound to a `ScalarField` as follows:

```
FemModel3d fem;
ScalarNodalField field;
NeoHookeanMaterial mat;

... other initialization ...

mat.setYoungsModulusField (field, false);  // bind to the field
fem.setMaterial (mat);   // set the material in the FEM model
```

It is important to perform field bindings on materials **before** they are set in an FEM model (or one of its subcomponents, such as `MuscleBundles`). That's because the `setMaterial()` method *copies* the input material, and so any settings made on it afterwards won't be seen by the FEM:

```
// works: field will be seen by the copied version of 'mat'
mat.setYoungsModulusField (field, false);
fem.setMaterial (mat);

// does NOT work: field not seen by the copied version of 'mat'
fem.setMaterial (mat);
mat.setYoungsModulusField (field, false);
```

To unbind YoungsModulus, one can do

```
mat.setYoungsModulusField (null, false);
```

where the value of the `useRestPos` argument is ignored.

6.10.4 Example: FEM with variable stiffness

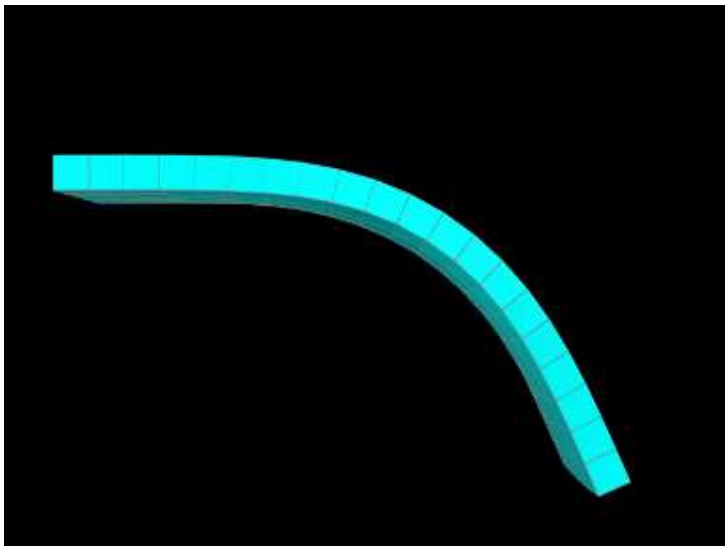


Figure 6.15: VariableStiffness model after being run in ArtiSynth.

A simple model demonstrating a stiffness that varies over an FEM mesh is defined in

```
artisynth.demos.tutorial.VariableStiffness
```

It consists of a simple thin hexahedral beam with a linear material for which the Young's modulus E is made to vary nonlinearly along the x axis of the rest position according to the formula

$$E = \frac{10^8}{1 + 1000x^3} \tag{6.10}$$

The model's `build` method is given below:

```
1   public void build (String[] args) {
2      MechModel mech = new MechModel ("mech");
3      addModel (mech);
4
5      // create regular hex grid FEM model
```

```
6       FemModel3d fem = FemFactory.createHexGrid (
7          null, 1.0, 0.25, 0.05, 20, 5, 1);
8       fem.transformGeometry (new RigidTransform3d (0.5, 0, 0)); // shift right
9       fem.setDensity (1000.0);
10      mech.addModel (fem);
11
12      // fix the left-most nodes
13      double EPS = 1e-8;
14      for (FemNode3d n : fem.getNodes()) {
15         if (n.getPosition().x < EPS) {
16            n.setDynamic (false);
17         }
18      }
19      // create a scalar nodel field to make the stiffness vary
20      // nonlinearly along the rest position x axis
21      ScalarNodalField stiffnessField = new ScalarNodalField(fem, 0);
22      for (FemNode3d n : fem.getNodes()) {
23         double s = 10*(n.getRestPosition().x);
24         double E = 100000000*(1/(1+s*s*s));
25         stiffnessField.setValue (n, E);
26      }
27      fem.addField (stiffnessField);
28      // create a linear material, bind its Youngs modulus property to the
29      // field, and set the material in the FEM model
30      LinearMaterial linearMat = new LinearMaterial (100000, 0.49);
31      linearMat.setYoungsModulusField (stiffnessField, /*useRestPos=*/true);
32      fem.setMaterial (linearMat);
33
34      // set some render properties for the FEM model
35      fem.setSurfaceRendering (SurfaceRender.Shaded);
36      RenderProps.setFaceColor (fem, Color.CYAN);
37   }
```

Lines 6-10 create the hex FEM model and shift it so that the left side is aligned with the origin, while lines 12-17 fix the leftmost nodes. Lines 21-27 create a scalar nodal field for the Young's modulus, with lines 23-24 computing E according to (6.10). The field is then bound to a linear material which is then set in the model (lined 30-32).

The example can be run in ArtiSynth by selecting All demos > tutorial > VariableStiffness from the Models menu. When it is run, the beam will bend under gravity, but mostly on the right side, due to the much higher stiffness on the left (Figure 6.15).

6.10.5 Example: specifying FEM muscle directions

Another example involves using a Vector3d field to specify the muscle activation directions over an FEM model and is defined in

```
artisynth.demos.tutorial.RadialMuscle
```

When muscles are added using MuscleBundles (Section 6.9), the muscle directions are stored and handled internally by the muscle bundle itself. However, it is possible to add a MuscleMaterial directly to the elements of an FEM model, using a MaterialBundle (Section 6.8), in which case the directions need to be set explicitly using a field.

The model's `build` method is given below:

```
1    public void build (String[] args) {
2       MechModel mech = new MechModel ("mech");
3       addModel (mech);
4
5       // create a thin cylindrical FEM model with two layers along z
6       double radius = 0.8;
7       FemMuscleModel fem = new FemMuscleModel ("radialMuscle");
8       mech.addModel (fem);
9       fem.setDensity (1000);
```

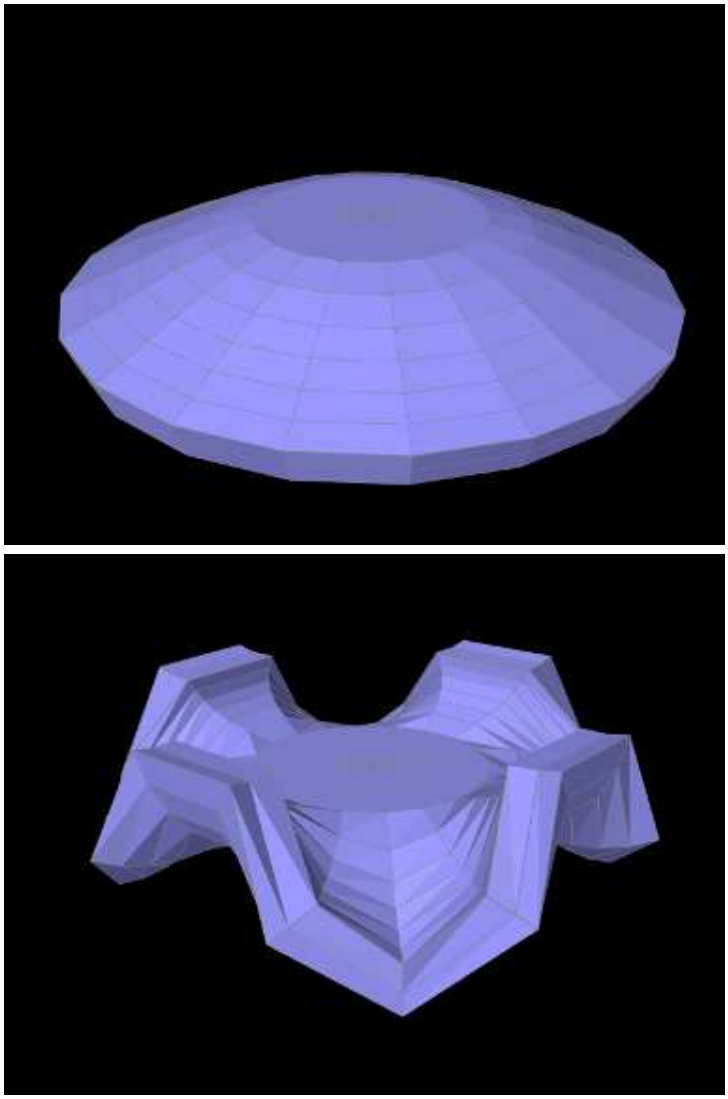


Figure 6.16: (Top) RadialMuscle model after being loaded into ArtiSynth and run with its excitation set to 0. (Bottom) RadialMuscle with its excitation set to around .375.

```
      FemFactory.createCylinder (fem, radius/8, radius, 20, 2, 8);
      fem.setMaterial (new NeoHookeanMaterial (200000.0, 0.33));
      // fix the nodes close to the center
      for (FemNode3d node : fem.getNodes()) {
         Point3d pos = node.getPosition();
         double radialDist = Math.sqrt (pos.x*pos.x + pos.y*pos.y);
         if (radialDist < radius/2) {
            node.setDynamic (false);
         }
      }
      // compute a direction field, with the directions arranged radially
      Vector3d dir = new Vector3d();
      Vector3dElementField dirField = new Vector3dElementField (fem);
      for (FemElement3d elem : fem.getElements()) {
         elem.computeCentroid (dir);
         // set directions only for the upper layer elements
         if (dir.z > 0) {
            dir.z = 0; // remove z component from direction
            dir.normalize();
            dirField.setValue (elem, dir);
         }
      }
```

```
32        fem.addField (dirField);
33        // add a muscle material, and use it to hold a simple force
34        // muscle whose 'restDir' property is attached to the field
35        MaterialBundle bun = new MaterialBundle ("bundle",/*all elements=*/true);
36        fem.addMaterialBundle (bun);
37        SimpleForceMuscle muscleMat = new SimpleForceMuscle (500000);
38        muscleMat.setRestDirField (dirField, true);
39        bun.setMaterial (muscleMat);
40
41        // add a control panel to control the excitation
42        ControlPanel panel = new ControlPanel();
43        panel.addWidget (bun, "material.excitation", 0, 1);
44        addControlPanel (panel);
45
46        // set some rendering properties
47        fem.setSurfaceRendering (SurfaceRender.Shaded);
48        RenderProps.setFaceColor (fem, new Color (0.6f, 0.6f, 1f));
49     }
```

Lines 5-19 create a thin cylindrical FEM model, centered on the origin, with radius r and height $r/8$, consisting of hexes with wedges at the center, with two layers of elements along the z axis (which is parallel to the cylinder axis). Its base material is set to a neo-hookean material. To keep the model from falling under gravity, all nodes whose distance to the z axis is less than $r/2$ are fixed.

Next, a Vector3d field is created to specify the directions, on a per-element basis, for the muscle material which will be added subsequently (lines 21-32). While we could create an instance of `VectorElementField<Vector3d>`, we use Vector3dElementField, since this is available and may provide additional functionality (such as the ability to render the directions). Directions are set to lie outward in a radial direction perpendicular to the z axis, and since the model is centered on the origin, they can be computed easily by first computing the element centroids, removing the z component, and then normalizing. In order to give the muscle action an upward bias, we only set directions for elements in the upper layer. Direction values for elements in the lower layer will then automatically have a default value of 0, which will cause the muscle material to not apply any stress.

We next add to the model a muscle material whose directions will be determined by the field. To hold the material, we first create and add a `MaterialBundle` which is set to act on all elements (line 35-36). Then we set this bundle's material to `SimpleForceMuscle`, which simply adds a stress along the muscle direction that equals the excitation value times the value of its maxStress property, and bind the material's restDir property to the direction field (lines 37-39).

Finally, we create and add a control panel to allow interactive control over the muscle material's excitation property (lines 42-44), and set some rendering properties for the FEM model.

The example can be run in ArtiSynth by selecting All demos > tutorial > RadialMuscle from the Models menu. When it is first runs, it falls around the edges under gravity (Figure 6.16, top). Applying an excitation causes a radial contraction which pulls the edges upward and, if high enough, causes then to buckle (Figure 6.16, bottom).

6.11 Collisions

As described in Section 4.5, collisions can be enabled for any class that implements the Collidable interface. Both FemModel3d and FemMeshComp implement Collidable. `FemModel3d` will use its surface mesh as the collision surface. A `FemMeshComp` will use its underlying mesh structure. At present, only meshes of type PolygonalMesh are supported.

Since `FemMeshComp` is also a `Collidable`, this means we can enable collisions with any embedded mesh inside an FEM. Any forces resulting from the collision are then automatically transferred back to the underlying nodes of the model using Equation (6.5).

Note: Collisions involving shell elements are not yet fully supported. This relates to the fact that shells are thin and can therefore pass through each other easily in a single time step. However, collisions *should* work properly if

1. The collider type is set to `AJL_CONTOUR` (Sections 4.5.3 and 4.6);
2. Collisions result in closed intersection contours between the colliding surfaces. This is more likely to occur if one of the surfaces encloses a volume.

6.11.1 Example: FEM collisions

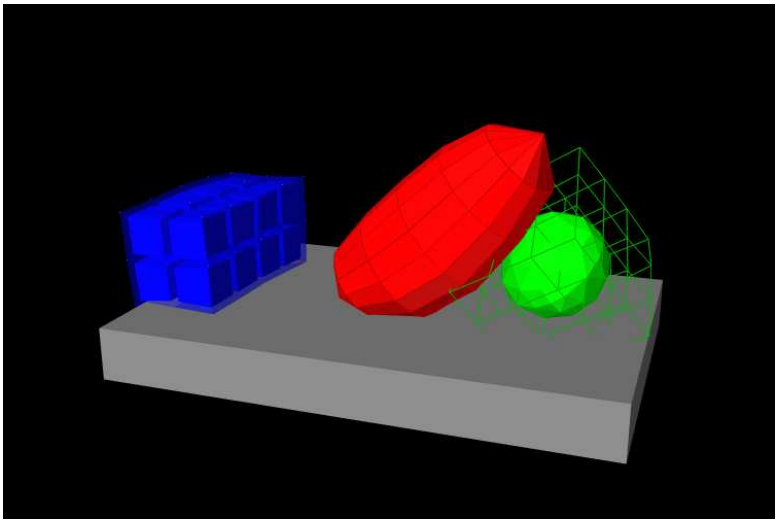


Figure 6.17: FemCollisions model loaded into ArtiSynth.

An example of FEM collisions is shown in Figure 6.17. The full source code can be found in the ArtiSynth repository under `artisynth.demos.tutorial.FemCollisions`. The collision-enabling code is as follows:

```
// Set up collisions
mech.setCollisionBehavior(ellipsoid, beam, true);    // beam-ellipsoid
mech.setCollisionBehavior(ellipsoid, table, true);   // ellipsoid-table
mech.setCollisionBehavior(table, beam, true);        // beam-table

FemMeshComp embeddedSphere = block.getMeshComp("embedded");       // get embedded ↩
    FemMeshComp
mech.setCollisionBehavior(embeddedSphere, table, true);       // sphere-table
mech.setCollisionBehavior(ellipsoid, embeddedSphere, true);   // sphere-ellipsoid
```

Notice in the figure that the surface of the green block passes through the table and ellipsoid; only the embedded sphere has collisions enabled.

6.12 Rendering and Visualizations

In addition to the standard `RenderProps` that control how the nodes and surfaces appear, finite element models and their sub-components have a few additional properties that affect rendering. Some of these are listed in Table 6.7.

Table 6.7: FEM-specific rendering properties

Property	Description
`elementWidgetSize`	size of element to render $\in [0,1]$
`directionRenderLen`	relative length to draw fibre direction indicator $\in [0,1]$
`directionRenderType`	where to draw directions: `ELEMENT`, `INTEGRATION_POINT`
`surfaceRendering`	how to render surface: `None`, `Shaded`, `Stress`, `Strain`, `MAPStress`, `MAPStrain`
`stressPlotRange`	range of values for stress/strain plot
`stressPlotRanging`	how to determine stress/strain plot range: `Auto`, `Fixed`
`colorMap`	delegate object controlling the map of stress/strain values to color

The property `elementWidgetSize` applies only to FemModel3d and FemElement3d. It specifies the scale to draw each element volume. For instance, the blue beam in Figure 6.17 uses a widget size of 0.8, resulting in a mosaic-like pattern.

The next two properties in Table 6.7 apply to the muscle classes FemMuscleModel, MuscleBundle, and MuscleElementDesc. When `directionRenderLen > 0`, lines are drawn inside elements to indicate fibre directions. If `directionRenderType`

= ELEMENT, then one line is drawn per element indicating the average contraction direction. If directionRenderType = INTEGRATION_POINT, a separate direction line is drawn per point.

The last four properties apply to FemModel3d and FemMeshComp. They control how the surface is colored. This can be used to enable stress/strain visualizations. The property surfaceRendering sets what to draw:

None	no surface
Shaded	the face color specified by the mesh's RenderProps
Stress	the von Mises stress
Strain	the von Mises strain
MAPStress	Maximum absolute value principal stress
MAPStrain	Maximum absolute value principal strain

The stressPlotRange controls the range of values to use when plotting stress/strain. Values outside this range are truncated. The colorMap is a delegate object that converts those stress and strain values to colors. Various types of maps exist, including GreyscaleColorMap, HueColorMap, RainbowColorMap, and JetColorMap. These all implement the ColorMap interface.

To display values corresponding to colors, a ColorBar needs to be added to the RootModel. Color bars are general Renderable objects that are only used for visualizations. They are added to the display using the

```
addRenderable (Renderable r);
```

method in RootModel. Color bars also have a ColorMap associated with it. The following functions are useful for controlling its visualization:

```
setNumberFormat ( String fmtStr );    // C-like numeric format specification
populateLabels ( double min, double max, int tick );     // initialize labels
updateLabels ( double min, double max );                 // update existing labels

setColorMap ( ColorMap map );                            // set color map

// Control position/size of the bar
setNormalizedLocation (double x, double y, double width, double height);
setLocationOverride (double x, double y, double width, double height)
```

The normalized location specifies sizes relative to the screen size (1 = screen width/height). The location override, if values are non-zero, will override the normalized location, specifying values in absolute pixels. Negative values for position correspond to distances from the left/top. For instance,

```
setNormalizedLocation(0, 0.1, 0, 0.8);  // set relative positions
setLocationOverride(-40, 0, 20, 0);     // override with pixel lengths
```

will create a bar that is 10% up from the bottom of the screen, 40 pixels from the right edge, with a height occupying 80% of the screen, and width 20 pixels.

Note that the color bar is not associated with any mesh or finite element model. Any synchronization of colors and labels must be done manually by the developer. It is recommended to do this in the RootModel's prerender(...) method, so that colors are updated every time the model's rendering configuration changes.

6.12.1 Example: stress and strain plotting

The following model extends FemBeam to render stress, with an added color bar. The loaded model is shown in Figure 6.18.

```
1 package artisynth.demos.tutorial;
2
3 import java.io.IOException;
4
5 import maspack.render.RenderList;
6 import maspack.util.DoubleInterval;
7 import artisynth.core.femmodels.FemModel.Ranging;
8 import artisynth.core.femmodels.FemModel.SurfaceRender;
9 import artisynth.core.renderables.ColorBar;
```

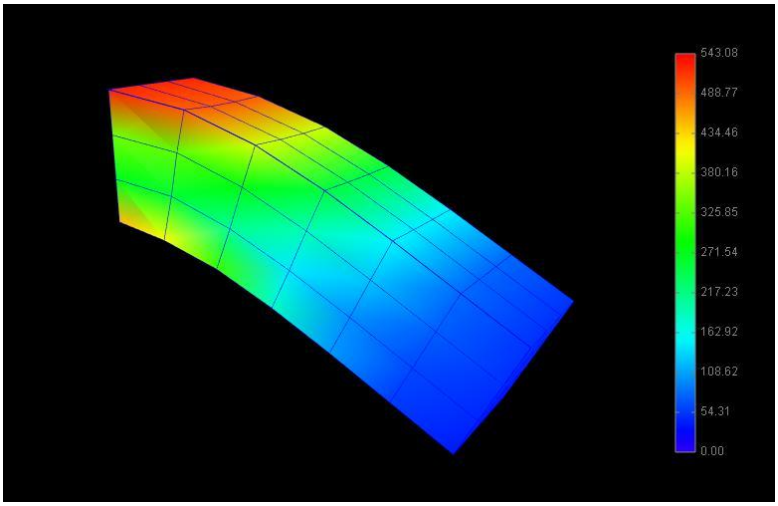


Figure 6.18: FemBeamColored model loaded into ArtiSynth.

```
10
11 public class FemBeamColored extends FemBeam {
12
13    @Override
14    public void build(String[] args) throws IOException {
15       super.build(args);
16
17       // Show stress on the surface
18       fem.setSurfaceRendering(SurfaceRender.Stress);
19       fem.setStressPlotRanging(Ranging.Auto);
20
21       // Create a colorbar
22       ColorBar cbar = new ColorBar();
23       cbar.setName("colorBar");
24       cbar.setNumberFormat("%.2f");      // 2 decimal places
25       cbar.populateLabels(0.0, 1.0, 10); // Start with range [0,1], 10 ticks
26       cbar.setLocation(-100, 0.1, 20, 0.8);
27       addRenderable(cbar);
28
29    }
30
31    @Override
32    public void prerender(RenderList list) {
33       super.prerender(list);
34       // Synchronize color bar/values in case they are changed. Do this *after*
35       // super.prerender(), in case values are changed there.
36       ColorBar cbar = (ColorBar)(renderables().get("colorBar"));
37       cbar.setColorMap(fem.getColorMap());
38       DoubleInterval range = fem.getStressPlotRange();
39       cbar.updateLabels(range.getLowerBound(), range.getUpperBound());
40
41
42    }
43
44 }
```

6.12.2 Example: rendering contact pressures

As mentioned in Section 4.6.5, it is possible to render the contact pressure involved in the collision between two bodies, and this is often particularly desirable in the case of FEM models. A simple example is defined in

`artisynth.demos.tutorial.ContactPressureRender`

which sets the drawColorMap property of the collision behavior to CONTACT_PRESSURE in order to display a color map of the contact pressure. The example is similar to that of Section 4.6.7, which shows how to render penetration depth.

> The caveats about color map rendering described in Section 4.6.5 apply. In particular, the resolution of the color map is limited by the resolution of the collision mesh for the collidable on which the map is drawn, and so the application should ensure that this is sufficiently fine. Also, to allow the map to blend properly with the rest of the collidable, the color corresponding to 0 should match the default face color for the collidable.

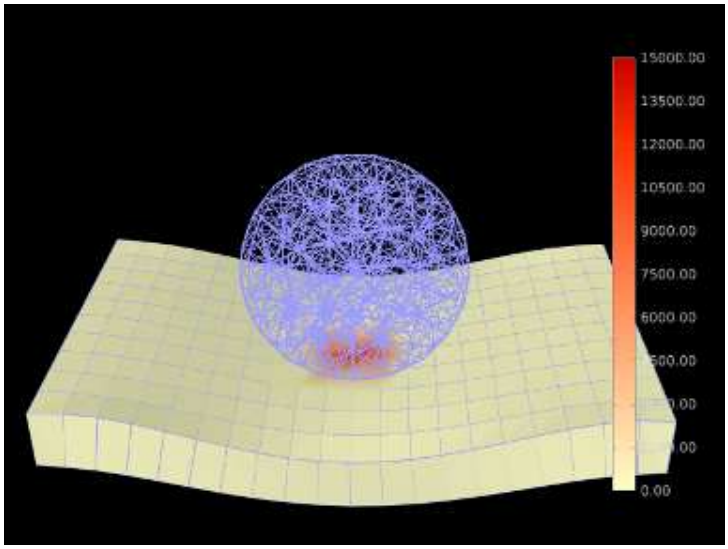


Figure 6.19: ContactPressureRender showing the contact pressure as a spherical FEM model falls onto an FEM sheet. The color map is drawn on the sheet model, with redder values indicating greater pressure.

The complete source code is shown below:

```
package artisynth.demos.tutorial;

import java.awt.Color;

import artisynth.core.femmodels.FemFactory;
import artisynth.core.femmodels.FemModel.SurfaceRender;
import artisynth.core.femmodels.FemModel3d;
import artisynth.core.femmodels.FemNode3d;
import artisynth.core.materials.LinearMaterial;
import artisynth.core.mechmodels.CollisionBehavior;
import artisynth.core.mechmodels.CollisionBehavior.ColorMapType;
import artisynth.core.mechmodels.CollisionManager;
import artisynth.core.mechmodels.MechModel;
import artisynth.core.renderables.ColorBar;
import artisynth.core.util.ScalarRange;
import artisynth.core.workspace.RootModel;
import maspack.matrix.RigidTransform3d;
import maspack.render.RenderList;
import maspack.render.RenderProps;
import maspack.render.color.JetColorMap;

public class ContactPressureRender extends RootModel {

   double density = 1000;
   double EPS = 1e-10;

   // Convenience method for creating colors from [0-255] RGB values
   private static Color createColor (int r, int g, int b) {
      return new Color (r/255.0f, g/255.0f, b/255.0f);
```

```
   }

   private static Color CREAM = createColor (255, 255, 200);
   private static Color BLUE_GRAY = createColor (153, 153, 255);

   // Creates and returns a ColorBar renderable object
   public ColorBar createColorBar() {
      ColorBar cbar = new ColorBar();
      cbar.setName("colorBar");
      cbar.setNumberFormat("%.2f");      // 2 decimal places
      cbar.populateLabels(0.0, 1.0, 10); // Start with range [0,1], 10 ticks
      cbar.setLocation(-100, 0.1, 20, 0.8);
      cbar.setTextColor (Color.WHITE);
      addRenderable(cbar);               // add to root model's renderables
      return cbar;
   }

   public void build (String[] args) {
      MechModel mech = new MechModel ("mech");
      addModel (mech);

      // create FEM ball
      FemModel3d ball = new FemModel3d("ball");
      ball.setDensity (density);
      FemFactory.createIcosahedralSphere (ball, /*radius=*/0.1, /*ndivs=*/2, 1);
      ball.setMaterial (new LinearMaterial (100000, 0.4));
      mech.addModel (ball);

      // create FEM sheet
      FemModel3d sheet = new FemModel3d("sheet");
      sheet.setDensity (density);
      FemFactory.createHexGrid (
         sheet, /*wx*/0.5, /*wy*/0.3, /*wz*/0.05, /*nx*/20, /*ny*/10, /*nz*/1);
      sheet.transformGeometry (new RigidTransform3d (0, 0, -0.2));
      sheet.setMaterial (new LinearMaterial (500000, 0.4));
      sheet.setSurfaceRendering (SurfaceRender.Shaded);
      mech.addModel (sheet);

      // fix the side nodes of the surface
      for (FemNode3d n : sheet.getNodes()) {
         double x = n.getPosition().x;
         if (Math.abs(x-(-0.25)) <= EPS || Math.abs(x-(0.25)) <= EPS) {
            n.setDynamic (false);
         }
      }

      // create and set a collision behavior between the ball and surface.
      CollisionBehavior behav = new CollisionBehavior (true, 0);
      behav.setDrawColorMap (ColorMapType.CONTACT_PRESSURE);
      behav.setColorMapCollidable (1); // show color map on collidable 1 (sheet);
      behav.setColorMapRange (new ScalarRange(0, 15000.0));
      mech.setCollisionBehavior (ball, sheet, behav);

      CollisionManager cm = mech.getCollisionManager();
      // set rendering properties in the collision manager:
      RenderProps.setVisible (cm, true);    // enable collision rendering
      // create a custom color map for rendering the penetration depth
      JetColorMap map = new JetColorMap();
      map.setColorArray (
         new Color[] {
            CREAM,                         // no penetration
            createColor (255, 204, 153),
            createColor (255, 153, 102),
            createColor (255, 102, 51),
            createColor (255, 51, 0),
```

```
95              createColor (204, 0, 0),     // most penetration
96           });
97        cm.setColorMap (map);
98
99        // create a separate color bar to show color map pressure values
100       ColorBar cbar = createColorBar();
101       cbar.updateLabels(0, 15000);
102       cbar.setColorMap (map);
103
104       // set color for all bodies
105       RenderProps.setFaceColor (mech, CREAM);
106       RenderProps.setLineColor (mech, BLUE_GRAY);
107    }
108 }
```

To begin, the demo creates two FEM models: a spherical ball (lines 52-56) and a rectangular sheet (lines 59-66), and then fixes the end nodes of the sheet (lines 69-74). Surface rendering is enabled for the sheet (line 65), but not for the ball, in order to improve the visibility of the color map.

Lines 77-81 create and set a collision behavior between the two models, with the drawColorMap property set to `CONTACT_PRESSURE`. Because for this example we want the color map to be drawn on the *second* collidable (the sheet), we set the setColorMapCollidable property to 1 (line 79); otherwise, the default value of 0 would cause the color map to be drawn on the first collidable (the ball). (Alternatively, we could have simply defined the collision behavior as being between the surface and the ball instead of the ball and the sheet.) The color map range is explicitly set to lie between $[0, 15000]$ (line 80); this is in contrast to the example in Section 4.6.7, where the range is auto-updated on each step. The color range is also set explicitly in the behavior, but if multiple objects were colliding it would likely be preferable to set it in the collision manager (with the behavior's value left as `null`) to ensure a uniform render range across all collisions. Other rendering properties are set for the collision manager at lines 83-97, including a custom color map that varies between `CREAM` (the color of the mesh) for no pressure and dark red for maximum pressure.

At line 100, a color bar is created and added to the scene, using the method `createColorBar()` (lines 36-45), to explicitly show the pressure that corresponds to the different colors. The color bar is given the same color map and value range used to render the pressure. Finally, default face and line colors for all components in the model are set at lines 105-106.

To run this example in ArtiSynth, select All demos > tutorial > ContactPressureRender from the Models menu. When run, the FEM models will collide and render the contact pressure on the sheet, as shown in Figure 6.19.

Chapter 7

Muscle Wrapping and Via Points

ArtiSynth provides support for multipoint springs and muscles, which are similar to axial springs and muscles (Sections 3.1.1 and 4.4), except that they can contain multiple via points and also wrap around obstacles. This allows the associated force directions to vary in response to obstacles and constraints in the model, which is particularly important in biomechanical models where point-to-point muscles need to wrap around anatomical structures such as bones. A schematic illustration is shown in Figure 7.1, where a single spring connects points $\mathbf{p}_0$ and $\mathbf{p}_2$, while passing through a single via point $\mathbf{p}_1$ and wrapping around obstacles W_1 and W_2. Figure 7.2 shows two examples involving a rigid body with fixed via points and a spring wrapping around three rigid bodies.

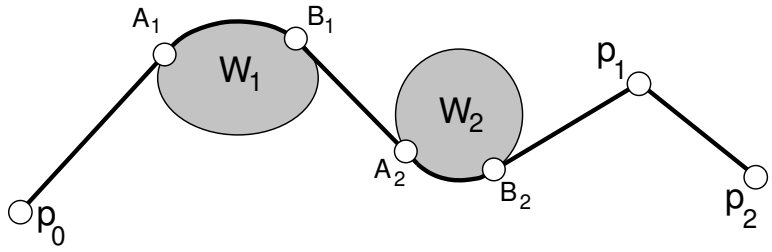


Figure 7.1: Schematic illustration of a multipoint spring passing through a via point $\mathbf{p}_1$ and wrapping around two obstacles W_1 and W_2. The points A_1, B_1 and A_2, B_2 denote the first and last locations where W_1 and W_2 make contact with the spring.

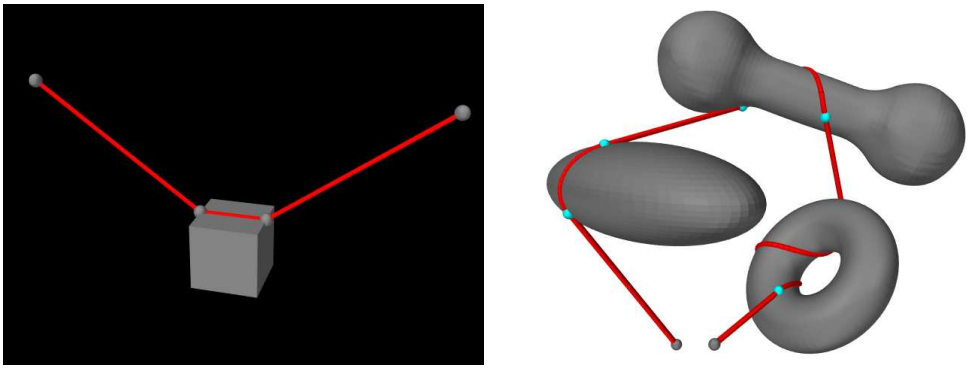


Figure 7.2: Left: A multipoint spring with two via points rigidly fixed to a box-shaped rigid body. Right: A multipoint spring wrapped around three obstacles.

As with axial springs and muscles, multipoint springs and muscles must have two points to denote their beginning and end. In between, they can have any number of *via* points, which are fixed locations which the spring must pass through in the specified order. Any ArtiSynth Point object may be specified as a via point, including particles and markers. The purpose of the via point is generally to direct the spring along some particular path. In particular, the path directions

before and after a via point will generally be different, and forces acting on the via point will be determined by the tension in the spring (or muscle) acting along these two different directions.

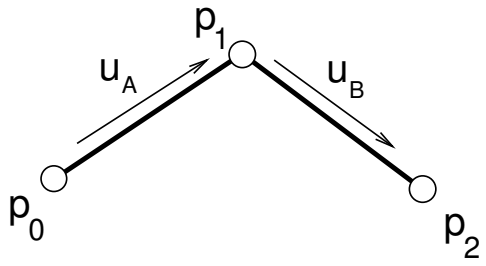


Figure 7.3: A multipoint spring with a single via point $\mathbf{p}_1$, showing the unit direction vectors $\mathbf{u}_A$ and $\mathbf{u}_B$ immediately before and after.

> Conceptually, the spring or muscle "slides" through its via points, which act analogously to virtual three dimensional pulleys. In particular, the proportional distance between via points does *not* remain fixed.

The tension f within the spring or muscle is computed from its material, using the relation $f(l,\dot{l},a)$ described in Sections 3.1.1 and 4.4.1, where l now denotes the *entire* length of the spring as it passes through the via points and wraps around obstacles. The total force $\mathbf{f}$ acting on each via point is then given by

$$\mathbf{f} = f \cdot (\mathbf{u}_B - \mathbf{u}_A)$$

where $\mathbf{u}_B$ and $\mathbf{u}_A$ are unit vectors giving the spring's direction immediately after and before the via point (Figure 7.3).

Multipoint springs can also be made to wrap around one or more *wrappable* objects. Unlike via points, wrappable objects can occur in any order along the spring and wrapping only occurs when the spring and the object actually collide. Any ArtiSynth object that implements Wrappable can be used as a wrapping object (currently, only RigidBody objects implement `Wrappable`). The forces acting on a wrappable are those generated by the forces $\mathbf{f}_A$ and $\mathbf{f}_B$ acting on the points A and B where the spring makes and leaves contact with the it (Figure 7.4). These forces are given by

$$\mathbf{f}_A = -f\mathbf{u}_A, \quad \mathbf{f}_B = f\mathbf{u}_B,$$

where $\mathbf{u}_B$ are $\mathbf{u}_A$ are unit vectors giving the spring's direction immediately before A and after B. Points A and B are collectively known as the A/B points.

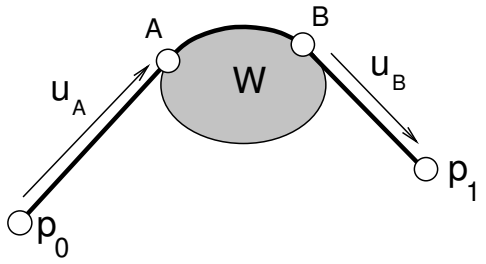


Figure 7.4: A multipoint spring wrapping around a single obstacle W, with initial and final contact at points A and B, and associated unit direction vectors $\mathbf{u}_A$ and $\mathbf{u}_B$.

7.1 Via Points

Multipoint springs and muscles are implemented by the classes MultiPointSpring and MultiPointMuscle, respectively. The relationship between MultiPointSpring and MultiPointMuscle is the same as that between AxialSpring and Muscle: The latter is a subclass of the former, and allows the creation of active tension forces in response to its excitation property.

An application allocates one of these components, sets the appropriate material properties for the tension forces, and then adds points and wrappable objects as desired.

Points can be added, queried, and removed using the methods

```
void addPoint (Point pnt)
Point getPoint (int idx)
int numPoints()
boolean removePoint (Point pnt)
```

As with AxialSpring, there must be at least two points anchoring the beginning and end of the spring. Any additional points will be *via points*.

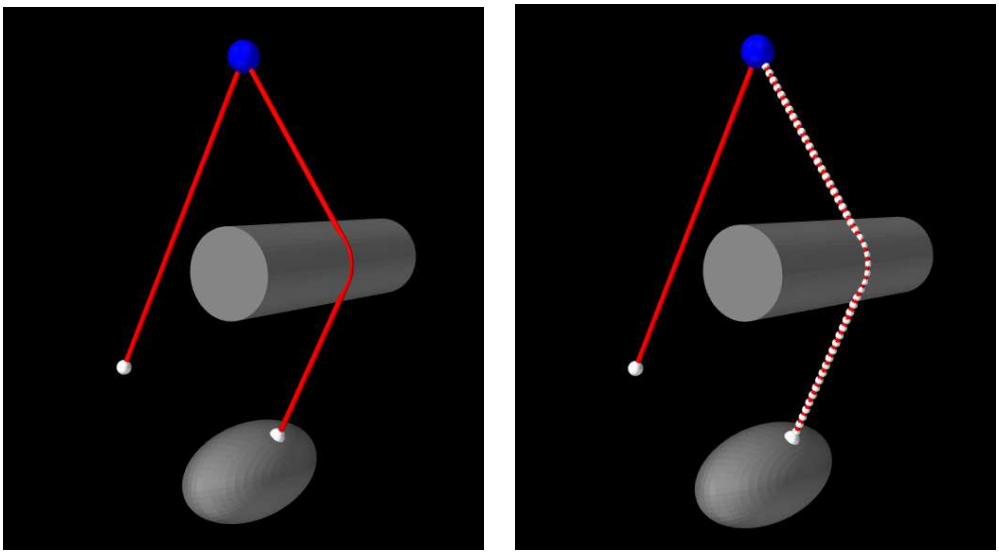


Figure 7.5: A multipoint spring with two segments, separated by a blue via point (top), with the rightmost segment set to be wrappable so that it can wrap around a cylinder. The right image shows the wrappable segment's knots.

The section of a multipoint spring between any two adjacent points is known as a *segment*. By default, each segment forms a straight line between the two points and does *not* interact with any wrappable obstacles. To interact with wrappables, a segment needs to be declared *wrappable*, as described in Section 7.2.

Spring construction is illustrated by the following code fragment:

```
MultiPoint spring = new MultiPointSpring();
spring.setMaterial (new LinearAxialMaterial (stiffness, damping));
spring.addPoint (p0); // start point
spring.addPoint (p1); // via point
spring.addPoint (p2); // via point
spring.addPoint (p3); // stop point
```

This creates a new `MultiPointSpring` and sets its material to a simple linear material with a specified stiffness and damping. Four points `p0`, `p1`, `p2`, `p3` are then added, forming a start point, two via points, and a stop point.

7.1.1 Example: a muscle with via points

A simple example of a muscle containing via points is given by `artisynth.demos.tutorial.ViaPointMuscle`. It consists of a MultiPointMuscle passing through two via points attached to a block. The code is given below:

```
1 package artisynth.demos.tutorial;
2
3 import java.awt.Color;
4
5 import artisynth.core.gui.ControlPanel;
6 import artisynth.core.materials.SimpleAxialMuscle;
```

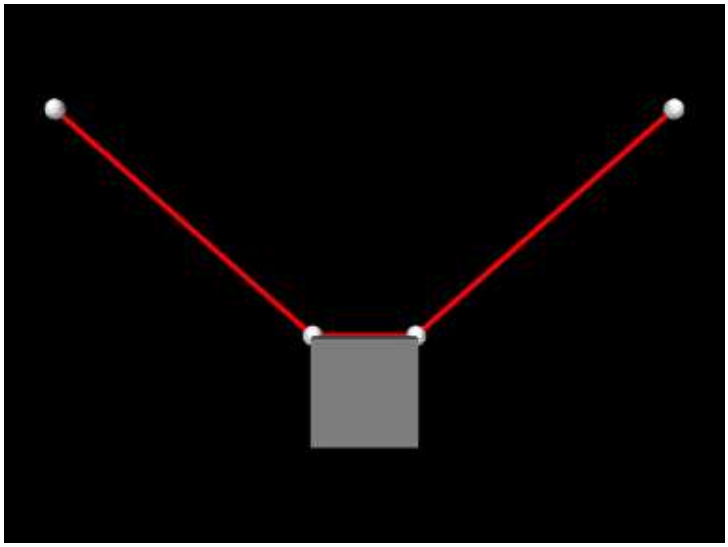


Figure 7.6: ViaPointMuscle model loaded into ArtiSynth.

```
7  import artisynth.core.mechmodels.FrameMarker;
8  import artisynth.core.mechmodels.MechModel;
9  import artisynth.core.mechmodels.MultiPointMuscle;
10 import artisynth.core.mechmodels.Particle;
11 import artisynth.core.mechmodels.RigidBody;
12 import artisynth.core.workspace.RootModel;
13 import maspack.matrix.Point3d;
14 import maspack.render.RenderProps;
15
16 public class ViaPointMuscle extends RootModel {
17
18    protected static double size = 1.0;
19
20    public void build (String[] args) {
21       MechModel mech = new MechModel ("mech");
22       addModel (mech);
23
24       mech.setFrameDamping (1.0); // set damping parameters
25       mech.setRotaryDamping (0.1);
26
27       // create block to which muscle will be attached
28       RigidBody block = RigidBody.createBox (
29          "block", /*widths=*/1.0, 1.0, 1.0, /*density=*/1.0);
30       mech.addRigidBody (block);
31
32       // create muscle start and end points
33       Particle p0 = new Particle (/*mass=*/0.1, /*x,y,z=*/-3.0, 0, 0.5);
34       p0.setDynamic (false);
35       mech.addParticle (p0);
36       Particle p1 = new Particle (/*mass=*/0.1, /*x,y,z=*/3.0, 0, 0.5);
37       p1.setDynamic (false);
38       mech.addParticle (p1);
39
40       // create markers to serve as via points
41       FrameMarker via0 = new FrameMarker();
42       mech.addFrameMarker (via0, block, new Point3d (-0.5, 0, 0.5));
43       FrameMarker via1 = new FrameMarker();
44       mech.addFrameMarker (via1, block, new Point3d (0.5, 0, 0.5));
45
46       // create muscle, set material, and add points
47       MultiPointMuscle muscle = new MultiPointMuscle ();
```

```
48        muscle.setMaterial (new SimpleAxialMuscle (/*k=*/1, /*d=*/0, /*maxf=*/10));
49        muscle.addPoint (p0);
50        muscle.addPoint (via0);
51        muscle.addPoint (via1);
52        muscle.addPoint (p1);
53        mech.addMultiPointSpring (muscle);
54
55        // set render properties
56        RenderProps.setSphericalPoints (mech, 0.1, Color.WHITE);
57        RenderProps.setCylindricalLines (mech, 0.03, Color.RED);
58
59        createControlPanel();
60     }
61
62     private void createControlPanel() {
63        // creates a panel to adjust the muscle excitation
64        ControlPanel panel = new ControlPanel ("options", "");
65        panel.addWidget (this, "models/mech/multiPointSprings/0:excitation");
66        addControlPanel (panel);
67     }
68  }
```

Lines 21-30 of the `build()` method create a `MechModel` and add a simple rigid body block to it. Two non-dynamic points (`p0` and `p1`) are then created to act as muscle end points (lines 33-38), along with two markers (`via0` and `via1`) which are attached to the block to act as via points (lines 41-44). The muscle itself is created by lines 42-53, with the end points and via points being added in order from start to end. The muscle material is a SimpleAxialMuscle, which computes tension according to the simple linear formula (4.1) described in Section 4.4.1. Lines 56-57 set render properties for the model, and line 59 creates a control panel (Section 5.1) that allows the muscle excitation property to be interactively controlled.

To run this example in ArtiSynth, select All demos > tutorial > ViaPointMuscle from the Models menu. The model should load and initially appear as in Figure 7.6. Running the model will cause the block to fall and swing about under gravity, while changing the muscle's excitation in the control panel will vary its tension.

7.2 Obstacle Wrapping

As mentioned in Section 7.1, segments between pairs of via points can be declared *wrappable*, allowing them to interact with wrappable obstacles. This can be done as via points are added to the spring, using the methods

```
void setSegmentWrappable (int numKnots)
void setSegmentWrappable (int numKnots, Point3d[] initialPoints)
```

These make *wrappable* the next segment to be created (i.e., the segment between the most recently added point and the next point to be added), with `numKnots` specifying the number of *knots* that should be used to implement the wrapping. Knots are points that divide the wrappable segment into a piecewise linear curve, and are used to check for collisions with the wrapping surfaces (Figure 7.5). The argument `initialPoints` used by the second method is an optional argument which, if non-null, can be used to specify intermediate guide points to give the segment an initial path around around any obstacles (for more details, see Section 7.4).

Each wrappable segment will be capable of colliding with any of the wrappable obstacles that are known to the spring. Wrappables can be added, queried and removed using the following methods:

```
void addWrappable (Wrappable wrappable)
Wrappable getWrappable (int idx)
int numWrappables()
boolean removeWrappable (Wrappable wrappable)
```

Unlike points, however, there is no implied ordering and wrappables can be added in any order and at any time during the spring's construction.

Wrappable spring construction is illustrated by the following code fragment:

```
MultiPoint spring = new MultiPointSpring();
spring.setMaterial (new LinearAxialMaterial (stiffness, damping));
spring.addPoint (p0); // start point
spring.setSegmentWrappable (50);  // wrappable segment
spring.addPoint (p1); // via point
spring.addPoint (p2); // end point
spring.addWrappable (wrappable1);
spring.addWrappable (wrappable2);
spring.updateWrapSegments(); // ``shrink wrap'' spring to the obstacles
```

This creates a new `MultiPointSpring` with a linear material and three points `p0`, `p1`, and `p2`, forming a start point, via point, and stop point. The segment between `p0` and `p1` is set to be wrappable with 50 knot points. Two wrappable obstacles are added next, each of which will interact with the `p0`-`p1` segment, but *not* with the non-wrappable `p1`-`p2` segment. Finally, `updateWrapSegments()` is called to do an initial solve for the wrapping segments, so that they will be "pulled tight" around any obstacles before simulation begins.

It is also possible to make a segment wrappable *after* spring construction, using the method

```
void setSegmentWrappable (int segIdx, int numKnots, Point3d[] initialPoints)
```

where `segIdx` identifies the segment between points `segIdx` and `segIdx` + 1.

How many knots should be specified for a wrappable segment? Enough so that the resulting piecewise-linear approximation to the wrapping curve is sufficiently "smooth", and also enough to adequately detect contact with the obstacles without passing through them. Values between 50 and 100 generally give good results. Obstacles that are small with respect to the segment length may necessitate more knots. Making the number of knots very large will slow down the computation (although the computational cost is only $O(n)$ with respect to the number of knots).

At the time of this writing, ArtiSynth implements two types of Wrappable object, both of which are instances of RigidBody. The first are specialized *analytic* subclasses of `RigidBody`, listed in Table 7.1, which define specific geometries and use analytic methods for the collision handling with the knot points. The use of analytic methods allows for greater accuracy and (possibly) computational efficiency, and so because of this, these special geometry wrappables should be used whenever possible.

Wrappable	Description
RigidCylinder	A cylinder with a specified height and radius
RigidSphere	A sphere with a specified radius
RigidEllipsoid	An ellipsoid with specified semi-axis lengths
RigidTorus	A torus with specified inner and outer radii

Table 7.1: Specialized analytic subclasses of `RigidBody`

The second are general rigid bodies which are *not* analytic subclasses, and for which the wrapping surface is determined directly from the geometry of its collision mesh returned by getCollisionMesh(). (Typically the collision mesh corresponds to the surface mesh, but it is possible to specify alternates; see Section 3.2.8.) This is useful in that it permits wrapping around *arbitrary* mesh geometries (Figure 7.7), but in order for the wrapping to work well, these geometries should be smooth, without sharp edges or corners. Wrapping around general meshes is implemented using a quadratically interpolated signed-distance grid (Section 4.7), and the resolution of this grid also affects the effective smoothness of the wrapping surface. More details on this are given in Section 7.3.

7.2.1 Example: wrapping around a cylinder

A example showing multipoint spring wrapping is given by `artisynth.demos.tutorial.CylinderWrapping`. It consists of a `MultiPointSpring` passing through a single via point, with both segments on either side of the point made wrappable. Two analytic wrappables are used: a fixed `RigidCylinder`, and a moving `RigidEllipsoid` attached to the end of the spring. The code, excluding include directives, is given below:

```
1 public class CylinderWrapping extends RootModel {
2
3    public void build (String[] args) {
4      MechModel mech = new MechModel ("mech");
```

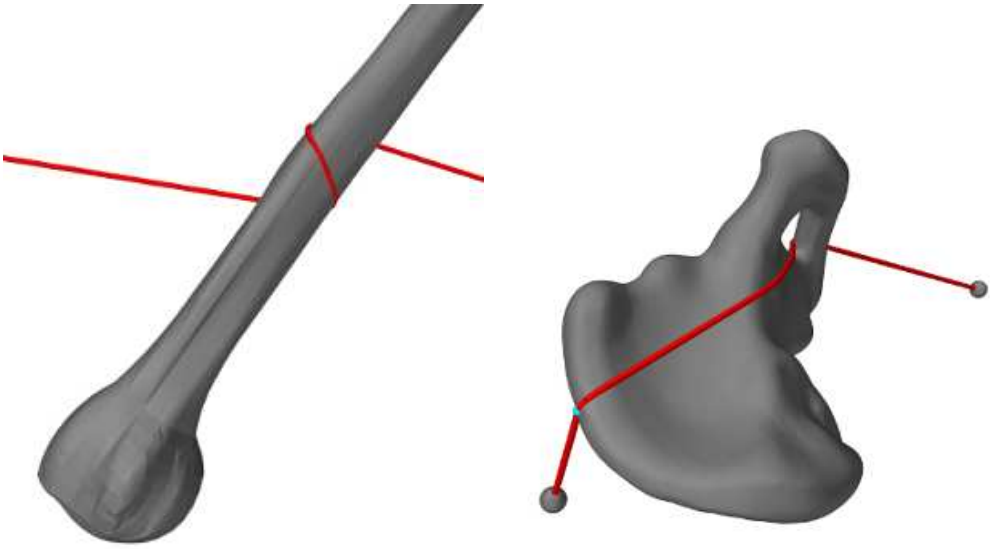


Figure 7.7: Muscle strands wrapped around general bone-shaped meshes: a humerus (left), and a pelvis (right)

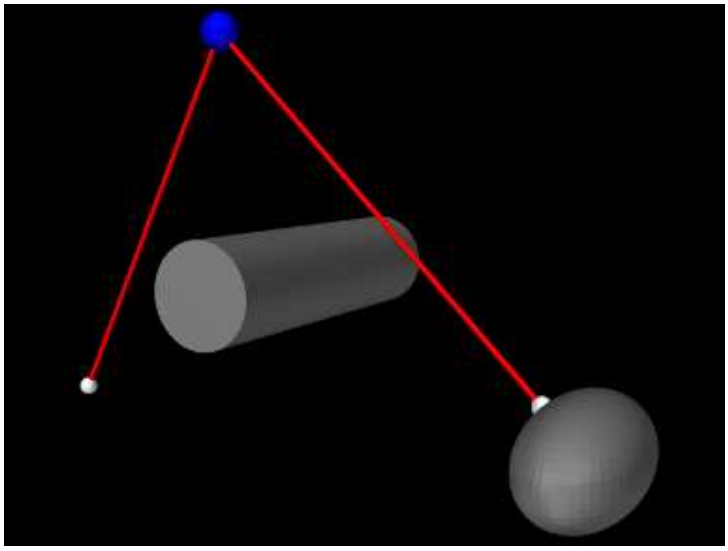


Figure 7.8: CylinderWrapping model loaded into ArtiSynth.

```
5       addModel (mech);
6
7       mech.setFrameDamping (100.0); // set damping parameters
8       mech.setRotaryDamping (10.0);
9
10      double density = 150;
11
12      Particle via0 = new Particle (/*mass=*/0, /*x,y,z=*/-1.0, 0.0, 4.0);
13      via0.setDynamic (false);
14      mech.addParticle (via0);
15      Particle p1 = new Particle (/*mass=*/0, /*x,y,z=*/-3.0, 0.0, 0.0);
16      p1.setDynamic (false);
17      mech.addParticle (p1);
18
19      // create cylindrical wrapping object
20      RigidCylinder cylinder = new RigidCylinder (
21         "cylinder", /*rad=*/0.5, /*height=*/3.5, density, /*nsides=*/50);
22      cylinder.setPose (new RigidTransform3d (0, 0, 1.5, 0, 0, Math.PI/2));
23      cylinder.setDynamic (false);
```

```
24        mech.addRigidBody (cylinder);
25
26        // create ellipsoidal wrapping object
27        double rad = 0.6;
28        RigidEllipsoid ellipsoid = new RigidEllipsoid (
29           "ellipsoid", /*a,b,c=*/rad, 2*rad, rad, density, /*nslices=*/50);
30        ellipsoid.transformGeometry (new RigidTransform3d (3, 0, 0));
31        mech.addRigidBody (ellipsoid);
32
33        // attach a marker to the ellipsoid
34        FrameMarker p0 = new FrameMarker ();
35        double halfRoot2 = Math.sqrt(2)/2;
36        mech.addFrameMarker (
37           p0, ellipsoid, new Point3d (-rad*halfRoot2, 0, rad*halfRoot2));
38
39        // enable collisions between the ellipsoid and cylinder
40        mech.setCollisionBehavior (cylinder, ellipsoid, true);
41
42        // create the spring, making both segments wrappable with 50 knots
43        MultiPointSpring spring = new MultiPointSpring ("spring", 300, 1.0, 0);
44        spring.addPoint (p0);
45        spring.setSegmentWrappable (50);
46        spring.addPoint (via0);
47        spring.setSegmentWrappable (50);
48        spring.addPoint (p1);
49        spring.addWrappable (cylinder);
50        spring.addWrappable (ellipsoid);
51        mech.addMultiPointSpring (spring);
52
53        // set various rendering properties
54        RenderProps.setSphericalPoints (mech, 0.1, Color.WHITE);
55        RenderProps.setSphericalPoints (p1, 0.2, Color.BLUE);
56        RenderProps.setSphericalPoints (spring, 0.1, Color.GRAY);
57        RenderProps.setCylindricalLines (spring, 0.03, Color.RED);
58
59        createControlPanel (spring);
60     }
61
62     private void createControlPanel (MultiPointSpring spring) {
63        ControlPanel panel = new ControlPanel ("options", "");
64        // creates a panel to control knot and A/B point visibility
65        panel.addWidget (spring, "drawKnots");
66        panel.addWidget (spring, "drawABPoints");
67        addControlPanel (panel);
68     }
69  }
```

Lines 4-17 of the `build()` method create a `MechModel` with two fixed particles `via0` and `p1` to be used as via and stop points. Next, two analytic wrappables are created: a `RigidCylinder` and a `RigidEllipsoid`, with the former fixed in place and the latter connected to the start of the spring via the marker `p0` (lines 20-37). Collisions are enabled between these two wrappables at line 40. The spring itself is created (lines 44-52), using `setSegmentWrappable()` to make the segments (`p0`, `via0`) and (`via0`, `p1`) wrappable with 50 knots each, and `addWrappable()` to make it aware of the two wrappables. Finally, render properties at set (lines 55-58), and a control panel (Section 5.1) is added that allows the spring's drawKnots and drawABPoints properties to be interactively set.

To run this example in ArtiSynth, select All demos > tutorial > CylinderWrapping from the Models menu. The model should load and initially appear as in Figure 7.8. Running the model will cause the ellipsoid to fall and the spring to wrap around the cylinder. Using the pull tool (Section "Pull Manipulation" in the ArtiSynth User Interface Guide) on the ellipsoid can cause additional motions and make it also collide with the spring. Selecting drawKnots or drawABPoints in the control panel will cause the spring to render its knots and/or A/B points.

7.3 General Surfaces and Distance Grids

As mentioned in Section 7.2, wrapping around general mesh geometries is implemented using a quadratically interpolated signed distance grid. By default, for a rigid body, this grid is generated automatically from the body's collision mesh (as returned by `getCollisionMesh()`; see Section 4.6.2).

Using a distance grid allows very efficient collision handling between the body and the wrap segment knots. However, it also means that the true wrapping surface is not actually the collision mesh itself, but instead the zero-valued isosurface associated with quadratic grid interpolation. Well-behaved wrapping behavior requires that this isosurface be smooth and free of sharp edges, so that knot motions remain relatively smooth as they move across it. Quadratic interpolation helps with this, which is the reason for employing it. Otherwise, one should try to ensure that (a) the collision mesh from which the grid is generated is itself smooth and free of sharp edges, and (b) the grid has sufficient resolution to not introduce discretization artifacts.

Muscle wrapping is often performed around structures such as bones, for which the representing surface mesh is often insufficiently smooth (especially if segmented from medical image data). In some cases, the distance grid's quadratic interpolation may provide sufficient smoothing on its own; to determine this, one should examine the quadratic isosurface as described below. In other cases, it may be necessary to explicitly smooth the mesh itself, either externally or within ArtiSynth using the LaplacianSmoother class, which can apply iterations of either Laplacian or volume-preserving Taubin smoothing, via the method

```
LaplacianSmoother.smooth (mesh, numi, lam, mu);
```

Here `numi` is the number of iterations and `tau` and `mu` are the Taubin parameters. Setting `lam` $= 1$ and `mu` $= 0$ results in traditional Laplacian smoothing. If this causes the mesh to shrink more than desired, one can counter this by setting `tau` and `mu` to values used for Taubin smoothing, as described in [18].

> If the mesh is large (i.e., has many vertices), then smoothing it may take noticeable computational time. In such cases, it is generally best to simply save and reuse the smoothed mesh.

By default, if a rigid body contains only one polygonal mesh, then its surface and collision meshes (returned by getSurfaceMesh() and getCollisionMesh(), respectively) are the same. However, if it is necessary to significantly smooth or modify the collision mesh, for wrapping or other purposes, it may be desirable to use different meshes for the surface and collision. This can be done by making the surface mesh non-collidable and adding an additional mesh that *is* collidable, as discussed in Section 3.2.8 as illustrated by the following code fragment:

```
PolygonalMesh surfaceMesh;
PolygonalMesh wrappingMesh;

// ... initialize surface and wrapping meshes ...

// create the body from the surface mesh
RigidBody body = RigidBody.createFromMesh (
    "body", mesh, /*density=*/1000, /*scale=*/1.0);

// set the surface mesh to be non-collidable, and add the wrapping mesh as
// collidable but not having mass
body.getSurfaceMeshComp().setIsCollidable (false);
RigidMeshComp wcomp = body.addMesh (
    wrappingMesh, /*hasMass=*/false, /*collidable=*/true);
RenderProps.setVisible (wcomp, false); // hide the wrapping mesh
```

Here, to ensure that the wrapping mesh does *not* to contribute to the body's inertia, its hasMass property is set to false.

Although it is possible to specify a collision mesh that is separate from the surface mesh, there is currently no way to specify *separate* collision meshes for wrapping and collision handling. If this is desired for some reason, then one alternative is to create a separate body for wrapping purposes, and then attach it to the main body, as described in Section 7.5.

To verify that the distance grid's quadratic isosurface is sufficiently smooth for wrapping purposes, it is useful to visualize the both distance grid and its isosurface directly, and if necessary adjust the resolution used to generate the grid. This can be accomplished using the body's DistanceGridComp, which is a subcomponent named `distanceGrid` and which may be obtained using the method

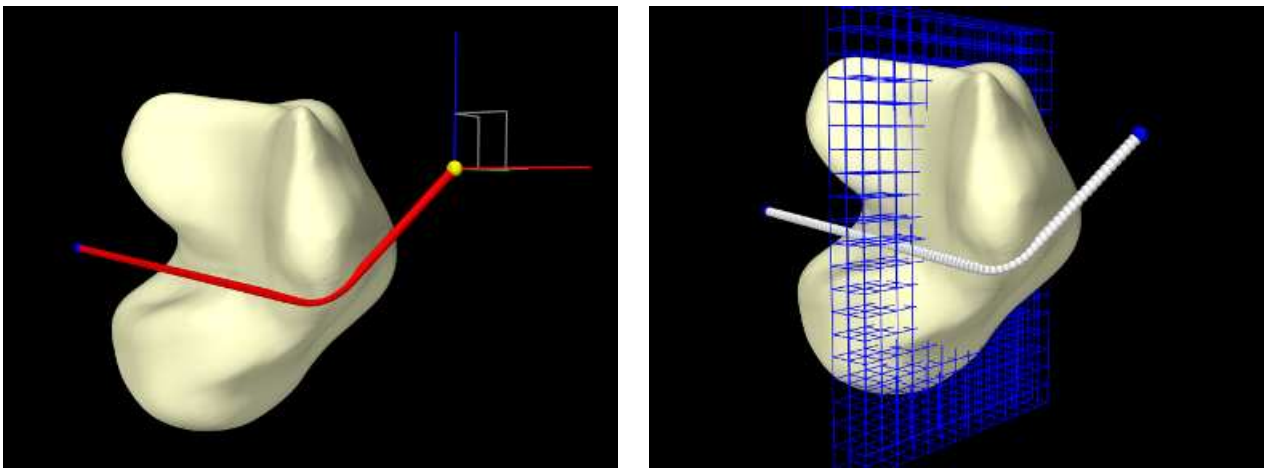


Figure 7.9: TalusWrapping model, with a dragger being used to move `p0` (left), and the knots visible and grid visible with restricted range (right).

```
DistanceGridComp getDistanceGridComp()
```

A `DistanceGridComp` exports a number of properties that can be used to control the grid's visualization, resolution, and fit around the collision mesh. These properties are described in detail in Section 4.7, and can be set either in code using their set/get accessors, or interactively using custom control panels or by selecting the grid component in the GUI and choosing Edit properties ... from the right-click context menu.

> When rendering the mesh isosurface, it is usually desirable to also disable rendering of the collision meshes within the rigid body. For convenience, this can be accomplished by setting the body's gridSurfaceRendering property to `true`, which will cause the grid isosurface to be rendered *instead* of the body's meshes. The isosurface type will be that indicated by the grid component's surfaceType property (which should be `QUADRATIC` for the quadratic isosurface), and the rendering will occur independently of the visibility settings for the meshes or the grid component.

7.3.1 Example: wrapping around a bone

An example of wrapping around a general mesh is given by `artisynth.demos.tutorial.TalusWrapping`. It consists of a MultiPointSpring anchored by two via points and wrapped around a rigid body representing a talus bone. The code, with include directives omitted, is given below:

```
public class TalusWrapping extends RootModel {

   private static Color BONE = new Color (1f, 1f, 0.8f);
   private static double DTOR = Math.PI/180.0;

   public void build (String[] args) {

      MechModel mech = new MechModel ("mech");
      addModel (mech);

      // read in the talus bone mesh
      PolygonalMesh mesh = null;
      try {
         mesh = new PolygonalMesh (
            PathFinder.findSourceDir(this) + "/data/TalusBone.obj");
      }
      catch (Exception e) {
         System.out.println ("Error reading mesh:" + e);
      }
```

```
20        // smooth the mesh using 20 iterations of regular Laplacian smoothing
21        LaplacianSmoother.smooth (mesh, /*count=*/20, /*lambda=*/1, /*mu=*/0);
22        // create the talus body from the mesh
23        RigidBody talus = RigidBody.createFromMesh (
24           "talus", mesh, /*density=*/1000, /*scale=*/1.0);
25        mech.addRigidBody (talus);
26        talus.setDynamic (false);
27        RenderProps.setFaceColor (talus, BONE);
28
29        // create start and end points for the spring
30        Particle p0 = new Particle (/*mass=*/0, /*x,y,z=*/2, 0, 0);
31        p0.setDynamic (false);
32        mech.addParticle (p0);
33        Particle p1 = new Particle (/*mass=*/0, /*x,y,z=*/-2, 0, 0);
34        p1.setDynamic (false);
35        mech.addParticle (p1);
36
37        // create a wrappable spring using a SimpleAxialMuscle material
38        MultiPointSpring spring = new MultiPointSpring ("spring");
39        spring.setMaterial (
40           new SimpleAxialMuscle (/*k=*/0.5, /*d=*/0, /*maxf=*/0.04));
41        spring.addPoint (p0);
42        // add an initial point to the wrappable segment to make sure it wraps
43        // around the bone the right way
44        spring.setSegmentWrappable (
45           100, new Point3d[] { new Point3d (0.0, -1.0, 0.0) });
46        spring.addPoint (p1);
47        spring.addWrappable (talus);
48        spring.updateWrapSegments(); // update the wrapping path
49        mech.addMultiPointSpring (spring);
50
51        // set render properties
52        DistanceGridComp gcomp = talus.getDistanceGridComp();
53        RenderProps.setSphericalPoints (mech, 0.05, Color.BLUE); // points
54        RenderProps.setLineWidth (gcomp, 0); // normal rendering off
55        RenderProps.setCylindricalLines (spring, 0.03, Color.RED); // spring
56        RenderProps.setSphericalPoints (spring, 0.05, Color.WHITE); // knots
57
58        // create a control panel for interactive control
59        ControlPanel panel = new ControlPanel();
60        panel.addWidget (talus, "gridSurfaceRendering");
61        panel.addWidget (gcomp, "resolution");
62        panel.addWidget (gcomp, "maxResolution");
63        panel.addWidget (gcomp, "renderGrid");
64        panel.addWidget (gcomp, "renderRanges");
65        panel.addWidget (spring, "drawKnots");
66        panel.addWidget (spring, "wrapDamping");
67        addControlPanel (panel);
68     }
69  }
```

The mesh describing the talus bone is loaded from the file `"data/TalusBone.obj"` located beneath the model's source directory (lines 11-19), with the utility class PathFinder used to determine the file path (Section 2.6). To ensure better wrapping behavior, the mesh is smoothed using Laplacian smoothing (line 21) before being used to create the rigid body (lines 23-27). The spring and its anchor points `p0` and `p1` are created between lines 30-49, with the talus added as a wrappable. The spring contains a single segment which is made wrappable using 100 knots, and initialized with an intermediate point (line 45) to ensure that it wraps around the bone in the correct way. Intermediate points are described in more detail in Section 7.4.

Render properties are set at lines 52-56; this includes turning off rendering for grid normals by zeroing the lineWidth render property for the grid component.

Finally, lines 59-67 create a control panel (Section 5.1) for interactively controlling a variety of properties, including gridSurfaceRendering for the talus (to see the grid isosurface instead of the bone mesh), resolution, maxResolution,

renderGrid, and renderRanges for the grid component (to control its resolution and visibility), and drawKnots and wrapDamping for the spring (to make knots visible and to adjust the wrap damping as described in Section 7.6).

To run this example in ArtiSynth, select All demos > tutorial > TalusWrapping from the Models menu. Since all of the dynamic components are fixed, running the model will not cause any initial motion. However, while simulating, one can use the viewer's graphical dragger fixtures (see the section "Transformer Tools" in the ArtiSynth User Interface Guide) to move `p0` or `p1` and hence pull the spring across the bone surface (Figure 7.9, left). One can also interactively adjust the property settings in the control panel to view the grid, isosurface, and knots, and the adjust the grid's resolution. Figure 7.9, right, shows the model with renderGrid and drawKnots set to `true` and renderRanges set to `"10:12 * *"`.

7.4 Initializing the Wrap Path

By default, when a multipoint spring or muscle is initialized (either at the start of the simulation or as a result of calling updateWrapSegments()), each wrappable segment is initialized to a straight line between its via points. This path is then adjusted to avoid and wrap around obstacles, using artificial linear forces as described in Section 7.6. The result is a local shortest path that wraps around obstacles instead of penetrating them. However, in some cases, the initial path may not be the one desired; instead, one may want it to wrap around obstacles some other way. This can be achieved by specifying additional intermediate points to initialize the segment as a piecewise linear path which threads its way around obstacles in the desired manner (Figure 7.10). These are specified using the optional `initialPnts` argument to the `setSegmentWrappable()` methods.

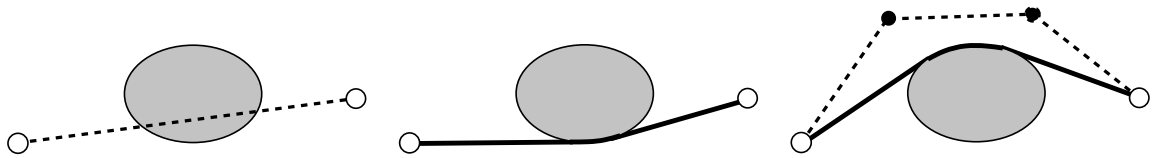


Figure 7.10: By default, the path for each wrappable segment is initialized to a straight line between its via points (dotted line, left), which is then adjusted to wrap around obstacles (solid line, middle). To cause the path to wrap around obstacles in a different way, it can instead be initialized using a piecewise-linear path defined by intermediate initial points (dotted line, right), which will then adjust to an alternate configuration.

When initial points are specified, it is recommended to finish construction of the spring or muscle with a call to updateWrapSegments(). This fits the wrappable segments to their correct path around the obstacles, which can then be seen immediately when the model is first loaded. On the other hand, by *omitting* an initial call to `updateWrapSegments()`, it is possible to see the initial path as specified by the initial points. This may be useful to verify that they are in the correct locations.

In some cases, initial points may also be necessary to help ensure that the initial path does not penetrate obstacles. While obstacle penetration will normally be resolved by the artificial forces described in Section 7.6, this may not always work correctly if the starting path penetrates an obstacle too deeply.

7.4.1 Example: wrapping around a torus

An example of using initial points is given by `artisynth.demos.tutorial.TorusWrapping`, in which a spring is wrapped completely around the inner section of a torus. The primary code for the build method is given below:

```
1       MechModel mech = new MechModel ("mech");
2       addModel (mech);
3
4       mech.setFrameDamping (1.0); // set damping parameters
5       mech.setRotaryDamping (10.0);
6
7       // create the torus
8       double DTOR = Math.PI/180;
```

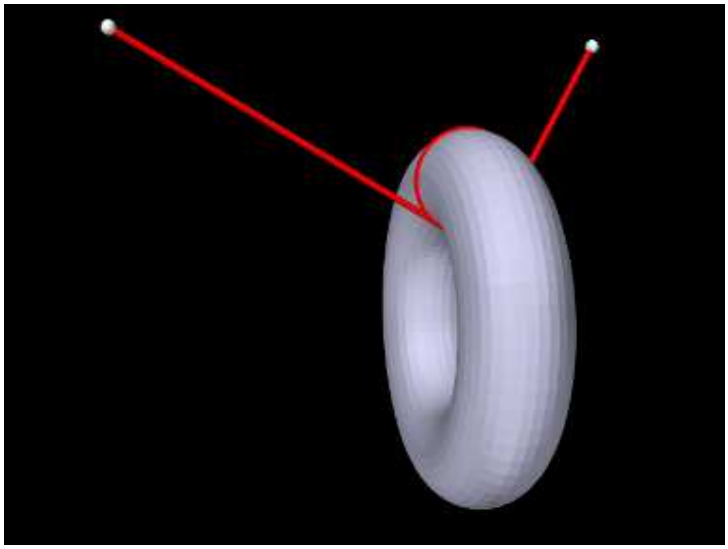


Figure 7.11: `TorusWrapping` model loaded into ArtiSynth.

```
9       double innerRad = 0.75;
10      double outerRad = 2.0;
11      RigidTorus torus =
12         new RigidTorus ("torus", outerRad, innerRad, /*density=*/1);
13      torus.setPose (new RigidTransform3d (2, 0, -2,  0, DTOR*90, 0));
14      mech.addRigidBody (torus);
15
16      // create start and end points for the spring
17      Particle p0 = new Particle (0, /*x,y,z=*/4, 0.2, 2);
18      p0.setDynamic (false);
19      mech.addParticle (p0);
20      Particle p1 = new Particle (0, /*x,y,z=*/-3, -0.2, 2);
21      p1.setDynamic (false);
22      mech.addParticle (p1);
23
24      // create a wrappable MultiPointSpring between p0 and p1, with initial
25      // points specified so that it wraps around the torus
26      MultiPointSpring spring =
27         new MultiPointSpring (/*k=*/10, /*d=*/0, /*restlen=*/0);
28      spring.addPoint (p0);
29      spring.setSegmentWrappable (
30         100, new Point3d[] {
31            new Point3d (3, 0, 0),
32            new Point3d (2, 0, -1),
33            new Point3d (1, 0, 0),
34            new Point3d (2, 0, 1),
35            new Point3d (3, 0, 0),
36            new Point3d (2, 0, -1),
37         });
38      spring.addPoint (p1);
39      spring.addWrappable (torus);
40      spring.updateWrapSegments(); // ``shrink wrap'' around torus
41      mech.addMultiPointSpring (spring);
```

The mech model is created in the usual way with frame and rotary damping set to 1 and 10 (lines 4-5). The torus is created using the analytic wrappable RigidTorus (lines 8-14). The spring start and end points `p0` and `p1` are created at lines (17-22), and the spring itself is created at lines (26-41), with six initial points being specified to `setSegmentWrappable()` to wrap the spring completely around the torus inner section.

To run this example in ArtiSynth, select All demos > tutorial > TorusWrapping from the Models menu. The torus will slide along the wrapped spring until it reaches equilibrium.

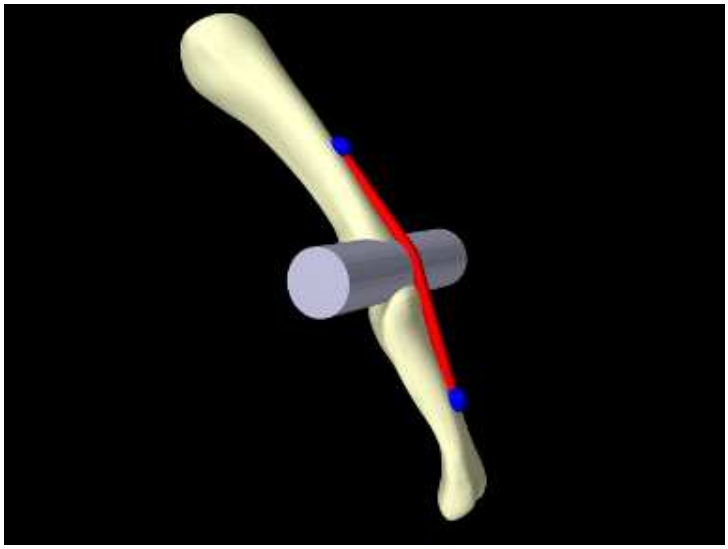


Figure 7.12: PhalanxWrapping model loaded into ArtiSynth.

7.5 Alternate Wrapping Surfaces

Although it common to use the general mesh geometry of a `RigidBody` as the wrapping surface, situations may arise where it is desirable to *not* do this. These may include:

- The general mesh geometry is not sufficiently smooth to form a good wrapping surface;
- Wrapping around the default mesh geometry is not stable, in that it is too easy for the wrap strand to "slip off";
- Using one of the simpler analytic geometries (Table 7.1) may result in a more efficient computation.

There are a couple of ways to handle this. One, discussed in Section 7.3, involves creating a collision mesh which is separate from the general mesh geometry. However, that same collision mesh must then also be used for collision handling (Section 4.5). If that is undesirable, or if *multiple* wrapping surfaces are needed, then a different approach may be used. This involves creating the desired wrappable as a separate object and then *attaching* it to the main `RigidBody`. Typically, this wrappable will be created with zero mass (or density), so that it does not alter the effective mass or inertia of the main body. The general procedure then becomes:

1. Create the main `RigidBody` with whatever desired geometry and inertia is needed;
2. Create the additional wrappable object(s), usually with zero density/mass;
3. Attach the wrappables to the main body using one of the `MechModel attachFrame()` methods described in Section 3.6.3.

7.5.1 Example: wrapping for a finger joint

An example using an alternate wrapping surface is given by `artisynth.demos.tutorial.PhalanxWrapping`, which shows a a muscle wrapping around a joint between two finger bones. Because the bones themselves are fairly narrow, using them as wrapping surfaces would likely lead to the muscle slipping off. Instead, a RigidCylinder is used for the wrapping and attached to one of the bones. The code, with include directives excluded, is given below:

```
1 public class PhalanxWrapping extends RootModel {
2
3    private static Color BONE = new Color (1f, 1f, 0.8f);
4    private static double DTOR = Math.PI/180.0;
5
6    private RigidBody createBody (MechModel mech, String name, String fileName) {
```

```
      // creates a bone from its mesh and adds it to a MechModel
      String filePath = PathFinder.findSourceDir(this) + "/data/" + fileName;
      RigidBody body = RigidBody.createFromMesh (
         name, filePath, /*density=*/1000, /*scale=*/1.0);
      mech.addRigidBody (body);
      RenderProps.setFaceColor (body, BONE);
      return body;
   }

   public void build (String[] args) {

      MechModel mech = new MechModel ("mech");
      addModel (mech);

      // create the two phalanx bones, and offset them
      RigidBody proximal = createBody (mech, "proximal", "HP3ProximalLeft.obj");
      RigidBody distal = createBody (mech, "distal", "HP3MiddleLeft.obj");
      distal.setPose (new RigidTransform3d (0.02500, 0.00094, -0.03979));

      // make the proximal phalanx non dynamic; add damping to the distal
      proximal.setDynamic (false);
      distal.setFrameDamping (0.03);

      // create a revolute joint between the bones
      RigidTransform3d TJW =
         new RigidTransform3d (0.018, 0, -0.022, 0, 0, -DTOR*90);
      HingeJoint joint = new HingeJoint (proximal, distal, TJW);
      joint.setShaftLength (0.02); // render joint as a blue cylinder
      RenderProps.setFaceColor (joint, Color.BLUE);
      mech.addBodyConnector (joint);

      // create markers for muscle origin and insertion points
      FrameMarker origin = mech.addFrameMarkerWorld (
         proximal, new Point3d (0.0098, -0.0001, -0.0037));
      FrameMarker insertion = mech.addFrameMarkerWorld (
         distal, new Point3d (0.0293, 0.0009, -0.0415));

      // create a massless RigidCylinder to use as a wrapping surface and
      // attach it to the distal bone
      RigidCylinder cylinder = new RigidCylinder (
         "wrapSurface", /*rad=*/0.005, /*h=*/0.04, /*density=*/0, /*nsegs=*/32);
      cylinder.setPose (TJW);
      mech.addRigidBody (cylinder);
      mech.attachFrame (cylinder, distal);

      // create a wrappable muscle using a SimpleAxialMuscle material
      MultiPointSpring muscle = new MultiPointMuscle ("muscle");
      muscle.setMaterial (
         new SimpleAxialMuscle (/*k=*/0.5, /*d=*/0, /*maxf=*/0.04));
      muscle.addPoint (origin);
      // add an initial point to the wrappable segment to make sure it wraps
      // around the cylinder the right way
      muscle.setSegmentWrappable (
         50, new Point3d[] { new Point3d (0.025, 0.0, -0.02) });
      muscle.addPoint (insertion);
      muscle.addWrappable (cylinder);
      muscle.updateWrapSegments(); // ``shrink wrap'' around cylinder
      mech.addMultiPointSpring (muscle);

      // set render properties
      RenderProps.setSphericalPoints (mech, 0.002, Color.BLUE);
      RenderProps.setCylindricalLines (muscle, 0.001, Color.RED);
      RenderProps.setFaceColor (cylinder, new Color (200, 200, 230));
   }
}
```

The method `createBody()` (lines 6-14) creates a rigid body from a geometry mesh stored in a file in the directory "`data`" beneath the source directory, using the utility class PathFinder used to determine the file path (Section 2.6).

Within the `build()` method, a `MechModel` is created containing two rigid bodies representing the bones, `proximal` and `distal`, with `proximal` fixed and `distal` free to move with a frame damping of 0.03 (lines 18-28). A cylindrical joint is then added between the bones, along with markers describing the muscle's origin and insertion points (lines 31-42). A `RigidCylinder` is created to act as a wrapping obstacle and attached to the `distal` bone in the same location as the joint (lines 46-50); since it is created with a density of 0 it has no mass and hence does not affect the bone's inertia. The muscle itself is created at lines 53-64, using a SimpleAxialMuscle as a material and an extra initial point specified to `setSegmentWrappable()` to ensure that it wraps around the cylinder in the correct way (Section 7.4). Finally, some render properties are set at lines 67-69.

To run this example in ArtiSynth, select All demos > tutorial > PhalanxWrapping from the Models menu. The model should load and initially appear as in Figure 7.12. When running the model, one can move the `distal` bone either by using the pull tool (Section "Pull Manipulation" in the ArtiSynth User Interface Guide), or selecting the muscle in the GUI, invoking a property dialog by choosing Edit properties ... from the right-click context menu, and adjusting the excitation property.

7.6 Tuning the Wrapping Behavior

Wrappable segments are implemented internally using artificial linear elastic forces to draw the knots together and keep them from penetrating obstacles. These artificial forces are invisible to the simulation: the wrapping segment has no mass, and the knot forces are used to create what is essentially a first order physics that "shrink wraps" each segment around the obstacles at the beginning of each simulation step, forming a shortest-distance geodesic curve from which the wrapping contact points A and B are calculated. This process is now described in more detail.

Assume that a wrappable segment has m knots, indexed by $k = 1, \ldots, m$, each located at a position $\mathbf{x}_k$. Two types of artificial forces then act on each knot: a *wrapping force* that pulls it closer to other knots, and *contact forces* that push it away from wrappable obstacles. The wrapping force is given by

$$\mathbf{f}_{w,k} = K_w(\mathbf{x}_{k+1} - 2\mathbf{x}_k + \mathbf{x}_{k-1})$$

where K_w is the *wrapping stiffness*. To determine the contact forces, we compute, for each wrappable, the knot's distance to the surface d_k and associated normal direction $\mathbf{n}_k$, where $d_k < 0$ implies that the knot is inside. These quantities are determined either analytically (for analytic wrappables, Table 7.1), or using a signed distance grid (for general wrappables, Section 7.3). The contact forces are then given by

$$\mathbf{f}_{c,k} = \begin{cases} -K_c\, d_k\, \mathbf{n}_k & \text{if } d_k < 0 \\ 0 & \text{otherwise,} \end{cases}$$

where K_c is the *contact stiffness*.

The total force $\mathbf{f}_k$ acting on each knot is then given by

$$\mathbf{f}_k = \mathbf{f}_{w,k} + \sum_c \mathbf{f}_{c,k}$$

where the latter term is the sum of contact forces for all wrappables. If we let $\mathbf{x}$ and $\mathbf{f}$ denote the aggregate position and force vectors for all knots, then computing the wrap path involves finding the equilibrium position such that $\mathbf{f}(\mathbf{x}) = 0$. This is done at the beginning of each simulation step, or whenever updateWrapSegments() is called, and is achieved iteratively using Newton's method. If $\mathbf{x}^j$ and $\mathbf{f}(\mathbf{x}^j)$ denote the positions and forces at iteration j, and

$$\mathbf{K} \equiv \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$$

denotes the local force derivative (or "stiffness"), then the basic Newton update is given by

$$\mathbf{x}^{j+1} = \mathbf{x}^j - \mathbf{K}^{-1}\mathbf{f}(\mathbf{x}^j).$$

In practice, to help deal with the nonlinearities associated with contact, we use a damped Newton update,

$$\mathbf{x}^{j+1} = \mathbf{x}^j + \alpha(D\mathbf{I} - \mathbf{K})^{-1}\mathbf{f}(\mathbf{x}^j), \tag{7.1}$$

where D is a constant *wrap damping* parameter, and α is an adaptively computed step size adjustment. The computation of (7.1) can be performed quickly, in $O(m)$ time, since $\mathbf{K}$ is a block-tridiagonal matrix, and the number of iterations required is typically small (on the order of 10 or less), particularly since the iterative procedure continues across simulation steps and so $\mathbf{f}(\mathbf{x})$ does not need to be brought to 0 for any given step. The maximum number of Newton iterations used for each time step is $N_{\max}$.

> Again, it is important to understand the artificial knot forces $\mathbf{f}(\mathbf{x})$ described here are separate from the physical spring/muscle tension forces $f(l,\dot{l},a)$ discussed in Sections 3.1.1 and 4.4.1, and *only* facilitate the computation of each wrappable segment's path around obstacles.

The default values for the wrapping parameters are $K_w = 1$, $K_c = 10$, $D = 10$, and $N_{\max} = 10$, and these often give satisfactory results without the need for modification. However, in some situations the default muscle wrapping may not perform adequately and it is necessary to adjust these parameters. Problems may include:

- The wrapping path does not settle down and tends to "jump around". Solutions include increasing the damping parameter D or the maximum number of wrap iterations N_i. For general wrapping surfaces (Section 7.3), one should also ensure that the surface is sufficiently smooth.
- A wrapping surface is too thin and so the wrapping path "jumps through" it. Solutions include increasing the damping parameter D, increasing the number of knots in the segment, or decreasing the simulation step size. An alternative approach is to use an alternative wrapping surface (Section 7.5) that is thicker and better behaved.

Wrapping parameters are exported as properties of `MultiPointSpring` and `MultiPointMuscle`, and may be changed in code (using their set/get accessors), or interactively, either by exposing them through a control panel, or by selecting the spring/muscle in the GUI and choosing Edit properties ... from the right-click context menu. Property values include:

wrapStiffness Wrapping stiffness K_w between knot points (default value 1). Since the wrapping behavior is determined by the damping to stiffness *ratio*, it is generally not necessary to change this value.

wrapDamping Damping factor D (default value 10). Increasing this value relative to K_w results in wrap path motions that are smoother and less likely to penetrate obstacles, but which are also less dynamically responsive. Applications generally work with damping values between 10 and 100 (assuming $K_w = 1$).

contactStiffness Contact stiffness K_c used to resolve obstacle penetration (default value 10). It is generally not necessary to change this value. Decreasing it will increase the distance that knots are permitted to penetrate obstacles, which *may* result in a slightly more stable contact behavior.

maxWrapIterations Maximum number of Newton iterations $N_{\max}$ per time step (default value 10). If the wrapping simulation exhibits instability, particularly with regard to obstacle contact, increasing the number of iterations (to say 100) may help.

In addition, `MultiPointSpring` and `MultiPointMuscle` also export the following properties to control the rendering of knot and A/B points:

drawKnots If true, renders the knot points in each wrappable segment. This can be useful to visualize the knot density. Knots are rendered using the style, size, and color given by the pointStyle, pointRadius, pointSize, and pointColor values of the spring/muscle's render properties.

drawABPoints If true, renders the A/B points. These are the first and last points of contact that a wrap segment makes with each wrappable, and correspond to the points where the spring/muscle's tension acts on that wrappable (Section 7 and Figure 7.4). A/B points are rendered using the style and size given by the pointStyle, pointRadius ($\times 1.2$) and pointSize values of the spring/muscle's render properties, and the color given by the ABPointColor property.

Chapter 8

Skinning

A useful technique for creating anatomical and biomechanical models is to attach a passive mesh to an underlying set of dynamically active bodies so that it deforms in accordance with the motion of those bodies. ArtiSynth allows meshes to be attached, or "skinned", to collections of both rigid bodies and FEM models, facilitating the creation of structures that are either embedded in, or connect or envelope a set of underlying components. Such approaches are well known in the computer animation community, where they are widely used to represent the deformable tissue surrounding a "skeleton" of articulated rigid bodies, and have more recently been applied to biomechanics [12].

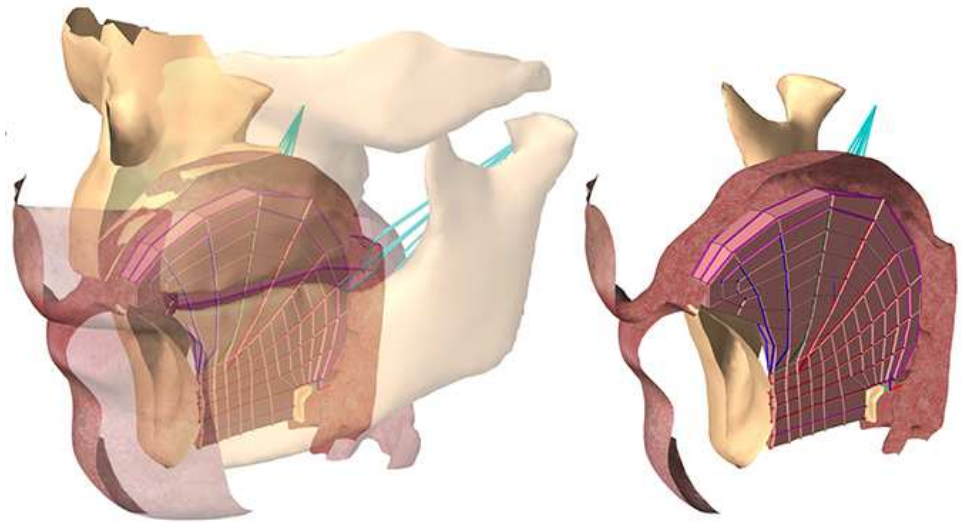


Figure 8.1: A skin mesh used to delimit the boundary of the human upper airway, connected to various surrounding structures including the palate, tongue, and jaw [17].

One application of skinning is to create a continuous skin surrounding an underlying set of anatomical components. For example, for modeling the human airway, a disparate set of models describing the tongue, jaw, palate and pharynx can be connected together with a surface skin to form a seamless airtight mesh (Figure 8.1), as described in [17]. This then provides a uniform boundary for handling air or fluid interactions associated with tasks such as speech or swallowing.

ArtiSynth provides support for "skinning" a mesh over an underlying set of *master bodies*, consisting of rigid bodies and/or FEM models, such that the mesh vertices deform in response to changes in the position, orientation and shape of the master bodies.

8.1 Implementation

This section describes the technical details of the ArtiSynth skinning mechanism. A skin mesh is implemented using a SkinMeshBody, which contains a base mesh and references to a set of underlying dynamic *master bodies*. A master body can be either a Frame (of which RigidBody is a subclass), or a FemModel3d. The positions of the mesh vertices

(along with markers and other points that can be attached to the skin mesh) are determined by a weighted sum of influences from each of the m master bodies, such that as the latter move and/or deform, the vertices and attached points deform as well. More precisely, for each master body $k, k \in \{0,\ldots,m-1\}$, let w_k be the weighting factor and $f_k(\mathbf{q}_k)$ the *connection function* that describes the contribution of body k to the position of the vertices (or attached points) as a function of its generalized coordinates $\mathbf{q}_k$. Then if the position of a vertex (or attached point) is denoted by $\mathbf{p}$ and its initial (or *base*) position is $\mathbf{p}_0$, we have

$$\mathbf{p} = \sum_{k=0}^{m-1} w_k f_k(\mathbf{q}_k) + w_m \mathbf{p}_0. \tag{8.1}$$

The weight w_m in the last term is known as the *base weight* and describes an optional contribution from the base position $\mathbf{p}_0$. Usually $w_m = 0$, unless the vertex is not connected to `any` master bodies, in which case $w_m = 1$, so that the vertex is anchored to its initial position.

In general, connection weights w_k are computed based on the distances d_k between the vertex (or attached point) and each master body k. More details on this are given in Sections 8.2 and 8.3.

For `Frame` master bodies, the connection function is one associated with various rigid body skinning techniques known in the literature. These include linear, linear dual quaternion, and iterative dual quaternion skinning. Which technique is used is determined by the frameBlending property of the `SkinMeshBody`, which can be queried or set in code using the methods

```
FrameBlending getFrameBlending()

void setFrameBlending (FrameBlending blending)
```

where FrameBlending is an enumerated type defined by `SkinMeshBody` with the following values:

LINEAR

Linear blending, in which the connection function $f_k()$ implements a standard rigid connection between the vertex and the frame coordinates. Let the frame's generalized coordinates $\mathbf{q}_k$ be given by the 3×3 rotation matrix $\mathbf{R}$ and translation vector $\mathbf{p}_F$ describing its pose, with its initial pose given by $\mathbf{R}_0$ and $\mathbf{p}_{F0}$. The connection function $f_k()$ then takes the form

$$f_k(\mathbf{R}, \mathbf{p}_F) = \mathbf{R}\mathbf{R}_0^T(\mathbf{p}_0 - \mathbf{p}_{F0}) + \mathbf{p}_F. \tag{8.2}$$

Linear blending is faster than other blending techniques but is more prone to pinching and creasing artifacts in the presence of large rotations between frames.

DUAL_QUATERNION_LINEAR

Linear dual quaternion blending, which is more computationally expensive but typically gives better results than linear blending, and is described in detail as DLB in [5]. Let the frame's generalized coordinates $\mathbf{q}_k$ be given by the dual-quaternion $\hat{\mathbf{q}}_k$ (describing both rotation and translation), with the initial pose given by the dual-quaternion $\hat{\mathbf{q}}_{k0}$. Then define the relative dual-quaternion $\tilde{\mathbf{q}}_k$ as

$$\tilde{\mathbf{q}}_k = \frac{\hat{\mathbf{q}}_k \hat{\mathbf{q}}_{k0}^{-1}}{\| \sum_j w_j \hat{\mathbf{q}}_j \hat{\mathbf{q}}_{j0}^{-1} \|}, \tag{8.3}$$

where the denominator is formed by summing over *all* master bodies j which are frames. The connection function $f_k()$ is then given by

$$\mathbf{f}_k(\hat{\mathbf{q}}_k) = \tilde{\mathbf{q}}_k \mathbf{p}_0 \tilde{\mathbf{q}}_k^{-1} - \mathbf{p}_0, \tag{8.4}$$

where we note that a dual quaternion multiplied by a position vector yields a position vector.

DUAL_QUATERNION_ITERATIVE

Dual quaternion iterative blending, which is a more complex dual quaternion technique described in detail as DIB in [5]. The connection function for iterative dual quaternion blending involves an iterative process and is not described here. It also does not conform to (8.1), because the connection functions $f_k()$ for the `Frame` master bodies do not combine linearly. Instead, if there are r `Frame` master bodies, there is a *single* connection function

$$f(w_0,\ldots,w_{r-1},\hat{\mathbf{q}}_0,\ldots,\hat{\mathbf{q}}_{r-1}) \tag{8.5}$$

that determines the connection for *all* of them, given their weighting factors w_j and generalized coordinates $\hat{\mathbf{q}}_j$. Iterative blending relies on two parameters: a blend tolerance, and a maximum number of blend steps, both of which are controlled by the SkinMeshBody properties DQBlendTolerance and DQMaxBlendSteps, which have default values of 1^{-8} and 3.

> Iterative dual quaternion blending is not completely supported in ArtiSynth. In particular, because of its complexity, the associated force and velocity mappings are computed using the simpler computations employed for linear dual quaternion blending. For the examples shown in this chapter, iterative dual quaternion gives results that are quite close to those of linear dual quaternion blending.

For FEM master bodies, the connection works by tying each vertex (or attached point) to a specific FEM element using a fixed-length offset vector $\mathbf{d}$ that rotates in conjunction with the element. This is illustrated in Figure 8.2 for the case of a single FEM master body. Starting with the initial vertex position $\mathbf{p}_0$, we find the nearest point $\mathbf{p}_{e0}$ on the nearest FEM element, along with the offset vector $\mathbf{d}_0 \equiv \mathbf{p}_0 - \mathbf{p}_{e0}$. The point $\mathbf{p}_{e0}$ can be expressed as the weighted sum of the initial element nodal positions $\mathbf{x}_{j0}$,

$$\mathbf{p}_{e0} = \sum_{j=0}^{n-1} \alpha_j \mathbf{x}_{j0}, \tag{8.6}$$

where n is the number of nodes and α_j represent the (constant) nodal coordinates. As the element moves and deforms, the element point $\mathbf{p}_e$ moves with the nodal positions $\mathbf{x}_j$ according to the same relationship, while the offset vector $\mathbf{d}$ rotates according to $\mathbf{d} = \mathbf{R}_E d_0$, where $\mathbf{R}_E$ is the rotation of the element's coordinate frame E with respect to its initial orientation. The connection function $f_k()$ then takes the form

$$f_k(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) = \sum_{j=0}^{n-1} \alpha_j \mathbf{x}_j + \mathbf{R}_E d_0. \tag{8.7}$$

$\mathbf{R}_E$ is determined by computing a polar decomposition $\mathbf{F} = \mathbf{R}_E \mathbf{P}$ on the deformation gradient $\mathbf{F}$ at the element's center. We note that the displacement $\mathbf{d}$ is only rotated and so the distance $\|\mathbf{d}\| = \|\mathbf{d}_0\|$ of the vertex from the element remains constant. If the vertex is initially on or inside the element, then $\mathbf{d}_0 = 0$ and (8.7) takes the form of a standard point/element attachment as described in 6.4.3.

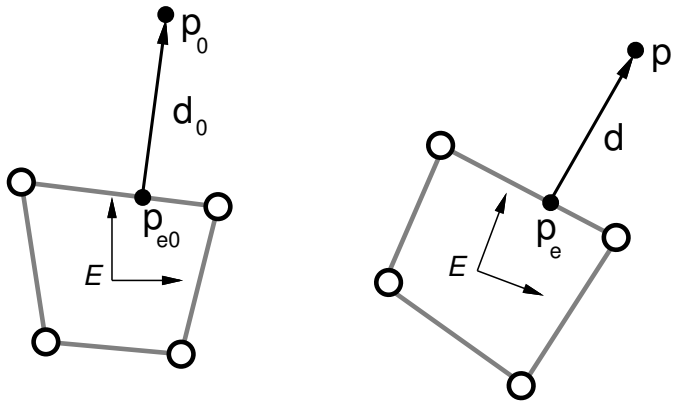


Figure 8.2: Illustration of FEM skinning, showing how a position $\mathbf{p}$ is tied to an FEM element. Given the initial position $\mathbf{p}_0$, we find the nearest point $\mathbf{p}_{e0}$ on the element, along with the offset vector $\mathbf{d}_0 = \mathbf{p}_0 - \mathbf{p}_{e0}$ (left). As the element moves and deforms, the updated position is obtained from $\mathbf{p} = \mathbf{p}_e + \mathbf{d}$, where $\mathbf{p}_e$ deforms with the element, and $\mathbf{d}$ rotates in tandem with its coordinate frame E.

> While it is sometimes possible to determine weights α_j that control a vertex position *outside* an element, without the need for an offset vector $\mathbf{d}$, the resulting vertex positions tend to be very sensitive to element distortions, particularly when the vertex is located at some distance. Keeping the element-vertex distance constant via an offset vector usually results in more plausible skinning behavior.

8.2 Creating a skin mesh

As mentioned above, skin meshes within ArtiSynth are implemented using the SkinMeshBody component. Applications typically create a skin mesh in code according to the following steps:

1. Create an instance of `SkinMeshBody` and assign its underlying mesh, usually within the constructor;
2. Add references to the required master bodies;
3. Compute the master body connections

This is illustrated by the following example:

```
MechModel mech;
PolygonalMesh mesh;

// ... initialize mesh ...

// create the body with an underlying mesh:
SkinMeshBody skinMesh = new SkinMeshBody(mesh);

// add references to the master bodies:
skinMesh.addMasterBody (rigidBody1);
skinMesh.addMasterBody (rigidBody2);
skinMesh.addMasterBody (femModel1);

// compute the weighted connections for each vertex:
skinMesh.computeAllVertexConnections();

// add to the MechModel
mech.addMeshBody (skinMesh)
```

Master body references are added using addMasterBody(). When all the master bodies have been added, the method computeAllVertexConnections() computes the weighted connections to each vertex. The connection weights w_k for each vertex are determined by a *weighting function*, based on the distances d_k between the vertex and each master body. The default weighting function is *inverse-square weighting*, which first computes a set of raw weights w_k^* according to

$$w_k^* = \frac{d_{\min}^2}{d_k^2}, \tag{8.8}$$

where $d_{\min} \equiv \min(d_j)$ is the minimum master body distance, and then normalizes these to determine w_k:

$$w_k = \frac{w_k^*}{\sum_j w_j^*}. \tag{8.9}$$

Other weighting functions can be specified, as described in Section 8.3.

`SkinMeshBody` provides the following set of methods to set and query its master body configuration:

```
void addMasterBody (ModelComponent body)    // add a master body
int numMasterBodies()                       // query number of master bodies
boolean hasMasterBody (ModelComponent body) // query master body presence
ModelComponent getMasterBody (int idx)      // get a master body by index

RigidTransform3d getBasePose (Frame frame)  // query base pose for frame
void setBasePose (Frame frame, RigidTransform3d T) // set base pose for frame
```

When a `Frame` master body is added using `addMasterBody()`, its initial, or *base*, pose (corresponding to $\mathbf{R}_0$ and $\mathbf{p}_{F0}$ in Section 8.1) is set from its current pose. If necessary, applications can later query and reset the base pose using the methods `getBasePose()` and `setBasePose()`.

Internally, each vertex is connected to the master bodies by a PointSkinAttachment, which contains a list of PointSkinAttachment.SkinConnection components describing each master connection. Applications can obtain the `PointSkinAttachment` for each vertex using the method

```
PointSkinAttachment getVertexAttachment (int vidx)
```

where `vidx` is the vertex index, which must be in the range 0 to `numv`-1, with `numv` the number of vertices as returned by `numVertices()`. Methods also exist to query and set each vertex's base (i.e., initial) position $\mathbf{p}_0$:

```
Point3d getVertexBasePosition (int vidx)

void setVertexBasePosition (int vidx, Point3d pos)
```

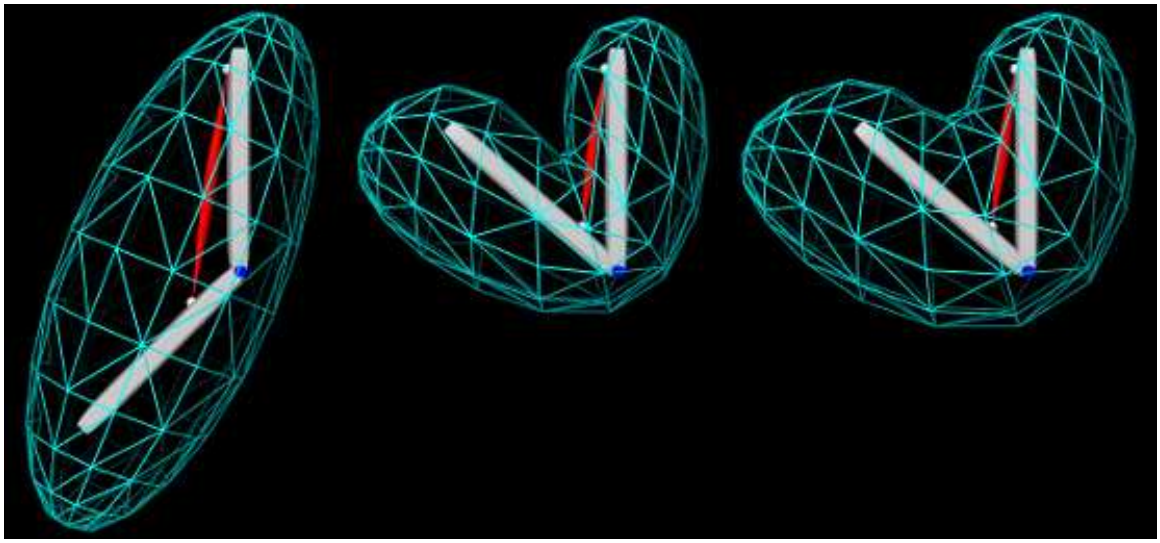


Figure 8.3: RigidBodySkinning model loaded into ArtiSynth (left), then run with excitation set to 0.7 (middle). The right image shows the result of changing the frameBlending property from `LINEAR` to `DUAL_QUATERNION_LINEAR`
.

8.2.1 Example: skinning over rigid bodies

An example of skinning a mesh over two rigid bodies is given by `artisynth.demos.tutorial.RigidBodySkinning`. It consists of a `SkinMeshBody` placed around two rigid bodies connected by a hinge joint to form a toy "arm", with a Muscle added to move the lower body with respect to the upper. The code for the `build()` method is given below:

```
1    public void build (String[] args) throws IOException {
2       MechModel mech = new MechModel ("mech");
3       addModel (mech);
4
5       // set damping parameters for rigid bodies
6       mech.setFrameDamping (10);
7       mech.setRotaryDamping (100.0);
8
9       // create a toy "arm" conisting of upper and lower rigid bodies connected
10      // by a revolute joint:
11      double len = 2.0;
12      RigidBody upper = addBody (mech, "upper");
13      upper.setPose (new RigidTransform3d (0, 0, len/2));
14      upper.setDynamic (false);  // upper body is fixed
15
16      RigidBody lower = addBody (mech, "lower");
17      // reposition the lower body"
18      double angle = Math.toRadians(225);
19      double sin = Math.sin(angle);
20      double cos = Math.cos(angle);
21      lower.setPose (new RigidTransform3d (sin*len/2, 0, cos*len/2, 0, angle, 0));
22
23      // add the revolute joint between the upper and lower bodies:
24      HingeJoint joint =
25         new HingeJoint (lower, upper, new Point3d(), Vector3d.Y_UNIT);
26      joint.setName ("elbow");
27      mech.addBodyConnector (joint);
28
29      // add two frame markers and a "muscle" to move the lower body
30      FrameMarker mku = mech.addFrameMarker (
31         upper, new Point3d(-len/20, 0, len/2.4));
32      FrameMarker mkl = mech.addFrameMarker (
33         lower, new Point3d(len/20, 0, -len/4));
34      Muscle muscle = new Muscle("muscle");
```

```
35        muscle.setMaterial (new SimpleAxialMuscle (1000.0, 0, 2000.0));
36        mech.attachAxialSpring (mku, mkl, muscle);
37
38        // create an ellipsoidal base mesh for the SkinMeshBody by scaling a
39        // spherical mesh
40        PolygonalMesh mesh = MeshFactory.createSphere (1.0, 12, 12);
41        mesh.scale (1, 1, 2.5);
42        mesh.transform (
43           new RigidTransform3d (-0.6, 0, 0, 0, Math.toRadians(22.5),0));
44
45        // create the skinMesh, with the upper and lower bodies as master bodies
46        SkinMeshBody skinMesh = new SkinMeshBody (mesh);
47        skinMesh.addMasterBody (upper);
48        skinMesh.addMasterBody (lower);
49        skinMesh.computeAllVertexConnections();
50        mech.addMeshBody (skinMesh);
51
52        // add a control panel to adjust the muscle excitation and frameBlending
53        ControlPanel panel = new ControlPanel();
54        panel.addWidget (muscle, "excitation");
55        panel.addWidget (skinMesh, "frameBlending");
56        addControlPanel (panel);
57
58        // set up render properties
59        RenderProps.setFaceStyle (skinMesh, Renderer.FaceStyle.NONE);
60        RenderProps.setDrawEdges (skinMesh, true);
61        RenderProps.setLineColor (skinMesh, Color.CYAN);
62        RenderProps.setSpindleLines (muscle, 0.06, Color.RED);
63        RenderProps.setSphericalPoints (mech, 0.05, Color.WHITE);
64        RenderProps.setFaceColor (joint, Color.BLUE);
65        joint.setShaftLength(len/3);
66        joint.setShaftRadius(0.05);
67        RenderProps.setFaceColor (mech, new Color (0.8f, 0.8f, 0.8f));
68     }
```

A `MechModel` is created in the usual way (lines 2-7). To this is added a very simple toy "arm" consisting of an upper and lower body connected by a hinge joint (lines 9-27), with a simple point-to-point muscle attached between frame markers on the upper and lower bodies to provide a means of moving the arm (lines 29-36). Creation of the arm bodies uses an `addBody()` method which is not shown.

The mesh to be skinned is an ellipsoid, created using the FemFactory method `createSphere()` to produce a spherical mesh which is then scaled and repositioned (lines 38-43). The skin body itself is then created around this mesh, with the upper and lower bodies assigned as master bodies and the connections computed using `computeAllVertexConnections()` (lines 45-50).

A control panel is added to allow control over the muscle's excitation as well as the skin body's frameBlending property (lines 52-56). Finally, render properties are set (lines 58-67): the skin mesh is made transparent by setting its faceStyle and drawEdges properties to `NONE` and `true`, respectively, with cyan colored edges; the muscle is rendered as a red spindle; the joint is drawn as a blue cylinder and the bodies are colored light grey.

To run this example in ArtiSynth, select All demos > tutorial > RigidBodySkinning from the Models menu. The model should load and initially appear as in Figure 8.3 (left). When running the simulation, the arm can be flexed by adjusting the muscle excitation property in the control panel, causing the skin mesh to deform (Figure 8.3, middle). Changing the frameBlending property from its default value of `LINEAR` to `DUAL_QUATERNION_LINEAR` causes the mesh deformation to become fatter and less prone to creasing (Figure 8.3, right).

8.3 Computing weights

As described above, the default method for computing skin connection weights is inverse-square weighting (equations 8.8) and 8.9). However, applications can specify alternatives to this. The method

```
    void setGaussianWeighting (double sigma)
```

causes weights to be computed according to a *Gaussian weighting* scheme, with `sigma` specifying the standard deviation σ. Raw weights w_k^* are then computed according to

$$w_i = \exp\left(-\frac{(d_k - d_{\min})^2}{2\sigma^2}\right),$$

and then normalized to form w_k.

The method

```
void setInverseSquareWeighting()
```

reverts the weighting function back to inverse-square weighting.

It is also possible to specify a custom weighting function by implementing a subclass of SkinWeightingFunction. Subclasses must implement the function

```
void computeWeights (
    double[] weights, Point3d pos, NearestPoint[] nearestPnts);
```

in which the weights for each master body are computed and returned in `weights`. `pos` gives the initial position of the vertex (or attached point) being skinning, while `nearestPnts` provides information about the distance from `pos` to each of the master bodies, using an array of SkinMeshBody.NearestPoint objects:

```
class NearestPoint {
   public Point3d nearPoint;   // nearest point on the body
   public double distance;     // distance to the body
   public ModelComponent body; // master body (either Frame or FemModel3d)
}
```

Once an instance of `SkinWeightingFunction` has been created, it can be set as the skin mesh weighting function by calling

```
void setWeightingFunction (SkinWeightingFunction fxn)
```

Subsequent calls to `computeAllVertexConnections()`, or the `addMarker` or `computeAttachment` methods described in Section 8.4, will then employ the specified weighting.

As an example, imagine an application wishes to compute weights according to an inverse-cubic weighting function, such that to

$$w_k^* = \frac{d_{\min}^3}{d_k^3}.$$

A subclass of `SkinWeightingFunction` implementing this could then be defined as

```
class MyWeighting extends SkinWeightingFunction {

   // implements inverse-cubic weighting
   public void computeWeights (
      double[] weights, Point3d pos, NearestPoint[] nearestPnts) {

      // find minimum distance to all the master bodies
      double dmin = Double.POSITIVE_INFINITY;
      for (int i=0; i<nearestPnts.length; i++) {
         if (nearestPnts[i].distance < dmin) {
            dmin = nearestPnts[i].distance;
         }
      }
      double sumw = 0; // sum of all weights (for normalizing)
      // compute raw weights:
      for (int i=0; i<nearestPnts.length; i++) {
         double d = nearestPnts[i].distance;
         double w;
         if (d == dmin) {
            w = 1;  // handles case where dmin = d = 0
         }
```

```
            else {
               w = dmin*dmin*dmin/(d*d*d);
            }
            weights[i] = w;
            sumw += w;
         }
         // normalize the weights:
         for (int i=0; i<nearestPnts.length; i++) {
            weights[i] /= sumw;
         }
      }
   }
```

and then set as the weighting function using the code fragment:

```
   SkinMeshBody skinMesh;
   // ...
   skinMesh.setWeightingFunction (new MyWeighting());
```

The current weighting function for a skin mesh can be queried using

```
   SkinWeightingFunction getWeightingFunction()
```

The inverse-square and Gaussian weighting methods described above are implemented using the system-provided `SkinWeightingFunction` subclasses InverseSquareWeighting and GaussianWeighting, respectively.

8.3.1 Setting weights explicitly

As an alternative to the weighting function, applications can also create connections to vertices or points in which the weights are explicitly specified. This allows for situations in which a weighting function is unable to properly specify all the weights correctly.

When a mesh is initially added to a skin body, via either the constructor SkinMeshBody(mesh), or by a call to setMesh(mesh), all master body connections are cleared and the vertex position is "fixed" to its initial position, also known as its *base position*. After the master bodies have been added, vertex connections can be created by calling computeAllVertexConnections(), as described above. However, connections can also be created on a per-vertex basis, using the method

```
   void computeVertexConnections (int vidx, VectorNd weights)
```

where `vidx` is the index of the desired vertex and `weights` is an optional argument which if non-`null` explicitly specifies the connection weights. A sketch example of how this can be used is given in the following code fragment:

```
   VectorNd weights = new VectorNd (skinMesh.numMasterBodies());
   // compute connections for each vertex
   for (int i=0; i<skinMesh.numVertices(); i++) {
      // ... compute connections weights as required ...
      skinMesh.computeVertexConnections (i, weights);
   }
```

For comparison, it should be noted that the code fragment

```
   for (int i=0; i<skinMesh.numVertices(); i++) {
      skinMesh.computeVertexConnections (i, null);
   }
```

in which weights are *not* explicitly specified, is is equivalent to calling `computeAllVertexConnections()`.

If necessary, after vertex connections have been computed, they can also be cleared, using the method

```
   void clearVertexConnections (int vidx)
```

This will disconnect the vertex with index `vidx` from the master bodies, and set its base weighting w_m (equation 8.1) to 1, so that it will remain fixed to its initial position.

In some special cases, it may be desirable for an application to set attachment base weights to some value other than 0 when connections are present. Base weights for vertex attachments can be queried and set using the methods

```
double getVertexBaseWeight (int vidx)

void setVertexBaseWeight (int vidx, double weight, boolean normalize)
```

In the second method, the argument `normalize`, if `true`, causes the weights of the other connections to be scaled so that the total weight sum remains the same. For skin markers and point attachments (Section 8.4), base weights can be set by calling the equivalent `PointSkinAttachment` methods

```
double getBaseWeight ()

void setBaseWeight (double weight, boolean normalize)
```

(If needed, the attachment for a skin marker can be obtained by calling its `getAttachment()` method.) In addition, base weights can also be specified in the `weights` argument to the method `computeVertexConnections(vidx,weights)`, as well as the methods `addMarker(name,pos,weights)` and `createPointAttachment(pnt,weights)` described in Section 8.4. This is done by giving `weights` a size equal to $m+1$, where m is the number of master bodies, and specifying the base weight in the last location.

8.4 Markers and point attachments

In addition to controlling the positions of mesh vertices, a `SkinMeshBody` can also be used to control the positions of dynamic point components, including markers and other points which can be attached to the skin body. For both markers and attached points, any applied forces are propagated back onto the skin body's master bodies, using the principle of virtual work. This allows skin bodies to be fully incorporated into a dynamic model.

Markers and point attachments can be created even if the `SkinMeshBody` does not have a mesh, a fact that can be used in situations where a mesh is unnecessary, such as when employing skinning techniques for muscle wrapping (Section 8.7).

8.4.1 Markers

Markers attached to a skin body are instances of SkinMarker, and are contained in the body's subcomponent list `markers` (analogous to the `markers` list for FEM models). Markers can be created and maintained using the following `SkinMeshBody` methods:

```
// create markers:
SkinMarker addMarker (Point3d pos)
SkinMarker addMarker (String name, Point3d pos)
SkinMarker addMarker (
   String name, Point3d pos, VectorNd weights)

// remove markers:
boolean removeMarker (SkinMarker mkr)
void clearMarkers();

// access the marker list:
PointList<SkinMarker> markers()
```

The `addMarker` methods each create and return a `SkinMarker` which is added to the skin body's list of markers. The marker's initial position is specified by `pos`, while the second and third methods also allow a name to be specified. Connections between the marker and the master bodies are created in the same way as for mesh vertices, with the connection weights either being determined by the skin body's weighting function (as returned by getWeightingFunction()), or explicitly specified by the argument `weights` (third method).

Once created, markers can be removed individually or all together by the `removeMarker()` and `clearMarkers()` methods. The entire marker list can be accessed on a read-only basis by the method `markers()`.

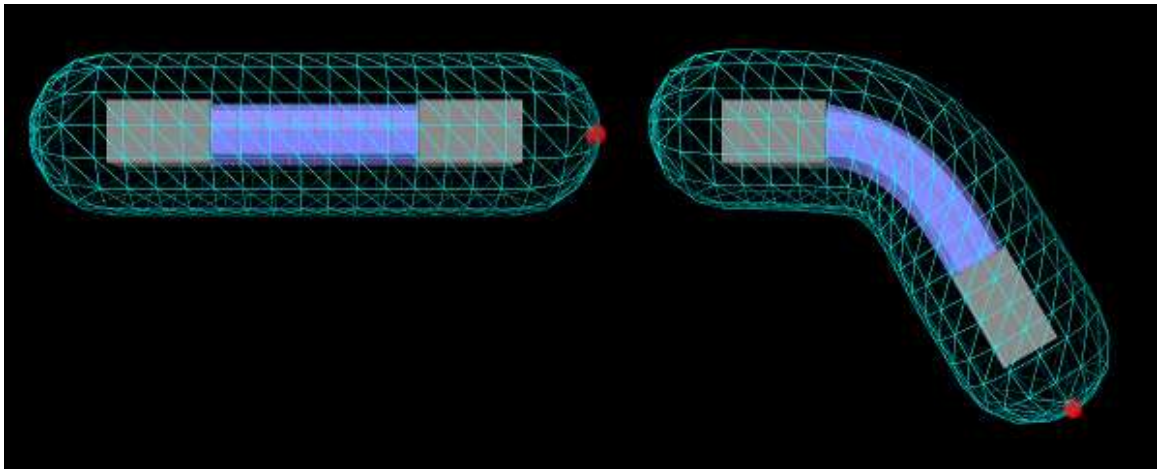


Figure 8.4: AllBodySkinning model as first loaded into ArtiSynth (left), and after the simulation is run and the model has fallen under gravity (right). The skin mesh is rendered in cyan using only its edges.

8.4.2 Point attachments

In addition to markers, applications can also attach any regular Point component (including particles and FEM nodes) to a skin body by using one of its `createPointAttachment` methods:

```
PointSkinAttachment createPointAttachment (Point pnt)
PointSkinAttachment createPointAttachment (Point pnt, VectorNd weights)
```

Both of these create a PointSkinAttachment that connects the point `pnt` to the master bodies in the same way as for mesh vertices and markers, with the connection weights either being determined by the skin body's weighting function or explicitly specified by the argument `weights` in the second method.

Once created, the point attachment must also be added to the underlying `MechModel`, as illustrated by the following code fragment:

```
MechModel mech;
SkinMeshBody skinBody;
Point pnt;

// ... initialize ...

PointSkinAttacment a = skinBody.createPointAttachment(pnt);
mech.addAttachment (a);
```

8.4.3 Example: skinning rigid bodies and FEM models

An example of skinning a mesh over both rigid bodies and FEM models is given by the demo model `artisynth.demos.tutorial.AllBodySkinning`. It consists of a skin mesh placed around a tubular FEM model connected to rigid bodies connected at each end, and a marker attached to the mesh tip. The code for the `build()` method is given below:

```
1    public void build (String[] args) {
2       MechModel mech = new MechModel ("mech");
3       addModel (mech);
4
5       // size and density parameters
6       double len = 1.0;
7       double rad = 0.15;
8       double density = 1000.0;
9
10      // create a tubular FEM model, and rotate it so it lies along the x axis
```

```
11        FemModel3d fem = FemFactory.createHexTube (
12           null, len, rad/3, rad, 8, 8, 2);
13        fem.transformGeometry (new RigidTransform3d (0, 0, 0, 0, Math.PI/2, 0));
14        mech.addModel (fem);
15
16        // create two rigid body blocks
17        RigidBody block0 =
18           RigidBody.createBox ("block0", len/2, 2*rad, 2*rad, density);
19        block0.setPose (new RigidTransform3d (-3*len/4, 0, 0));
20        mech.addRigidBody (block0);
21        block0.setDynamic (false);
22
23        RigidBody block1 =
24           RigidBody.createBox ("block1", len/2, 2*rad, 2*rad, density);
25        block1.setPose (new RigidTransform3d (3*len/4, 0, 0));
26        mech.addRigidBody (block1);
27
28        // attach the blocks to each end of the FEM model
29        for (FemNode3d n : fem.getNodes()) {
30           if (Math.abs(n.getPosition().x-len/2) < EPS) {
31              mech.attachPoint (n, block1);
32           }
33           if (Math.abs(n.getPosition().x+len/2) < EPS) {
34              mech.attachPoint (n, block0);
35           }
36        }
37        fem.setMaterial (new LinearMaterial (500000.0, 0.49));
38
39        // create base mesh to be skinned
40        PolygonalMesh mesh =
41           MeshFactory.createRoundedCylinder (
42              /*r=*/0.4, 2*len, /*nslices=*/16, /*nsegs=*/15, /*flatbotton=*/false);
43        // rotate mesh so its long axis lies along the x axis
44        mesh.transform (new RigidTransform3d (0, 0, 0, 0, Math.PI/2, 0));
45
46        // create the skinBody, with the FEM model and blocks as master bodies
47        SkinMeshBody skinBody = new SkinMeshBody ("skin", mesh);
48        skinBody.addMasterBody (fem);
49        skinBody.addMasterBody (block0);
50        skinBody.addMasterBody (block1);
51        skinBody.computeAllVertexConnections();
52        mech.addMeshBody (skinBody);
53
54        // add a marker point to the end of the skin mesh
55        SkinMarker marker =
56           skinBody.addMarker ("marker", new Point3d(1.4, 0.000, 0.000));
57
58        // set up rendering properties
59        RenderProps.setFaceStyle (skinBody, FaceStyle.NONE);
60        RenderProps.setDrawEdges (skinBody, true);
61        RenderProps.setLineColor (skinBody, Color.CYAN);
62        fem.setSurfaceRendering (FemModel.SurfaceRender.Shaded);
63        RenderProps.setFaceColor (fem, new Color (0.5f, 0.5f, 1f));
64        RenderProps.setSphericalPoints (marker, 0.05, Color.RED);
65     }
```

A `MechModel` is first created and length and density parameters are defined (lines 2-8). Then a tubular FEM model is created, by using the FemFactory method `createHexTube()` and transforming the result by rotating it by 90 degrees about the *y* axis (lines 10-14). Two rigid body blocks are then created (lines 16-26) and attached to the ends of the FEM model by finding and attaching the left and rightmost nodes (lines 28-24). The model is anchored to ground by setting the left block to be non-dynamic (line 21).

The mesh to be skinned is a rounded cylinder, created using the MeshFactory method `createRoundedCylinder()` and rotating the result by 90 degrees about the *y* axis (lines 39-44). This is then used to create the skin body itself, to which both rigid bodies and the FEM model are added as master bodies and the vertex connections are computed using a

call to `computeAllVertexConnections()` (lines 46-52). A marker is then added to the tip of the skin body, using the `addMarker()` method (lines 54-56). Finally render properties are set (lines 58-64): The mesh is made transparent by drawing only its edges in cyan; the FEM model surface mesh is rendered in blue-grey, and the tip marker is drawn as a red sphere.

To run this example in ArtiSynth, select All demos > tutorial > AllBodySkinning from the Models menu. The model should load and initially appear as in Figure 8.4 (left). When running the simulation, the FEM and the rightmost rigid body fall under gravity, causing the skin mesh to deform. The pull tool can then be used to move things around by applying forces to the master bodies or the skin mesh itself.

8.4.4 Mesh-based markers and attachments

For the markers and point attachments described above, the connections to the underlying master bodies are created in the same manner as connections for individual mesh vertices. This means that the resulting markers and attached points move *independently* of the mesh vertices, as though they were vertices in their own right.

An advantage to this is that such markers and attachments can be created even if the `SkinMeshBody` does not even have a mesh, as noted above. However, a disadvantage is that such markers will not remain tightly connected to vertex-based features (such as the faces of a PolygonalMesh or the line segments of a PolylineMesh). For example, consider a marker defined by

```
SkinMarker mkr = skinBody.addMarker (pos);
```

where `pos` is a point that is initially located on a face of the body's mesh. As the master bodies move and the mesh deforms, the resulting marker may not remain strictly on the face. In many cases, this may not be problematic or the deviation may be too small to matter. However, *if* it is desirable for markers or point attachments to be tightly bound to mesh features, they can instead be created with the following methods:

```
// create mesh-based markers:
SkinMarker addMeshMarker (Point3d pos)
SkinMarker addMeshMarker (String name, Point3d pos)

// create mesh-based attachments:
PointSkinAttachment createPointMeshAttachment (Point pnt)
```

The requested position `pos` will then be projected onto the nearest mesh feature (e.g., a face for a `PolygonalMesh` or a line segment for a `PolylineMesh`), and the resulting position $\mathbf{p}$ will be defined as a linear combination of the vertex positions $\mathbf{p}_i$ for this feature,

$$\mathbf{p} = \sum_i \beta_i \mathbf{p}_i, \tag{8.10}$$

where β_i are the barycentric coordinates of $\mathbf{p}$ with respect to the feature. The master body connections are then defined by the same linear combination of the connections for each vertex. When the master bodies move, the marker or attached point will move with the feature and remain in the same relative position.

Since `SkinMeshBody` implements the interface PointAttachable, it provides the general point attachment method

```
PointSkinAttachment createPointAttachment (Point pnt)
```

which allows it to be acted on by agents such as the ArtiSynth marker tool (see the section "Marker tool" in the ArtiSynth User Interface Guide). Whether or not the resulting attachment is a regular attachment or mesh-based is controlled by the skin body's attachPointsToMesh property, which can be adjusted in the GUI or in code using the property's accessor methods:

```
boolean getAttachPointsToMesh()
void setAttachPointsToMesh (boolean enable)
```

8.5 Resolution and Limitations

Skinning techniques do have limitations, which are common to all methodologies.

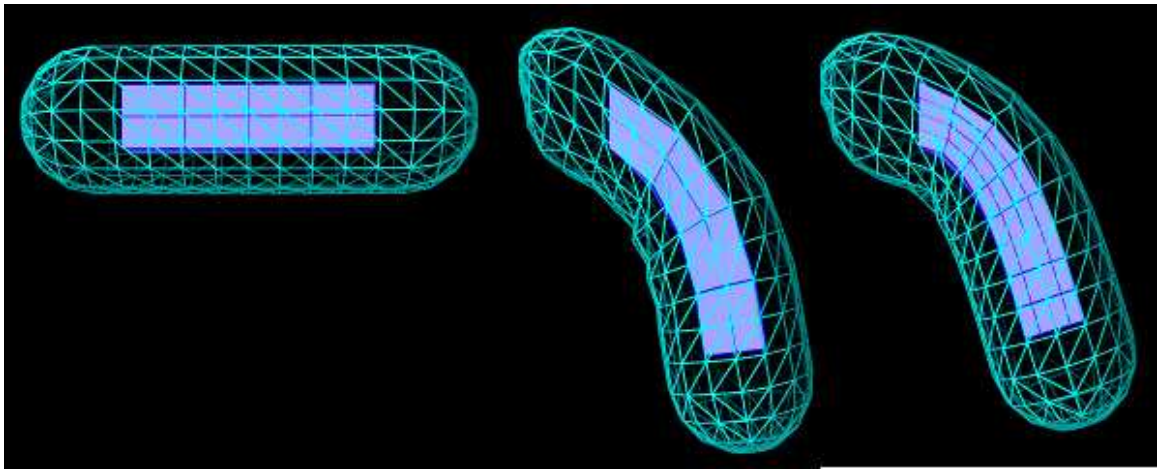


Figure 8.5: A skin mesh enveloping a single FEM body whose element resolution is lower than that of the mesh (left). When the FEM undergoes a large deformation, this disparity in resolution can cause artifacts such as crimping, as seen on the underside of the mesh in the middle image. Using an FEM model with a higher resolution can mitigate this (right).

- The passive nature of the connection between skinned vertices and the master bodies means that it can be easy for the skin mesh to self intersect and/or fold and crease in ways that are not physically realistic. These effects are often more pronounced when mesh vertices are relatively far away from the master bodies, or in places where the mesh undergoes a concave deformation. When the master bodies include frames, these effects can sometimes be reduced by setting the frameBlending property to `DUAL_QUATERNION_LINEAR` instead of the default `LINEAR`.
- When the master bodies include FEM models which undergo large deformations, crimping artifacts may arise if the skin mesh has a higher resolution that the FEM model (Figure 8.5). This is because each mesh vertex is connected to the coordinate frame of a single FEM element, and for reasons of computational efficiency the influence of these coordinate frames is not blended as it is for frame-based master bodies. If crimping artifacts occur, one solution may be to adjust the mesh and/or the FEM model so that their resolutions are more compatible (Figure 8.5, right).

8.6 Collisions

It is possible to make skin bodies collide with other ArtiSynth bodies, such as `RigidBody` and `FemModel3d`, which implement the Collidable interface (Section 4.5). `SkinMeshBody` itself implements CollidableBody, and is considered a deformable body, so that collisions can be activated either by setting one of the default collision behaviors involving deformable bodies, or by setting an explicit collision behavior between it and another body. Self collisions involving `SkinMeshBody` are not currently supported.

As described in Section 4.6, collisions work by computing the intersection between the meshes of the skin body and other collidables. The vertices, faces, and (possibly) edges of the resulting intersection region are then used to compute contact constraints, which propagate the effect of the contact back onto the collidable bodies' dynamic components. For a skin mesh, the dynamic components are the Frame master bodies and the nodes of the FEM master bodies.

Collisions involving `SkinMeshBody` frequently suffer from the problem of being overconstrained (Section 4.6.3), whereby the the number of contacts exceeds the number of master body DOFs available to handle the collision. This may occur if the skin body contains only rigid bodies, or if the mesh resolution exceeds the resolution of the FEM master bodies. Managing overconstrained collisions is discussed in Section 4.6.3, with the easiest method being constraint reduction, which can be activated by setting to `true` the reduceConstraints property for either the collision manager *or* a specific CollisionBehavior involving the skin body.

> Caveats: The distance between the vertices of a skinned mesh and its master bodies can sometimes cause odd or counter-intuitive collision behavior. Collision handling may also be noticeably slower if frameBlending is set to `DUAL_QUATERNION_LINEAR` or `DUAL_QUATERNION_ITERATIVE`. If any of the master bodies are FEM models, collisions resulting in large friction forces may result in unstable behavior.

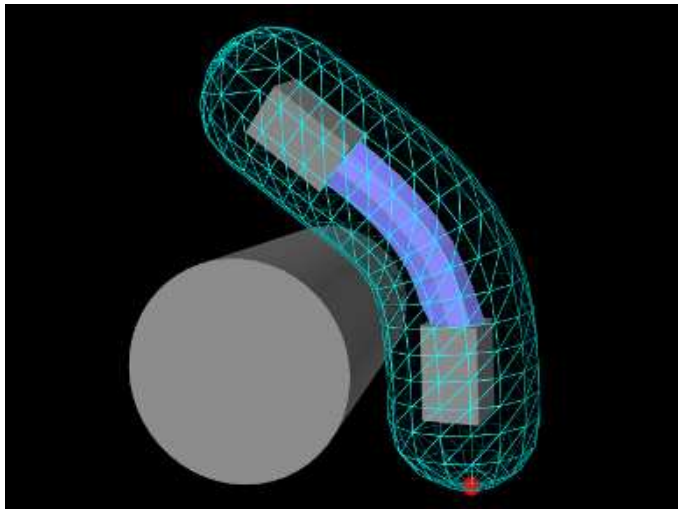


Figure 8.6: `SkinBodyCollide` demo, showing a skin mesh colliding with a cylinder about 0.6 seconds into the simulation.

8.6.1 Example: collision with a cylinder

Collisions involving `SkinMeshBody` are illustrated by the demo model `artisynth.demos.tutorial.SkinBodyCollide`, which extends the demo `AllBodySkinning` to add a cylinder with which the skin body can collide. The code for the demo is given below:

```
package artisynth.demos.tutorial;

import artisynth.core.femmodels.SkinMeshBody;
import artisynth.core.mechmodels.CollisionBehavior;
import artisynth.core.mechmodels.MechModel;
import artisynth.core.mechmodels.RigidBody;
import maspack.matrix.RigidTransform3d;

public class SkinBodyCollide extends AllBodySkinning {

   public void build (String[] args) {
      super.build (args);

      // get components from the super class
      MechModel mech = (MechModel)models().get("mech");
      SkinMeshBody skinBody = (SkinMeshBody)mech.meshBodies().get("skin");
      RigidBody block0 = mech.rigidBodies().get("block0");

      // set block0 dynamic so the skin body and its masters can
      // fall under gravity
      block0.setDynamic (true);

      // create a cylinder for the skin body to collide with
      RigidBody cylinder =
         RigidBody.createCylinder (
            "cylinder", 0.5, 2.0, /*density=*/1000.0, /*nsides=*/50);
      cylinder.setDynamic (false);
      cylinder.setPose (
         new RigidTransform3d (-0.5, 0, -1.5, 0, 0, Math.PI/2));
      mech.addRigidBody (cylinder);

      // enable collisions between the cylinder and the skin body
      CollisionBehavior cb = new CollisionBehavior (true, 0);
      mech.setCollisionBehavior (cylinder, skinBody, cb);
      mech.getCollisionManager().setReduceConstraints (true);
   }
```

```
37 }
```

The model subclasses `AllBodySkinning`, using the superclass `build()` method within its own `build` method to create the original model (line 12). It then obtains references to the `MechModel`, `SkinMeshBody`, and leftmost block, using their names to find them in various component lists (lines 14-17). (If the original model had stored references to these components as accessible member attributes, this step would not be needed.)

These component references are then used to make changes to the model: the left block is made dynamic so that the skin mesh can fall freely (line 21), a cylinder is created and added (lines 23-30), collisions are enabled between the skin body and the cylinder (lines 32-34), and the collision manager is asked to use constraint reduction to minimize the chance of overconstrained contact (line 35).

To run this example in ArtiSynth, select All demos > tutorial > SkinBodyCollide from the Models menu. When run, the skin body should fall and collide with the cylinder as shown in Figure 8.6.

8.7 Application to muscle wrapping

It is sometimes possible to use skinning as a computationally cheaper way to implement muscle wrapping (Chapter 7).

Typically, the end points (i.e., origin and insertion points) of a point-to-point muscle are attached to different bodies. As these bodies move with respect to each other, the path of the muscle may *wrap* around portions of these bodies and perhaps other intermediate bodies as well. The wrapping mechanism of Chapter 7 manages this by performing the computations necessary to allow one or more *wrappable* segments of a MultiPointSpring to wrap around a prescribed set of rigid bodies. However, if the induced path deformation is not too great, it may be possible to achieve a similar effect at much lower computational cost by simply "skinning" the via points of the multipoint spring to the underlying rigid bodies.

The general approach involves:

1. Creating a SkinMeshBody which references as master bodies the bodies containing the origin and insertion points, and possibly other bodies as well;
2. Creating the wrapped muscle using a MultiPointSpring with via points that are attached to the skin body.

It should be noted that for this application, the skin body does not need to contain a mesh. Instead, "skinning" connections can be made solely between the master bodies and the via points. An easy way to do this is to simply use skin body markers as via points. Another way is to create the via points as separate particles, and then attach them to the skin body using one of its `createPointAttachment` methods.

> It should also be noted that unlike with the wrapping methods of Chapter 7, skin-based wrapping can be applied around FEM models as well as rigid bodies.

> Generally, we observe better wrapping behavior if the frameBlending property of the `SkinMeshBody` is set to `DUAL_QUATERNION_LINEAR` instead of the default value of `LINEAR`.

8.7.1 Example: wrapping for a finger joint

An example of skinning-based muscle wrapping is given by `artisynth.demos.tutorial.PhalanxSkinWrapping`, which is identical to the demo `artisynth.demos.tutorial.PhalanxWrapping` except for using skinning to achieve the wrapping effect. The portion of the code which differs is shown below:

```
1       // create a SkinMeshBody and use it to create "skinned" muscle via points
2       SkinMeshBody skinBody = new SkinMeshBody();
3       skinBody.addMasterBody (proximal);
4       skinBody.addMasterBody (distal);
5       skinBody.setFrameBlending (FrameBlending.DUAL_QUATERNION_LINEAR);
6       mech.addMeshBody (skinBody);
7       SkinMarker via1 = skinBody.addMarker (new Point3d (0.0215, 0, -0.015));
```

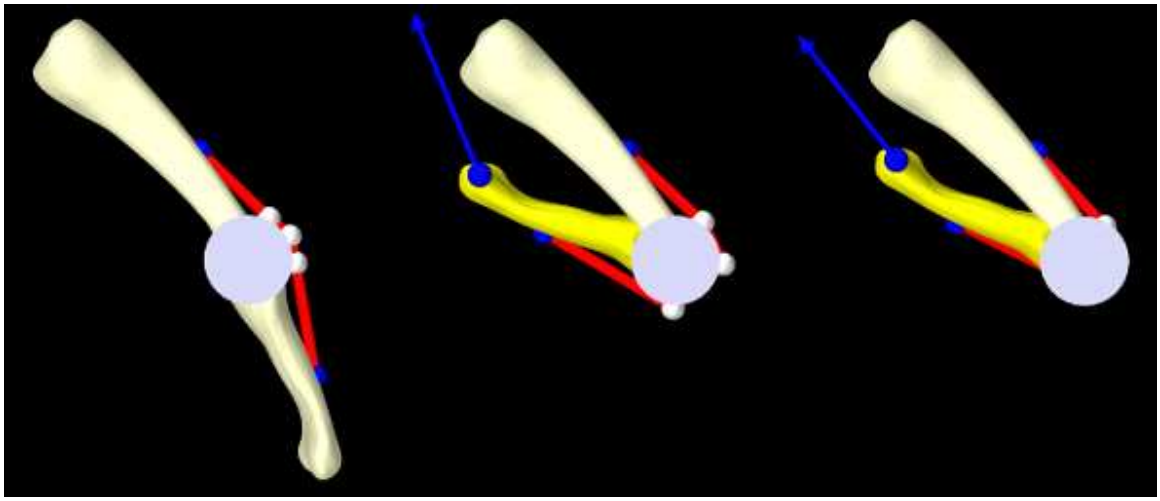


Figure 8.7: PhalanxSkinWrapping model loaded into ArtiSynth and running with no load on the distal bone (left). The pull tool is then used to exert forces on the distal bone and pull it around the joint (middle). The viewer has been set to orthographic projection to enable better visualization of the wrapping behavior of the muscle via points (shown in white). The results appear better when the SkinMeshBody's frameBlending property is set to `DUAL_QUATERNION_LINEAR` (middle) instead of `LINEAR` (right).

```
8       SkinMarker via2 = skinBody.addMarker (new Point3d (0.025, 0, -0.018));
9       SkinMarker via3 = skinBody.addMarker (new Point3d (0.026, 0, -0.0225));
10
11      // create a cylindrical mesh around the joint as a visualization aid to
12      // see how well the via points "wrap" as the lower bone moves
13      PolygonalMesh mesh = MeshFactory.createCylinder (
14         /*rad=*/0.0075, /*h=*/0.04, /*nsegs=*/32);
15      FixedMeshBody meshBody = new FixedMeshBody (
16         MeshFactory.createCylinder (/*rad=*/0.0075, /*h=*/0.04, /*nsegs=*/32));
17      meshBody.setPose (TJW);
18      mech.addMeshBody (meshBody);
19
20      // create a wrappable muscle using a SimpleAxialMuscle material
21      MultiPointSpring muscle = new MultiPointMuscle ("muscle");
22      muscle.setMaterial (
23         new SimpleAxialMuscle (/*k=*/0.5, /*d=*/0, /*maxf=*/0.04));
24      muscle.addPoint (origin);
25      // add via points to the muscle
26      muscle.addPoint (via1);
27      muscle.addPoint (via2);
28      muscle.addPoint (via3);
29      muscle.addPoint (insertion);
30      mech.addMultiPointSpring (muscle);
31
32      // create control panel to allow frameBlending to be set
33      ControlPanel panel = new ControlPanel();
34      panel.addWidget (skinBody, "frameBlending");
35      addControlPanel (panel);
36
37      // set render properties
38      RenderProps.setSphericalPoints (mech, 0.002, Color.BLUE);
39      RenderProps.setSphericalPoints (skinBody, 0.002, Color.WHITE);
40      RenderProps.setCylindricalLines (muscle, 0.001, Color.RED);
41      RenderProps.setFaceColor (meshBody, new Color (200, 200, 230));
```

First, a `SkinMeshBody` is created referencing the proximal and distal bones as master bodies, with the frameBlending property set to `DUAL_QUATERNION_LINEAR` (lines 1-6). Next, we create a set of three via points that will be attached to the skin body to guide the muscle around the joint in lieu of making it actually wrap around a cylinder (lines 7-9).

In the original `PhalanxWrapping` demo, a RigidCylinder was used as a muscle wrap surface. In this demo, we replace this with a simple cylindrical mesh which has no dynamic function but allows us to visualize the wrapping behavior of the via points (lines 11-18). The muscle itself is created using the three via points but with no wrappable segments or bodies (lines 20-30).

A control panel is added to allow for the adjustment of the skin body's frameBlending property (lines 32-35). Finally, render properties are set as for the original demo, only with the skin body markers rendered as white spheres to make them more visible (lines 37-41).

To run this example in ArtiSynth, select All demos > tutorial > PhalanxSkinWrapping from the Models menu. The model should load and initially appear as in Figure 8.7 (left). The pull tool can then be used to move the distal joint while simulating, to illustrate how well the via points "wrap" around the joint, using the cylindrical mesh as a visual reference (Figure 8.7, middle). Changing the frameBlending property to `LINEAR` results in a less satisfactory behavior (Figure 8.7, right).

Chapter 9

DICOM Images

Some models are derived from image data, and it may be useful to show the model and image in the same space. For this purpose, a DICOM image widget has been designed, capable of displaying 3D DICOM volumes as a set of three perpendicular planes. An example widget and its property panel is shown in Figure 9.1.

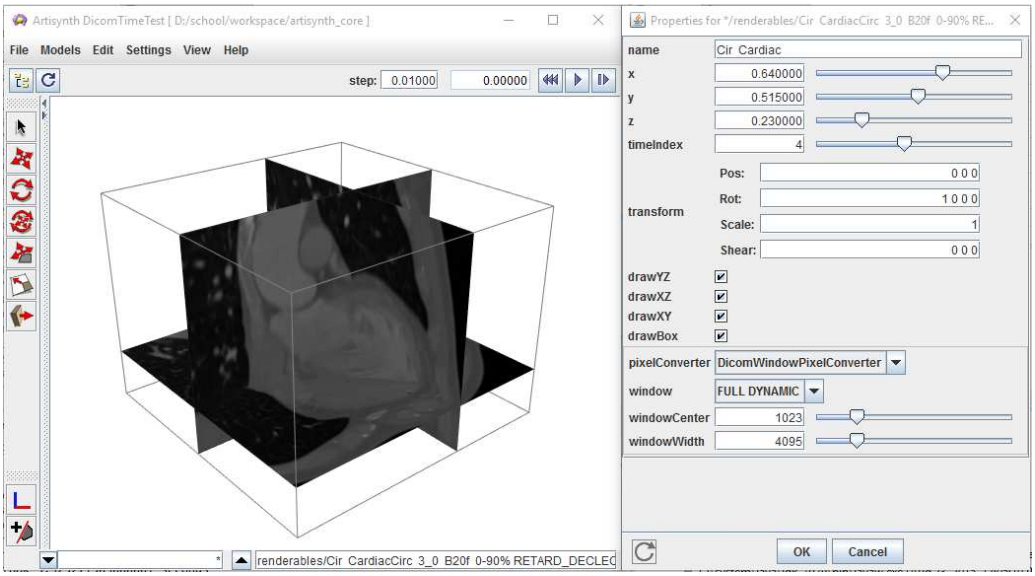


Figure 9.1: DICOM image of the heart, downloaded from http://www.osirix-viewer.com.

The main classes related to the reading and displaying of DICOM images are:

DicomElement
: Describes a single attribute in a DICOM file.

DicomHeader
: Contains all header attributes (all but the image data) extracted from a DICOM file.

DicomPixelBuffer
: Contains the *decoded* image pixels for a single image frame.

DicomSlice
: Contains both the header and image information for a single 2D DICOM slice.

DicomImage
: Container for DICOM slices, creating a 3D volume (or 3D + time)

DicomReader
: Parses DICOM files and folders, appending information to a DicomImage.

DicomViewer
: Displays the DicomImage in the viewer.

If the purpose is simply to display a DICOM volume in the ArtiSynth viewer, then only the last three classes will be of interest. Readers who simply want to display a DICOM image in their model can skip to Section 9.3.

9.1 The DICOM file format

For a complete description of the DICOM format, see the specification page at http://medical.nema.org/standard.html. A brief description is provided here. Another excellent resource is the blog by Roni Zaharia: http://dicomiseasy.blogspot.ca/.

Each DICOM file contains a number of concatenated attributes (a.k.a. elements), one of which defines the embedded binary image pixel data. The other attributes act as meta-data, which can contain identity information of the subject, equipment settings when the image was acquired, spatial and temporal properties of the acquisition, voxel spacings, etc.... The image data typically represents one or more 2D images, concatenated, representing slices (or 'frames') of a 3D volume whose locations are described by the meta-data. This image data can be a set of raw pixel values, or can be encoded using almost any image-encoding scheme (e.g. JPEG, TIFF, PNG). For medical applications, the image data is typically either raw or compressed using a lossless encoding technique. Complete DICOM acquisitions are typically separated into multiple files, each defining one or few frames. The frames can then be assembled into 3D image 'stacks' based on the meta-information, and converted into a form appropriate for display.

Table 9.1: A selection of Value Representations

VR	Description
CS	Code String
DS	Decimal String
DT	Date Time
IS	Integer String
OB	Other Byte String
OF	Other Float String
OW	Other Word String
SH	Short String
UI	Unique Identifier
US	Unsigned Short
OX	One of OB, OW, OF

Each DICOM attribute is composed of:

- a standardized unique integer *tag* in the format (XXXX,XXXX) that defines the *group* and *element* of the attribute
- a *value representation* (VR) that describes the data type and format of the attribute's value (see Table 9.1)
- a *value length* that defines the length in bytes of the attribute's value to follow
- a *value field* that contains the attribute's value

This layout is depicted in Figure 9.2. A list of important attributes are provided in Table 9.2.

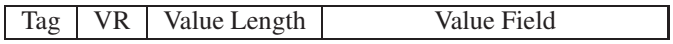


Figure 9.2: DICOM attribute structure

9.2 The DICOM classes

Each DicomElement represents a single attribute contained in a DICOM file. The DicomHeader contains the collection of `DicomElements` defined in a file, apart from the pixel data. The image pixels are decoded and stored in a DicomPixelBuffer. Each DicomSlice contains a `DicomHeader`, as well as the decoded `DicomPixelBuffer` for a single slice (or 'frame'). All slices are assembled into a single DicomImage, which can be used to extract 3D voxels and spatial locations from the set of slices. These five classes are described in further detail in the following sections.

9.2.1 `DicomElement`

The DicomElement class is a simple container for DICOM attribute information. It has three main properties:

- an integer *tag*
- a *value representation* (VR)
- a value

Table 9.2: A selection of useful DICOM attributes

Attribute name	VR	Tag
Transfer syntax UID	UI	0x0002, 0x0010
Slice thickness	DS	0x0018, 0x0050
Spacing between slices	DS	0x0018, 0x0088
Study ID	SH	0x0020, 0x0010
Series number	IS	0x0020, 0x0011
Aquisition number	IS	0x0020, 0x0012
Image number	IS	0x0020, 0x0013
Image position patient	DS	0x0020, 0x0032
Image orientation patient	DS	0x0020, 0x0037
Temporal position identifier	IS	0x0020, 0x0100
Number of temporal positions	IS	0x0020, 0x0105
Slice location	DS	0x0020, 0x1041
Samples per pixel	US	0x0028, 0x0002
Photometric interpretation	CS	0x0028, 0x0004
Planar configuration (color)	US	0x0028, 0x0006
Number of frames	IS	0x0028, 0x0008
Rows	US	0x0028, 0x0010
Columns	US	0x0028, 0x0011
Pixel spacing	DS	0x0028, 0x0030
Bits allocated	US	0x0028, 0x0100
Bits stored	US	0x0028, 0x0101
High bit	US	0x0028, 0x0102
Pixel representation	US	0x0028, 0x0103
Pixel data	OX	0x7FE0, 0x0010

These properties can be obtained using the corresponding `get` function: `getTag()`, `getVR()`, `getValue()`. The tag refers to the concatenated group/element tag. For example, the *transfer syntax UID* which corresponds to group 0x0002 and element 0x0010 has a numeric tag of 0x00020010. The VR is represented by an enumerated type, DicomElement.VR. The 'value' is the *raw* value extracted from the DICOM file. In most cases, this will be a `String`. For raw numeric values (i.e. stored in the DICOM file in binary form) such as the unsigned short (US), the 'value' property is exactly the numeric value.

For VRs such as the integer string (IS) or decimal string (DS), the string will still need to be parsed in order to extract the appropriate sequence of numeric values. There are static utility functions for handling this within `DicomElement`. For a 'best-guess' of the desired parsed value based on the VR, one can use the method `getParsedValue()`. Often, however, the desired value is also context-dependent, so the user should know a priori what type of value(s) to expect. Parsing can also be done automatically by querying for values directly through the `DicomHeader` object.

9.2.2 `DicomHeader`

When a DICOM file is parsed, all meta-data (attributes apart from the actual pixel data) is assembled into a DicomHeader object. This essentially acts as a map that can be queried for attributes using one of the following methods:

```
DicomElement getElement(int tag);                // includes VR and data
String getStringValue(int tag);                  // all non-numeric VRs
String[] getMultiStringValue(int tag);           // UT, SH
int getIntValue(int tag, int defaultValue);      // IS, DS, SL, UL, SS, US
int[] getMultiIntValue(int tag);                 // IS, DS, SL, UL, SS, US
double getDecimalValue(int tag, double defaultValue);    // DS, FL, FD
double[] getMultiDecimalValue(int tag);          // DS, FL, FD
VectorNd getVectorValue(int tag);                // DS, IS, SL, UL, SS, US, FL, FD
DicomDateTime getDateTime(int tag);              // DT, DA, TM
```

The first method returns the full element as described in the previous section. The remaining methods are used for convenience when the desired value type is known for the given tag. These methods automatically parse or convert the `DicomElement`'s value to the desired form.

If the tag does not exist in the header, then the `getIntValue(...)` and `getDecimalValue(...)` will return the supplied `defaultValue`. All other methods will return `null`.

9.2.3 DicomPixelBuffer

The DicomPixelBuffer contains the decoded image information of an image slice. There are three possible pixel types currently supported:

- byte grayscale values (`PixelType.BYTE`)
- short grayscale values (`PixelType.SHORT`)
- byte RGB values, with layout `RGBRGB...RGB` (`PixelType.BYTE_RGB`)

The pixel buffer stores all pixels in one of these types. The pixels can be queried for directly using `getPixel(idx)` to get a single pixel, or `getBuffer()` to get the entire pixel buffer. Alternatively, a DicomPixelInterpolator object can be passed in to convert between pixel types via one of the following methods:

```
public int getPixelsByte (
   int x, int dx, int nx, byte[] pixels, int offset, DicomPixelInterpolator interp);

public int getPixelsShort (
   int x, int dx, int nx, short[] pixels, int offset, DicomPixelInterpolator interp);

public int getPixelsRGB (
   int x, int dx, int nx, byte[] pixels, int offset, DicomPixelInterpolator interp);

public int getPixels(
   int x, int dx, int nx, DicomPixelBuffer pixels, int offset,
   DicomPixelInterpolator interp);
```

These methods populate an output array or buffer with converted pixel values, which can later be passed to a renderer. For further details on these methods, refer to the Javadoc documentation.

9.2.4 DicomSlice

A single DICOM file contains both header information, and one or more image 'frames' (slices). In ArtiSynth, we separate each frame and attach them to the corresponding header information in a DicomSlice. Thus, each slice contains a single `DicomHeader` and `DicomPixelBuffer`. These can be obtained using the methods: `getHeader()` and `getPixelBuffer()`.

For convenience, the `DicomSlice` also has all the same methods for extracting and converting between pixel types as the `DicomPixelBuffer`.

9.2.5 DicomImage

An complete DICOM acquisition typically consists of multiple slices forming a 3D image stack, and potentially contains multiple 3D stacks to form a dynamic 3D+time image. The collection of `DicomSlices` are thus assembled into a DicomImage, which keeps track of the spatial and temporal positions.

The `DicomImage` is the main object to query for pixels in 3D(+time). To access pixels, it has the following methods:

```
public int getPixelsByte (
   int x, int y, int z, int dx, int dy, int dz, int nx, int ny, int nz,
   int time, byte[] pixels, DicomPixelInterpolator interp);

public int getPixelsShort (
   int x, int y, int z, int dx, int dy, int dz, int nx, int ny, int nz,
   int time, short[] pixels, DicomPixelInterpolator interp);

public int getPixelsRGB (
   int x, int y, int z, int dx, int dy, int dz, int nx, int ny, int nz,
   int time, byte[] pixels, DicomPixelInterpolator interp);
```

```
public int getPixels (
   int x, int y, int z, int dx, int dy, int dz, int nx, int ny, int nz,
   int time, DicomPixelBuffer pixels, DicomPixelInterpolator interp);
```

The inputs {x, y, z} refer to voxel indices, and time refers to the time instance index, starting at zero. The four voxel dimensions of the image can be queried with: getNumCols() getNumRows(), getNumSlices(), and getNumTimes().

The DicomImage also contains spatial transform information for converting between voxel indices and patient-centered spatial locations. The affine transform can be acquired with the method getPixelTransform(). This allows the image to be placed in the appropriate 3D location, to correspond with any derived data such as segmentations. The spatial transformation is automatically extracted from the DICOM header information embedded in the files.

9.3 Loading a DicomImage

DICOM files and folders are read using the DicomReader class. The reader populates a supplied DicomImage with slices, forming the full 3D(+time) image. The basic pattern is as follows:

```
String DICOM_directory = ...                // define directory of interest
DicomReader reader = new DicomReader();     // create a new reader

// read all files in a directory, returning a newly constructed image
DicomImage image = reader.read(null, DICOM_directory);
```

The first argument in the read(...) command is an existing image in which to append slices. In this case, we pass in null to signal that a new image is to be created.

In some cases, we might wish to exclude certain files, such as meta-data files that happen to be in the DICOM folder. By default, the reader attempts to read all files in a given directory, and will print out an error message for those it fails to detect as being in a valid DICOM format. To limit the files to be considered, we allow the specification of a Java Pattern, which will test each filename against a regular expression. Only files with names that match the pattern will be included. For example, in the following, we limit the reader to files ending with the "dcm" extension.

```
String DICOM_directory = ...                // define directory of interest
DicomReader reader = new DicomReader();     // create a new reader
Pattern dcmPattern = Pattern.compile(".*\\.dcm") ;  // files ending with .dcm

// read all files in a directory, returning a newly constructed image
DicomImage image = reader.read(null, DICOM_directory, dcmPattern, /*subdirs*/ false);
```

The pattern is applied to the absolute filename, with either windows and mac/linux file separators (both are checked against the regular expression). The method also has an option to recursively search for files in subdirectories. If the full list of files is known, then one can use the method:

```
public DicomImage read(DicomImage im, List<File> files);
```

which will load all specified files.

9.3.1 Time-dependent images

In most cases, time-dependent images will be properly assembled using the previously mentioned methods in the DicomReader. Each slice *should* have a temporal position identifier that allows for the separate image stacks to be separated. However, we have found in practice that at times, the temporal position identifier is omitted. Instead, each stack might be stored in a separate DICOM folder. For this reason, additional read methods have been added that allow manual specification of the time index:

```
public DicomImage read(DicomImage im, List<File> files, int temporalPosition);
public DicomImage read(DicomImage im, String directory, Pattern filePattern,
                       boolean checkSubdirectories, int temporalPosition);
```

If the supplied `temporalPosition` is non-negative, then the temporal position of all included files will be manually set to that value. If negative, then the method will attempt to read the temporal position from the DICOM header information. If no such information is available, then the reader will guess the temporal position to be one past the last temporal position in the original image stack (or 0 if `im == null`). For example, if the original image has temporal positions {0, 1, 2}, then all appended slices will have a temporal position of three.

9.3.2 Image formats

The `DicomReader` attempts to automatically decode any pixel information embedded in the DICOM files. Unfortunately, there are virtually an unlimited number of image formats allowed in DICOM, so there is no way to include native support to decode all of them. By default, the reader can handle raw pixels, and any image format supported by Java's `ImageIO` framework, which includes JPEG, PNG, BMP, WBMP, and GIF. Many medical images, however, rely on lossless or near-lossless encoding, such as lossless JPEG, JPEG 2000, or TIFF. For these formats, we provide an interface that interacts with the third-party command-line utilities provided by **ImageMagick** (`http://www.imagemagick.org`). To enable this interface, the **ImageMagick** utilities `identify` and `convert` must be available and exist somewhere on the system's `PATH` environment variable.

> **ImageMagick Installation**
>
> To enable ImageMagick decoding, required for image formats not natively supported by Java (e.g. JPEG 2000, TIFF), download and install the ImageMagick command-line utilities from: `http://www.imagemagick.org/script/binary-releases.php`
>
> The install path must also be added to your system's `PATH` environment variable so that ArtiSynth can locate the `identify` and `convert` utilities.

9.4 The `DicomViewer`

Once a `DicomImage` is loaded, it can be displayed in a model by using the DicomViewer component. The viewer has several key properties:

name
: the name of the viewer component

x, y, z
: the *normalized* slice positions, in the range [0,1], at which to display image planes

timeIndex
: the temporal position (image stack) to display

transform
: an affine transformation to apply to the image (on top of the voxel-to-spatial transform extracted from the DICOM file)

drawYZ
: draw the YZ plane, corresponding to position `x`

drawXZ
: draw the XZ plane, corresponding to position `y`

drawXY
: draw the XY plane, corresponding to position `z`

drawBox
: draw the 3D image's bounding box

pixelConverter

the interpolator responsible for converting pixels decoded in the DICOM slices into values appropriate for display. The converter has additional properties:

window
: name of a preset window for linear interpolation of intensities

center
: center intensity

width
: width of window

Each property has a corresponding `getXxx(...)` and `setXxx(...)` method that can adjust the settings in code. They can also be modified directly in the ArtiSynth GUI. The last property, the `pixelConverter` allows for shifting and scaling intensity values for display. By default a set of intensity 'windows' are loaded directly from the DICOM file. Each window has a name, and defines a center and width used for linearly scale the intensity range. In addition to the windows extracted from the DICOM, two new windows are added: `FULL_DYNAMIC`, corresponding to the entire intensity range of the image; and `CUSTOM`, which allows for custom specification of the window center and width properties.

To add a `DicomViewer` to the model, create the viewer by supplying a component name and reference to a `DicomImage`, then add it as a `Renderable` to the `RootModel`:

```
DicomViewer viewer = new DicomViewer("my image", dicomImage);
addRenderable(viewer);
```

The image will automatically be displayed in the patient-centered coordinates loaded from the `DicomImage`. In addition to this basic construction, there are convenience constructors to avoid the need for a `DicomReader` for simple DICOM files:

```
// loads all matching DICOM files to create a new image
public DicomViewer(String name, String imagePath, Pattern filePattern, boolean ↩
    checkSubdirs);
// loads a list of DICOM files to create a new image
public DicomViewer(String name, List<File> files);
```

These constructors generate a new `DicomImage` internal to the viewer. The image can be retrieved from the viewer using the `getImage()` method.

9.5 DICOM example

Some examples of DICOM use can be found in the `artisynth.core.demos.dicom` package. The model `DicomTest` loads a partial image of a heart, which is initially downloaded from the ArtiSynth website:

```
1  package artisynth.demos.dicom;
2
3  import java.awt.Color;
4  import java.io.File;
5  import java.io.IOException;
6
7  import artisynth.core.renderables.DicomViewer;
8  import artisynth.core.workspace.DriverInterface;
9  import artisynth.core.workspace.RootModel;
10 import maspack.fileutil.FileManager;
11 import maspack.util.PathFinder;
12
13 public class DicomTest extends RootModel {
14
15    // Dicom file name and URL from which to load it
16    String dicom_file = "MR-MONO2-8-16x-heart";
17    String dicom_url =
18       "http://www.artisynth.org/files/data/dicom/MR-MONO2-8-16x-heart.gz";
```

```
19
20    public void build(String[] args) throws IOException {
21
22       // cache image in a local directory 'data' beneath Java source
23       String localDir = PathFinder.getSourceRelativePath(
24          this, "data/MONO2_HEART");
25       // create a file manager to get the file and download it if necessary
26       FileManager fileManager = new FileManager(localDir, "gz:"+dicom_url+"!/");
27       fileManager.setConsoleProgressPrinting(true);
28       fileManager.setOptions(FileManager.DOWNLOAD_ZIP); // download zip file first
29
30       // get the file from local directory, downloading first if needed
31       File dicomPath = fileManager.get(dicom_file);
32
33       // create a DicomViewer for the file
34       DicomViewer dcp = new DicomViewer("Heart", dicomPath.getAbsolutePath(),
35          null, /*check subdirectories*/false);
36
37       addRenderable(dcp); // add it to root model's list of renderable
38    }
39
```

Lines 23-28 are responsible for downloading and extracting the sample DICOM zip file. In the end, `dicomPath` contains a reference to the desired DICOM file on the local system, which is used to create a viewer on line 34. We then add the viewer to the model for display purposes.

To run this example in ArtiSynth, select All demos > dicom > DicomTest from the Models menu. The model should load and initially appear as in Figure 9.3.

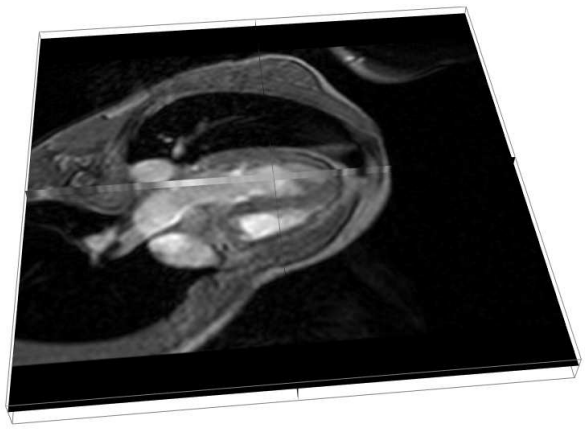


Figure 9.3: DICOM viewer image from `DicomTest`

.

Appendix A

Mathematical Review

This appendix reviews some of the mathematical concepts used in this manual.

A.1 Rotation transforms

Rotation matrices are used to describe the orientation of 3D coordinate frames in space, and to transform vectors between these coordinate frames.

Consider two 3D coordinate frames A and B that are rotated with respect to each other (Figure A.1). The orientation of B with respect to A can be described by a 3×3 rotation matrix $\mathbf{R}_{BA}$, whose columns are the unit vectors giving the directions of the rotated axes $\mathbf{x}'$, $\mathbf{y}'$, and $\mathbf{z}'$ of B with respect to A.

$\mathbf{R}_{BA}$ is an *orthogonal* matrix, meaning that its columns are both perpendicular and mutually orthogonal, so that

$$\mathbf{R}_{BA}^T \mathbf{R}_{BA} = \mathbf{I} \tag{A.1}$$

where $\mathbf{I}$ is the 3×3 identity matrix. The inverse of $\mathbf{R}_{BA}$ is hence equal to its transpose:

$$\mathbf{R}_{BA}^{-1} = \mathbf{R}_{BA}^T. \tag{A.2}$$

Because $\mathbf{R}_{BA}$ is orthogonal, $|\det \mathbf{R}_{BA}| = 1$, and because it is a rotation, $\det \mathbf{R}_{BA} = 1$ (the other case, where $\det \mathbf{R}_{BA} = -1$, is not a rotation but a *reflection*). The 6 orthogonality constraints associated with a rotation matrix mean that in spite of having 9 numbers, the matrix only has 3 degrees of freedom.

Now, assume we have a 3D vector $\mathbf{v}$, and consider its coordinates with respect to both frames A and B. Where necessary, we use a preceding superscript to indicate the coordinate frame with respect to which a quantity is described, so that ${}^A\mathbf{v}$ and ${}^B\mathbf{v}$ and denote $\mathbf{v}$ with respect to frames A and B, respectively. Given the definition of $\mathbf{R}_{AB}$ given above, it is fairly straightforward to show that

$${}^A\mathbf{v} = \mathbf{R}_{BA}\, {}^B\mathbf{v} \tag{A.3}$$

and, given (A.2), that

$${}^B\mathbf{v} = \mathbf{R}_{BA}^T\, {}^A\mathbf{v}. \tag{A.4}$$

Hence in addition to describing the orientation of B with respect to A, $\mathbf{R}_{BA}$ is also a transformation matrix that maps vectors in B to vectors in A.

It is straightforward to show that

$$\mathbf{R}_{BA}^{-1} = \mathbf{R}_{BA}^T = \mathbf{R}_{AB}. \tag{A.5}$$

A simple rotation by an angle θ about one of the basic coordinate axes is known as a *basic* rotation. The three basic rotations about x, y, and z are:

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix},$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix},$$

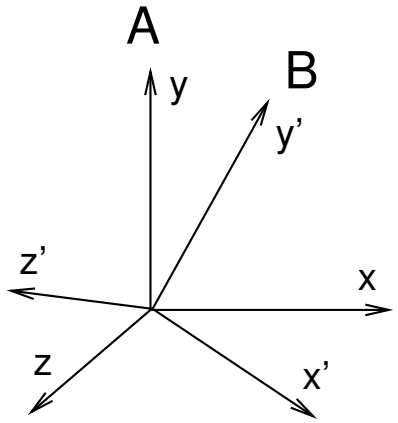


Figure A.1: Two coordinate frames A and B rotated with respect to each other.

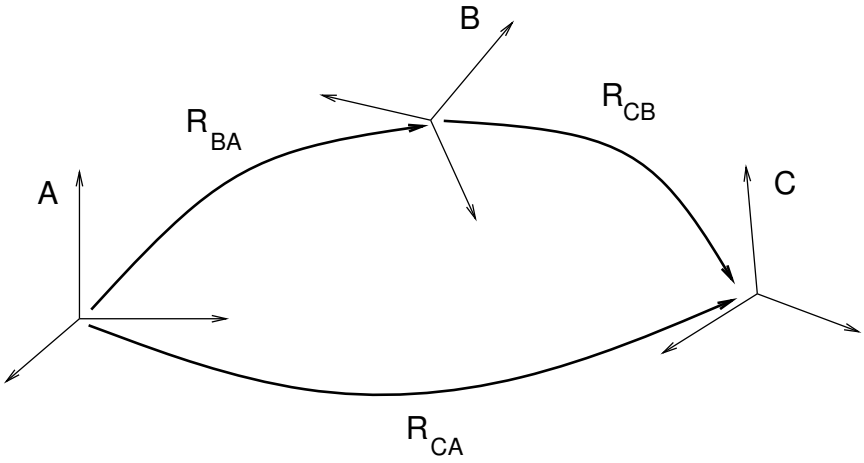


Figure A.2: Schematic illustration of three coordinate frames A, B, and C and the rotational transforms relating them.

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Next, we consider transform composition. Suppose we have three coordinate frames, A, B, and C, whose orientation are related to each other by $\mathbf{R}_{BA}$, $\mathbf{R}_{CB}$, and $\mathbf{R}_{CA}$ (Figure A.6). If we know $\mathbf{R}_{BA}$ and $\mathbf{R}_{CA}$, then we can determine $\mathbf{R}_{CB}$ from

$$\mathbf{R}_{CB} = \mathbf{R}_{BA}^{-1}\mathbf{R}_{CA}. \tag{A.6}$$

This can be understood in terms of vector transforms. $\mathbf{R}_{CB}$ transforms a vector from C to B, which is equivalent to first transforming from C to A,

$${}^{A}\mathbf{v} = \mathbf{R}_{CA}\,{}^{C}\mathbf{v}, \tag{A.7}$$

and then transforming from A to B:

$${}^{B}\mathbf{v} = \mathbf{R}_{BA}^{-1}\,{}^{A}\mathbf{v} = \mathbf{R}_{BA}^{-1}\,\mathbf{R}_{CA}\,{}^{C}\mathbf{v} = \mathbf{R}_{CB}\,{}^{C}\mathbf{v}. \tag{A.8}$$

Note also from (A.5) that $\mathbf{R}_{CB}$ can be expressed as

$$\mathbf{R}_{CB} = \mathbf{R}_{AB}\mathbf{R}_{CA}. \tag{A.9}$$

In addition to specifying rotation matrix components explicitly, there are numerous other ways to describe a rotation. Three of the most common are:

Roll-pitch-yaw angles

There are 6 variations of roll-pitch-yaw angles. The one used in ArtiSynth corresponds to older robotics texts (e.g., Paul, Spong) and consists of a roll rotation r about the z axis, followed by a pitch rotation p about the new y axis, followed by a yaw rotation y about the new x axis. The net rotation can be expressed by the following product of basic rotations: $\mathbf{R}_z(r)\,\mathbf{R}_y(p)\,\mathbf{R}_x(y)$.

Axis-angle

An axis angle rotation parameterizes a rotation as a rotation by an angle θ about a specific axis $\mathbf{u}$. Any rotation can be represented in such a way as a consequence of Euler's rotation theorem.

Euler angles

There are 6 variations of Euler angles. The one used in ArtiSynth consists of a rotation ϕ about the z axis, followed by a rotation θ about the new y axis, followed by a rotation ψ about the new z axis. The net rotation can be expressed by the following product of basic rotations: $\mathbf{R}_z(\phi)\,\mathbf{R}_y(\theta)\,\mathbf{R}_z(\psi)$.

A.2 Rigid transforms

Rigid transforms are used to specify both the transformation of points and vectors between coordinate frames, as well as the relative position and orientation between coordinate frames.

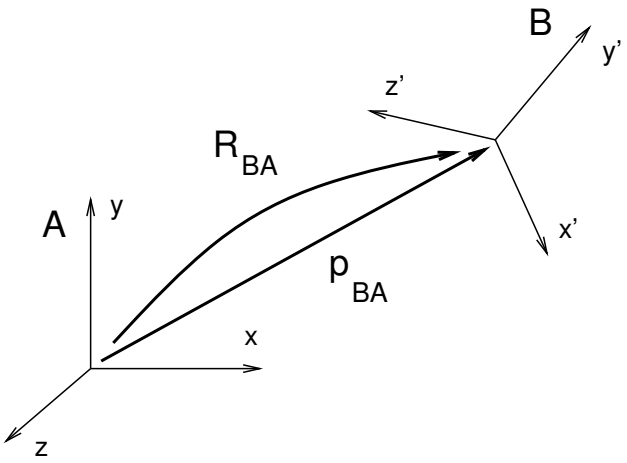


Figure A.3: A position vector $\mathbf{p}_{BA}$ and rotation matrix $\mathbf{R}_{BA}$ describing the position and orientation of frame B with respect to frame A.

Consider two 3D coordinate frames in space, A and B (Figure A.3). The translational position of B with respect to A can be described by a vector $\mathbf{p}_{BA}$ from the origin of A to the origin of B (described with respect to frame A). Meanwhile, the orientation of B with respect to A can be described by the 3×3 rotation matrix $\mathbf{R}_{BA}$ (Section A.1). The combined position and orientation of B with respect to A is known as the *pose* of B with respect to A.

Now, assume we have a 3D point $\mathbf{q}$, and consider its coordinates with respect to both frames A and B (Figure A.4). Given the pose descriptions given above, it is fairly straightforward to show that

$$ {}^{A}\mathbf{q} = \mathbf{R}_{BA}\,{}^{B}\mathbf{q} + \mathbf{p}_{BA}, \tag{A.10} $$

and, given (A.2), that

$$ {}^{B}\mathbf{q} = \mathbf{R}_{BA}^{T}\,({}^{A}\mathbf{q} - \mathbf{p}_{BA}). \tag{A.11} $$

If we extend our points into a 4D *homogeneous* coordinate space with the fourth coordinate w equal to 1, i.e.,

$$ \mathbf{q}^{*} \equiv \begin{pmatrix} \mathbf{q} \\ 1 \end{pmatrix}, \tag{A.12} $$

then (A.10) and (A.11) can be simplified to

$$ {}^{A}\mathbf{q}^{*} = \mathbf{T}_{BA}\,{}^{B}\mathbf{q}^{*} \quad \text{and} \quad {}^{B}\mathbf{q}^{*} = \mathbf{T}_{BA}^{-1}\,{}^{A}\mathbf{q}^{*} $$

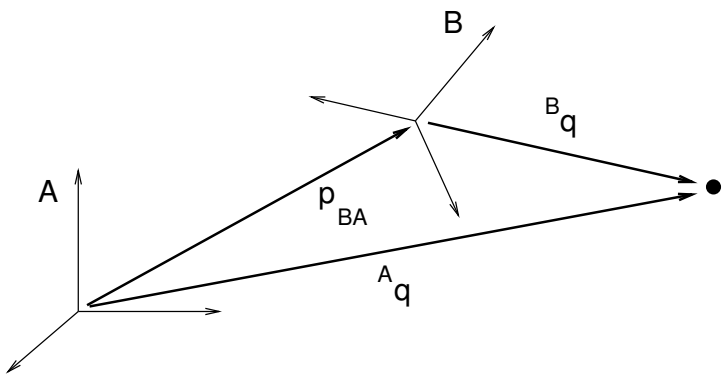


Figure A.4: Point vectors ${}^A\mathbf{q}$ and ${}^B\mathbf{q}$ describing the position of a point $\mathbf{q}$ with respect to frames A and B.

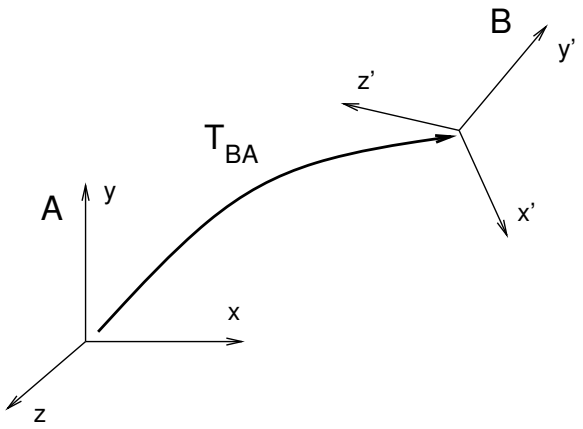


Figure A.5: The transform matrix $\mathbf{T}_{BA}$ from B to A.

where

$$\mathbf{T}_{BA} = \begin{pmatrix} \mathbf{R}_{BA} & \mathbf{p}_{BA} \\ 0 & 1 \end{pmatrix} \tag{A.13}$$

and

$$\mathbf{T}_{BA}^{-1} = \begin{pmatrix} \mathbf{R}_{BA}^T & -\mathbf{R}_{BA}^T \mathbf{p}_{BA} \\ 0 & 1 \end{pmatrix}. \tag{A.14}$$

$\mathbf{T}_{BA}$ is the 4×4 *rigid transform matrix* that transforms points from B to A and also describes the pose of B with respect to A (Figure A.5).

It is straightforward to show that $\mathbf{R}_{BA}^T$ and $-\mathbf{R}_{BA}^T \mathbf{p}_{BA}$ describe the orientation and position of A with respect to B, and so therefore

$$\mathbf{T}_{BA}^{-1} = \mathbf{T}_{AB}. \tag{A.15}$$

Note that if we are transforming a vector $\mathbf{v}$ instead of a point between B and A, then we are only concerned about relative orientation and the vector transforms (A.3) and (A.4) should be used instead. However, we can express these using $\mathbf{T}_{BA}$ if we embed vectors in a homogeneous coordinate space with the fourth coordinate w equal to 0, i.e.,

$$\mathbf{v}^* \equiv \begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix}, \tag{A.16}$$

so that

$${}^B\mathbf{v}^* = \mathbf{T}_{BA}\, {}^A\mathbf{v}^* \quad \text{and} \quad {}^A\mathbf{v}^* = \mathbf{T}_{BA}^{-1}\, {}^B\mathbf{v}^*.$$

Finally, we consider transform composition. Suppose we have three coordinate frames, A, B, and C, each related to the other by transforms $\mathbf{T}_{BA}$, $\mathbf{T}_{CB}$, and $\mathbf{T}_{CA}$ (Figure A.6). Using the same reasoning used to derive (A.6) and (A.9), it is easy to show that

$$\mathbf{T}_{CB} = \mathbf{T}_{BA}^{-1}\, \mathbf{T}_{CA} = \mathbf{T}_{AB}\, \mathbf{T}_{CA}. \tag{A.17}$$

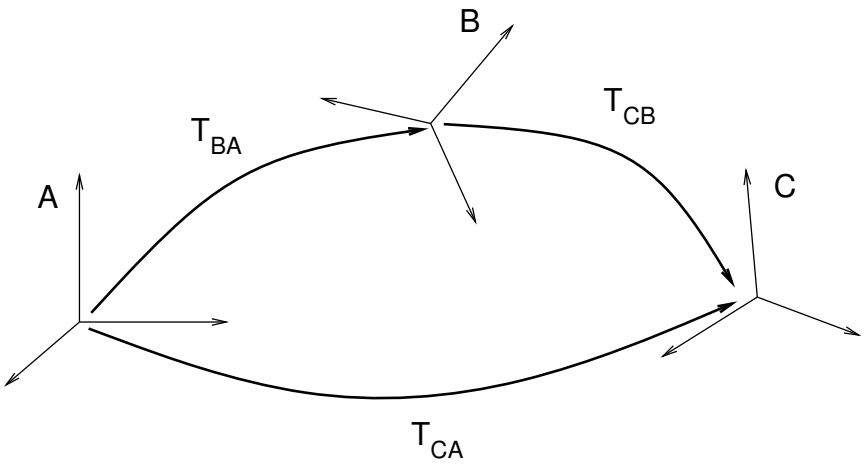


Figure A.6: Three coordinate frames A, B, and C and the transforms relating each one to the other.

A.3 Affine transforms

An *affine transform* is a generalization of a rigid transform, in which the rotational component **R** is replaced by a general 3×3 matrix **A**. This means that an affine transform implements a generalized basis transformation combined with an offset of the origin (Figure A.7). As with **R** for rigid transforms, the columns of **A** still describe the transformed basis vectors $\mathbf{x}'$, $\mathbf{y}'$, and $\mathbf{z}'$, but these are generally no longer orthonormal.

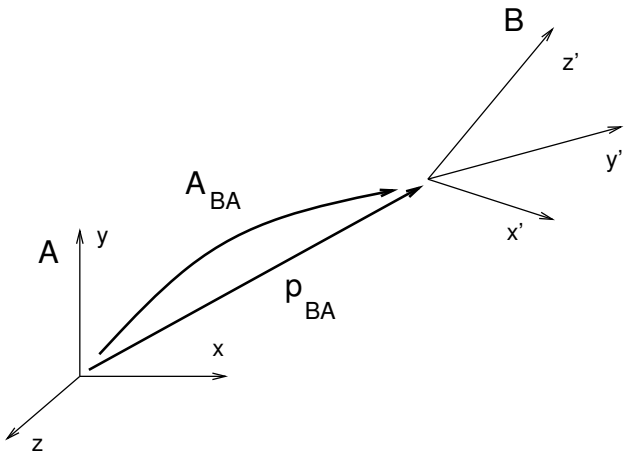


Figure A.7: A position vector $\mathbf{p}_{BA}$ and a general matrix $\mathbf{A}_{BA}$ describing the affine position and basis transform of frame B with respect to frame A.

Expressed in terms of homogeneous coordinates, the affine transform $\mathbf{X}_{AB}$ takes the form

$$\mathbf{X}_{BA} = \begin{pmatrix} \mathbf{A}_{BA} & \mathbf{p}_{BA} \\ 0 & 1 \end{pmatrix} \tag{A.18}$$

with

$$\mathbf{X}_{BA}^{-1} = \begin{pmatrix} \mathbf{A}_{BA}^{-1} & -\mathbf{A}_{BA}^{-1}\mathbf{p}_{BA} \\ 0 & 1 \end{pmatrix}. \tag{A.19}$$

As with rigid transforms, when an affine transform is applied to a vector instead of a point, only the matrix **A** is applied and the translation component **p** is ignored.

Affine transforms are typically used to effect transformations that require stretching and shearing of a coordinate frame. By the polar decomposition theorem, **A** can be factored into a regular rotation **R** plus a symmetric shearing/scaling matrix **P**:

$$\mathbf{A} = \mathbf{R}\mathbf{P} \tag{A.20}$$

Affine transforms can also be used to perform reflections, in which $\mathbf{A}$ is orthogonal (so that $\mathbf{A}^T\mathbf{A} = \mathbf{I}$) but with $\det\mathbf{A} = -1$.

A.4 Rotational velocity

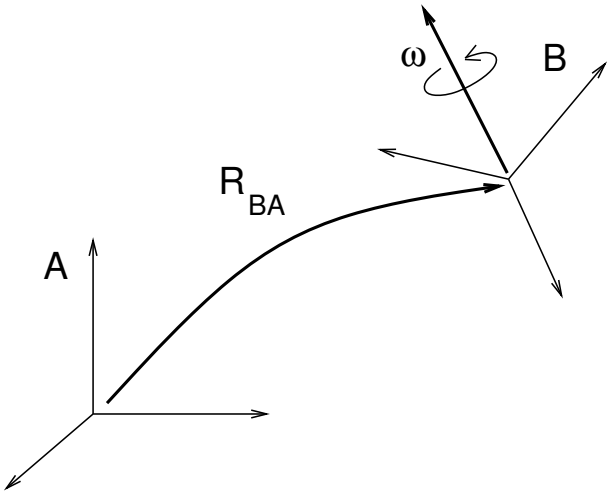


Figure A.8: Frame B rotating with respect to frame A.

Given two 3D coordinate frames A and B, the rotational, or *angular*, velocity of B with respect to A is given by a 3D vector ω_{BA} (Figure A.8). ω_{BA} is related to the derivative of $\mathbf{R}_{BA}$ by

$$\dot{\mathbf{R}}_{BA} = [{}^A\omega_{BA}]\mathbf{R}_{BA} = \mathbf{R}_{BA}[{}^B\omega_{BA}] \tag{A.21}$$

where ${}^A\omega_{BA}$ and ${}^B\omega_{BA}$ indicate ω_{BA} with respect to frames A and B and $[\omega]$ denotes the 3×3 cross product matrix

$$[\omega] \equiv \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}. \tag{A.22}$$

If we consider instead the velocity of A with respect to B, it is straightforward to show that

$$\omega_{AB} = -\omega_{BA}. \tag{A.23}$$

A.5 Spatial velocities and forces

Given two 3D coordinate frames A and B, the *spatial velocity*, or *twist*, $\hat{\mathbf{v}}_{BA}$ of B with respect to A is given by the 6D composition of the translational velocity $\mathbf{v}_{BA}$ of the origin of B with respect to A and the angular velocity ω_{BA}:

$$\hat{\mathbf{v}}_{BA} \equiv \begin{pmatrix} \mathbf{v}_{BA} \\ \omega_{BA} \end{pmatrix}. \tag{A.24}$$

Similarly, the *spatial force*, or *wrench*, $\hat{\mathbf{f}}$ acting on a frame B is given by the 6D composition of the translational force $\mathbf{f}_B$ acting on the frame's origin and the moment τ, or torque, acting through the frame's origin:

$$\hat{\mathbf{f}}_B \equiv \begin{pmatrix} \mathbf{f}_B \\ \tau_B \end{pmatrix}. \tag{A.25}$$

If we have two frames A and B rigidly connected within a rigid body (Figure A.9), and we know the spatial velocity $\hat{\mathbf{v}}_{BC}$ of B with respect to some third frame C, we may wish to know the spatial velocity $\hat{\mathbf{v}}_{AC}$ of A with respect to C. The

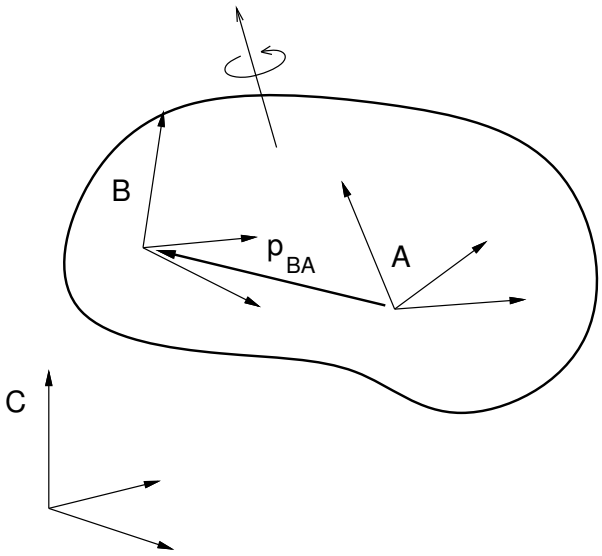


Figure A.9: Two frames A and B rigidly connected within a rigid body and moving with respect to a third frame C.

angular velocity components are the same, but the translational velocity components are coupled by the angular velocity and the offset $\mathbf{p}_{BA}$ between A and B, so that

$$\mathbf{v}_{AC} = \mathbf{v}_{BC} + \mathbf{p}_{BA} \times \omega_{BC}.$$

$\hat{\mathbf{v}}_{AC}$ is hence related to $\hat{\mathbf{v}}_{BC}$ via

$$\begin{pmatrix} \mathbf{v}_{AC} \\ \omega_{AC} \end{pmatrix} = \begin{pmatrix} \mathbf{I} & [\mathbf{p}_{BA}] \\ 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{v}_{BC} \\ \omega_{BC} \end{pmatrix}.$$

where $[\mathbf{p}_{BA}]$ is defined by (A.22).

The above equation assumes that all quantities are expressed with respect to the same coordinate frame. If we instead consider $\hat{\mathbf{v}}_{AC}$ and $\hat{\mathbf{v}}_{BC}$ to be represented in frames A and B, respectively, then we can show that

$$^{A}\hat{\mathbf{v}}_{AC} = \mathbf{X}_{BA}\, ^{B}\hat{\mathbf{v}}_{BC}, \tag{A.26}$$

where

$$\mathbf{X}_{BA} \equiv \begin{pmatrix} \mathbf{R}_{BA} & [\mathbf{p}_{BA}]\mathbf{R}_{BA} \\ 0 & \mathbf{R}_{BA} \end{pmatrix}. \tag{A.27}$$

The transform $\mathbf{X}_{BA}$ is easily formed from the components of the rigid transform $\mathbf{T}_{BA}$ relating B to A.

The spatial forces $\hat{\mathbf{f}}_A$ and $\hat{\mathbf{f}}_B$ acting on frames A and B within a rigid body are related in a similar way, only with spatial forces, it is the moment that is coupled through the moment arm created by $\mathbf{p}_{BA}$, so that

$$\tau_A = \tau_B + \mathbf{p}_{BA} \times \mathbf{f}_B.$$

If we again assume that $\hat{\mathbf{f}}_A$ and $\hat{\mathbf{f}}_B$ are expressed in frames A and B, we can show that

$$^{A}\hat{\mathbf{f}}_A = \mathbf{X}^{*}_{BA}\, ^{B}\hat{\mathbf{f}}_B, \tag{A.28}$$

where

$$\mathbf{X}^{*}_{BA} \equiv \begin{pmatrix} \mathbf{R}_{BA} & 0 \\ [\mathbf{p}_{BA}]\mathbf{R}_{BA} & \mathbf{R}_{BA} \end{pmatrix}. \tag{A.29}$$

A.6 Spatial inertia

Assume we have a rigid body with mass m and a coordinate frame located at the body's center of mass. If $\mathbf{v}$ and ω give the translational and rotational velocity of the coordinate frame, then the body's linear and angular momentum $\mathbf{p}$ and $\mathbf{L}$ are given by

$$\mathbf{p} = m\mathbf{v} \quad \text{and} \quad \mathbf{L} = \mathbf{J}\omega, \tag{A.30}$$

where $\mathbf{J}$ is the 3×3 *rotational inertia* with respect to the center of mass. These relationships can be combined into a single equation

$$\hat{\mathbf{p}} = \mathbf{M}\hat{\mathbf{v}}, \tag{A.31}$$

where $\hat{\mathbf{p}}$ and $\mathbf{M}$ are the *spatial momentum* and *spatial inertia*:

$$\hat{\mathbf{p}} \equiv \begin{pmatrix} \mathbf{p} \\ \mathbf{L} \end{pmatrix}, \qquad \mathbf{M} \equiv \begin{pmatrix} m\mathbf{I} & 0 \\ 0 & \mathbf{J} \end{pmatrix}. \tag{A.32}$$

The spatial momentum satisfies Newton's second law, so that

$$\hat{\mathbf{f}} = \frac{d\hat{\mathbf{p}}}{dt} = \mathbf{M}\frac{d\hat{\mathbf{v}}}{dt} + \dot{\mathbf{M}}\hat{\mathbf{v}}, \tag{A.33}$$

which can be used to find the acceleration of a body in response to a spatial force.

When the body coordinate frame is *not* located at the center of mass, then the spatial inertia assumes the more complicated form

$$\begin{pmatrix} m\mathbf{I} & -m[\mathbf{c}] \\ m[\mathbf{c}] & \mathbf{J} - m[\mathbf{c}][\mathbf{c}] \end{pmatrix}, \tag{A.34}$$

where $\mathbf{c}$ is the center of mass and $[\mathbf{c}]$ is defined by (A.22).

Like the rotational inertia, the spatial inertia is always symmetric positive definite if $m > 0$.

References

[1] Mihai Anitescu and Florian A. Potra. “A Time-Stepping Method for Stiff Multibody Dynamics with Contact and Friction”. In: *International Journal for Numerical Methods in Engineering* 55.7 (2002), pp. 753–784.

[2] Silvia S Blemker and Scott L Delp. “Three-dimensional representation of complex muscle architectures and geometries”. In: *Annals of biomedical engineering* 33.5 (2005), pp. 661–673.

[3] J. Bonet and R. D. Wood. *Nonlinear continuum mechanics for finite element analysis*. Cambridge University Press, 2000.

[4] Scott L Delp et al. “OpenSim: open-source software to create and analyze dynamic simulations of movement”. In: *IEEE T Bio-Med Eng* 54.11 (2007), pp. 1940–1950.

[5] Ladislav Kavan et al. “Geometric skinning with approximate dual quaternion blending”. In: *ACM Transactions on Graphics (TOG)* 27.4 (2008), pp. 1–23.

[6] C. Lacoursière. “Ghosts and machines: regularized variational methods for interactive simulations of multibodies with dry frictional contacts”. PhD thesis. Umeå University, Department of Computing Science, 2007.

[7] Claude Lacoursière. “Ghosts and machines: regularized variational methods for interactive simulations of multibodies with dry frictional contacts”. PhD thesis. Computer Science Dept., Umea University, Sweden, 2007, p. 444.

[8] John E Lloyd, Ian Stavness, and Sidney Fels. “ArtiSynth: A fast interactive biomechanical modeling toolkit combining multibody and finite element simulation”. In: *Soft tissue biomechanical modeling for computer assisted surgery*. Springer, 2012, pp. 355–394.

[9] Steve Maas et al. “FEBio: Finite Elements for Biomechanics”. In: *Journal of biomechanical engineering* 134 (Jan. 2012), p. 011005. DOI: 10.1115/1.4005694.

[10] Steve Maas et al. *FEBio Theory Manual*. https://help.febio.org/FEBio/FEBio_tm_2_7.

[11] Matthew Millard et al. “Flexing computational muscle: modeling and simulation of musculotendon dynamics”. In: *Journal of biomechanical engineering* 135.2 (2013).

[12] Matthieu Nesme et al. “Preserving topology and elasticity for embedded deformable models”. In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 3. ACM. 2009, p. 52.

[13] Wai-Hin Ngan and John Lloyd. “Efficient Deformable Body Simulation using Stiffness-Warped Nonlinear Finite Elements”. In: *Symposium on Interactive 3D Graphics and Games (i3D)*. poster. Feb. 2008.

[14] CC Peck, GEJ Langenbach, and AG Hannam. “Dynamic simulation of muscle and articular properties during human wide jaw opening”. In: *Archives of Oral Biology* 45.11 (2000), pp. 963–982.

[15] Florian A. Potra et al. “A linearly implicit trapezoidal method for integrating stiff multibody dynamics with contact, joints, and friction”. In: *International Journal for Numerical Methods in Engineering* 66.7 (2006), pp. 1079–1124.

[16] M. Servin, C. Lacoursiere, and N. Melin. “Interactive simulation of elastic deformable materials”. In: *Proceedings of SIGRAD Conference 2006 in Skövde, Sweden*. 2006, pp. 22–32.

[17] Ian Stavness et al. “Unified skinning of rigid and deformable models for anatomical simulati on s”. In: *SIGGRAPH Asia 2014 Technical Briefs*. ACM. 2014, p. 9.

[18] Gabriel Taubin. “Curve and surface smoothing without shrinkage”. In: *Computer Vision, 1995. Proceedings., Fifth International Conference on*. IEEE. 1995, pp. 852–857.