ArtiSynth Modeling Guide

John Lloyd and Antonio Sánchez
Last update: Apr 10, 2017

Contents

Preface

This guide describes how to create mechanical and biomechanical models in ArtiSynth using its Java API.

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 preceeding 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.

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.

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.

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.

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.

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 `StepAdjusment` 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.

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.

Physics simulation

Only a brief summary of ArtiSynth physics simulation is described here. Full details are given in [5] 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

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

where $\mathbf{w}$ is a slack variable, λ and $\mathbf{z}$ give the force constraint impulses over the time step, and $\mathbf{g}$ and $\mathbf{n}$ are derivative terms arising if $\mathbf{G}$ and $\mathbf{N}$ are time varying. 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.

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

$$\mathbf{u}_j = -\mathbf{G}_{j\alpha}\mathbf{u}_\alpha \tag{1.7}$$

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

$$\mathbf{u}_j = \mathbf{u}_k \tag{1.8}$$

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

$$\mathbf{u}_j = \mathbf{u}_k - \mathbf{l}_j \times \omega_k \tag{1.9}$$

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

$$\begin{pmatrix} \mathbf{I} & [l_j] \end{pmatrix} \tag{1.10}$$

in the $k-th$ block column, where

$$[l] \equiv \begin{pmatrix} 0 & -l_z & l_y \\ l_z & 0 & -l_x \\ -l_y & l_x & 0 \end{pmatrix} \tag{1.11}$$

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.

Basic packages

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

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
```

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 controller
```

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
```

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.

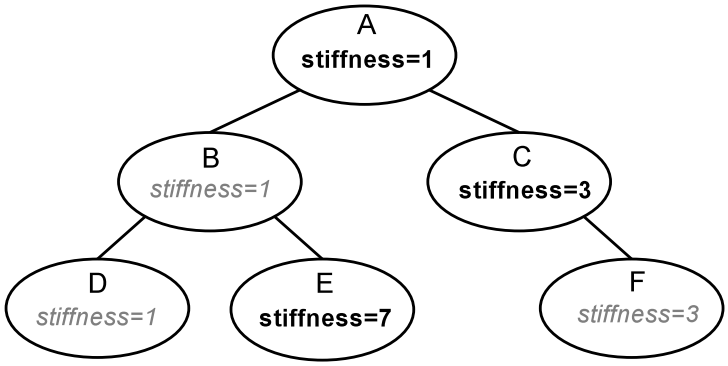


Figure 1.1: Inheritance of a property named *stiffness* among a component hierarchy. Explicit settings are in bold; inherited settings are in gray italic.

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.

Property handles and paths

Each property has a unique name which may be used to obtain a *property handle* through which the property's value may be queried or set for a particular component. 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");
```

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).

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.

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. 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. In cases where the model is specified directly on the ArtiSynth command line (using the -model <classname> option), it is possible to also specify arguments for the build() method. This is done by enclosing the desired arguments within square brackets [ ] immediately following the -model option. So, for example,

```
> artisynth -model projects.MyModel [ -foo 123 ]
```

would cause the strings "-foo" and "123" to be passed to the build() method of MyModel.

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 directory 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 directory tree separate from the source code, with the top-level class directory named classes, located one level below the project root directory. A typical top-level class directory 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 directory structure that mirrors package names, with respect to the sample project directory 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 directories 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 directories to the EXTCLASSPATH file

You can explicitly list class directories in the file EXTCLASSPATH, located in the ArtiSynth root directory (it may be necessary to create this file).

Add the directories 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 directories 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).

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 by class path

A model may be loaded by directly by choosing File > Load from class ... and directly specifying its class name. 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 the Models menu

A faster way to load a model is by selecting it in one of the Models submenus. This may require editing the model menu configuration files.

Loading from a file

If a model has previously been saved to a 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 models 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.

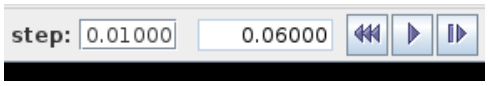


Figure 1.2: The ArtiSynth play controls. From left to right: step size control, current simulation time, and the reset, play/pause, and single-step buttons.

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.

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 formated 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"));
```

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
RigidTransfrom3d(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
RigidTransfrom3d(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.

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.

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.

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().

Mesh creation

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 });
```

However, meshes are more commonly created using either one of the factory methods supplied by MeshFactory, or by reading a definition from a file (Section 2.5.5).

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

```
MeshFactory.createSphere (radius, nslices, nlevels);
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 = MeshFactor.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 = MeshFactor.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);
```

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 analagous 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 simultion 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.

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.

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 explicity 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();
```

Reading and writing mesh files

The package `maspack.geometry.io` supplies a number of classes for writing and reading meshes to and from files of different formats.

Some of the supported formats and their associated readers and writers include:

Extension	Format	Reader/writer classes
.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`

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`.

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.

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.

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.

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, non-linear 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()`.

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 /**
```

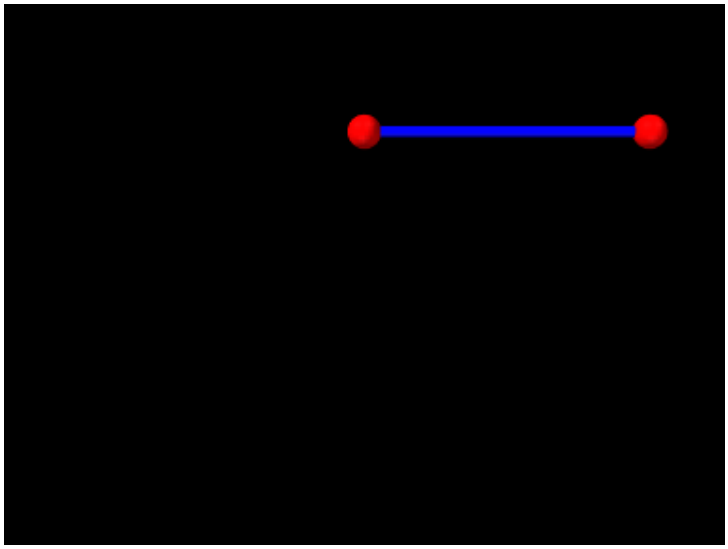


Figure 3.1: ParticleSpring model loaded into ArtiSynth.

```
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 covenience 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.

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 3.5. 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.

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.

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 1.4.2), so that if not set explicitly, it inherits the nearest explicitly set value in an ancestor component.

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.

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);
```

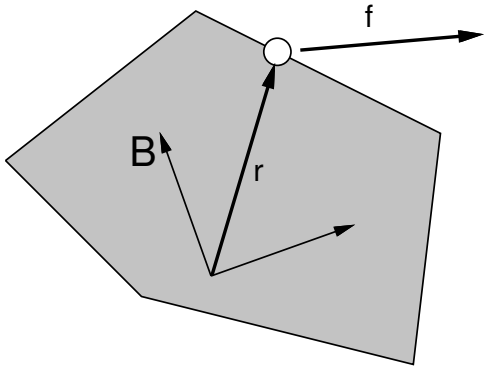


Figure 3.2: A force **f** applied to a frame marker attached to a rigid body. The marker is located at the point **r** with respect to the body coordinate frame B.

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

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

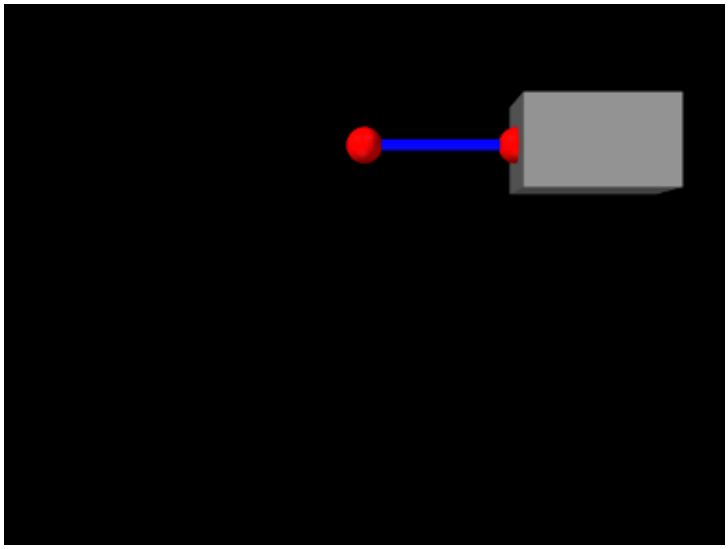


Figure 3.3: RigidBodySpring model loaded into ArtiSynth.

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:

```
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.

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);
```

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.

Inertia and meshes

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

```
setMesh (PolygonalMesh mesh);
setMesh (PolygonalMesh mesh, String meshFileName);
PolygonalMesh getMesh();
```

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.

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 volume defined by the surface mesh to compute the remaining inertial values. How a rigid body's inertia is computed is determined by its `inertiaMethod` property, which can be one

Density

Inertia is computed from density;

Mass

Inertia is computed from mass;

Explicit

Inertia is set explicitly.

This 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 and a density value and set `inertiaMethod` to `Density`, and the method

```
setInertiaFromMass (mass);
```

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

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

```
getDensity();
```

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.

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 joint-type connectors that enforce the connection using hard constraints.

Joints and coordinate frames

Consider two 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 bodies 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 restricts the set of poses that are possible between the two bodies and reduces the degrees of freedom available to $\mathbf{T}_{BA}$. For simplicity, joints have their own coordinate frames for describing their constraining actions, and then these frames are related to the frames A and B of the associated bodies by auxiliary transformations.

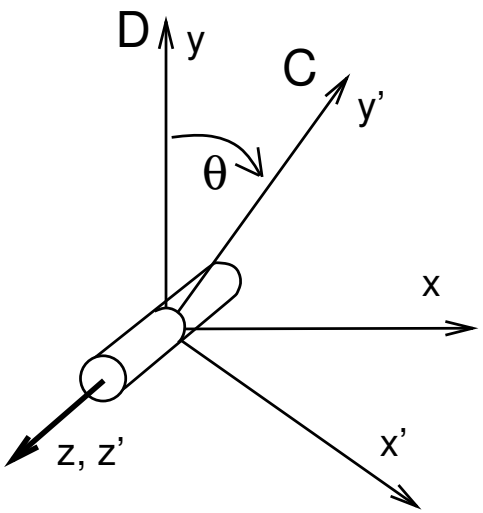


Figure 3.4: Coordinate frames D and C for a revolute joint.

Each joint is associated with two coordinate frames C and D which move with respect to each other as the joint moves. The allowed joint motions therefore correspond to the allowed values of the *joint transform* $\mathbf{T}_{CD}$. D is the *base* frame and C is the *motion* frame. For a revolute joint (Figure 3.4), C can move with respect to D by rotating about the z axis. Other motions are prohibited. $\mathbf{T}_{CD}$ should therefore alway have the form

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

where θ is the angle of joint rotation and is known as the *joint parameter*. Other joints have different parameterizations, with the number of parameters equaling the number of degrees of freedom available to the joint. When $\mathbf{T}_{CD} = \mathbf{I}$ (where $\mathbf{I}$ is the identity transform), the parameters are all (usually) equal to zero, and the joint is said to be in the *zero state*.

In practice, due to numerical errors and/or compliance in the joint, the joint transform $\mathbf{T}_{CD}$ may sometimes deviate from the allowed set of values dictated by the joint type. In ArtiSynth, this is accounted for by introducing an additional *constraint* frame G between D and C (Figure 3.5). G is computed to be the nearest frame to C that lies exactly in the

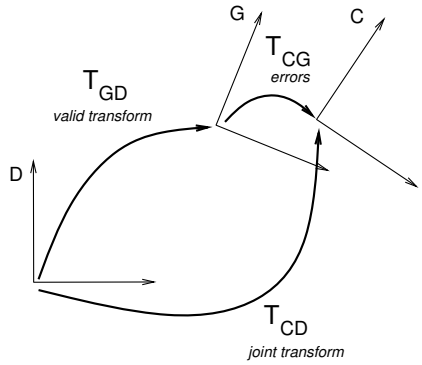


Figure 3.5: 2D schematic showing the joint frames D and C, along with the intermediate frame G that accounts for numeric error and complaint motion.

joint constraint space. $\mathbf{T}_{GD}$ is therefore a valid transform for the joint, $\mathbf{T}_{GC}$ accommodates the error, and the whole joint transform is given by the composition

$$\mathbf{T}_{CD} = \mathbf{T}_{GD}\,\mathbf{T}_{CG}. \tag{3.7}$$

If there is no compliance or joint error, then frames G and C are the same, $\mathbf{T}_{CG} = \mathbf{I}$, and $\mathbf{T}_{CD} = \mathbf{T}_{GD}$.

In general, each joint is attached to two rigid bodies A and B, with frame C being fixed to body A and frame D being fixed to body B. The restrictions of the joint transform $\mathbf{T}_{CD}$ therefore restrict the relative poses of A and B.

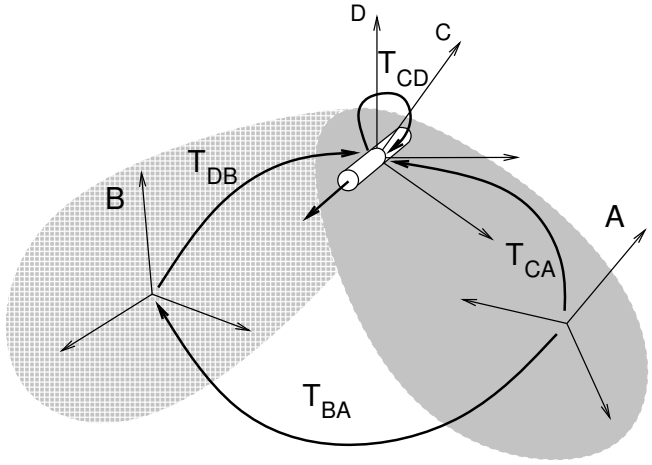


Figure 3.6: Transforms connecting joint coordinate frames C and D with rigid bodies A and B.

Except in special cases, the joint coordinate 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 3.6). Since $\mathbf{T}_{CA}$ and $\mathbf{T}_{DB}$ are both fixed, the pose of B relative to A can be determined from the joint transform $\mathbf{T}_{CD}$:

$$\mathbf{T}_{BA} = \mathbf{T}_{CA}\,\mathbf{T}_{CD}^{-1}\,\mathbf{T}_{DB}^{-1}. \tag{3.8}$$

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

Creating Joints

Joint components in ArtiSynth are implemented by subclasses of BodyConnector. Some of the commonly used ones are described in Section 3.3.4.

An application creates a joint by constructing it and adding it to a `MechModel`. Most joints generally have a constructor of the form

```
JointType (bodyA, bodyB, TDW);
```

which specifies the rigid 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. Then 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

$$\mathbf{T}_{CA} = \mathbf{T}_{AW}^{-1}\, \mathbf{T}_{CW} \tag{3.9}$$

$$\mathbf{T}_{DB} = \mathbf{T}_{BW}^{-1}\, \mathbf{T}_{DW} \tag{3.10}$$

where $\mathbf{T}_{AW}$ and $\mathbf{T}_{BW}$ are the poses of A and B. The same body and transform settings can be made on an existing joint using the method setBodies(bodyA, bodyB, TDW).

Alternatively, if we prefer to explicitly specify $\mathbf{T}_{CA}$ or $\mathbf{T}_{DB}$, then we can determine $\mathbf{T}_{DW}$ from $\mathbf{T}_{AW}$ or $\mathbf{T}_{BW}$ using

$$\mathbf{T}_{DW} = \mathbf{T}_{AW}\, \mathbf{T}_{CA} \tag{3.11}$$

$$\mathbf{T}_{DW} = \mathbf{T}_{BW}\, \mathbf{T}_{DB}. \tag{3.12}$$

For example, if we know $\mathbf{T}_{CA}$, this can be accomplished using the following code fragment:

```
RigidBody bodyA, bodyB;
RigidTransform3d TCA;

... initialize bodyA, bodyB, and TCA ...

RigidTransform3d TDW = new RigidTransform3d();
TDW.mul (bodyA.getPose(), TCA);  // bodyA.getPose() returns TAW
RevoluteJoint joint = new RevoluteJoint (bodyA, bodyB, TDW);
```

Another method, setBodies(bodyA, TCA, bodyB, TDB), allows us to set both values of $\mathbf{T}_{CA}$ or $\mathbf{T}_{BA}$ explicitly. This is useful if the joint transform $\mathbf{T}_{CD}$ is known to be some value *other* than the identity, in which case $\mathbf{T}_{CA}$ or $\mathbf{T}_{BA}$ can be computed from (3.8), where $\mathbf{T}_{BA}$ is given by

$$\mathbf{T}_{BA} = \mathbf{T}_{AW}^{-1}\, \mathbf{T}_{BW}. \tag{3.13}$$

For instance, if we know $\mathbf{T}_{CA}$ and the joint transform $\mathbf{T}_{CD}$, then we can compute $\mathbf{T}_{DB}$ from

$$\mathbf{T}_{DB} = \mathbf{T}_{BA}^{-1}\, \mathbf{T}_{CA}\, \mathbf{T}_{CD}^{-1} = \mathbf{T}_{BW}^{-1}\, \mathbf{T}_{AW}\, \mathbf{T}_{CA}\, \mathbf{T}_{CD}^{-1} \tag{3.14}$$

and set up the joint as follows:

```
RigidBody bodyA, bodyB;
RigidTransform3d TCA, TCD;

... initialize bodyA, bodyB, TCA, TCD ...

RigidTransform3d TBD = new RigidTransform3d();
TDB.mulInverseLeft (bodyB.getPose(), bodyA.getPose());
TDB.mul (TCA);
TDB.mulInverse (TCD);
RevoluteJoint joint = new RevoluteJoint ();
joint.setBodies (bodyA, TCA, bodyB, TDB);
```

Some joint implementations provide the ability to explicitly set the joint parameter(s) after it has been created and added to the `MechModel`, making it easy to "move" the joint to a specific configuration. For example, `RevoluteJoint` provides the method setTheta(). This causes the transform $\mathbf{T}_{CD}$ to be explicitly set to the value implied by the joint parameters, and the pose of either body A or B is changed to accommodate this. Whether body A or B is moved depends on which one is the least connected to "ground", and other bodies that have joint connections to the moved body are moved as well.

If desired, joints can be connected to only a single rigid body. In this case, the second body B is simply assumed to be "ground", and the coordinate frame B is instead taken to be the world coordinate frame W. The corresponding calls to the joint constructor or `setBodies()` take the form

```
JointType joint = new JointType (bodyA, null, TDW);
```

or

```
    JointType joint = new JointType();
    joint.setBodies (bodyA, null, TDW);
```

Example: A simple revolute joint

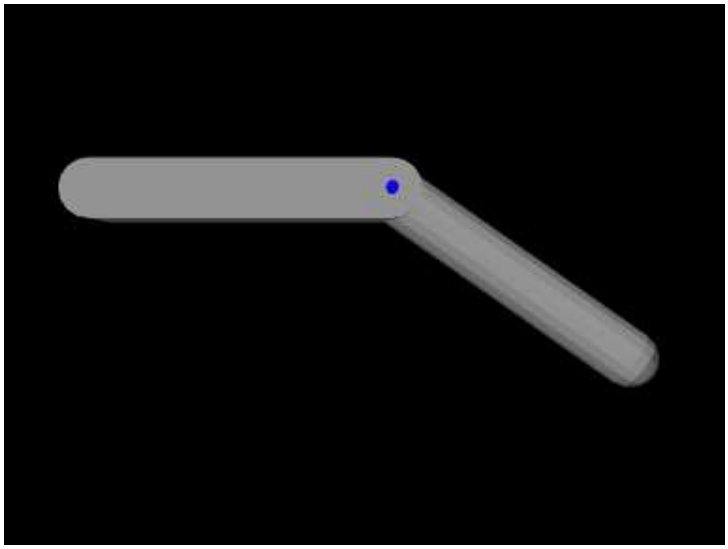


Figure 3.7: RigidBodyJoint model loaded into ArtiSynth.

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);
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        RevoluteJoint joint = new RevoluteJoint (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.setLineRadius (joint, 0.2);
43        joint.setAxisLength (4);
44     }
```

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 revolute joint is implemented using RevoluteJoint, which is constructed at line 32 with the joint coordinate frame D being located in world coordinates by `TDW` as described in Section 3.3.2.

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 freest to move).

Finally, render properties are set starting at line 42. A revolute joint is rendered as a line segment, using the line render properties (Section 4.3), with `lineStyle` and `lineColor` set to `Cylinder` and `BLUE`, respectively, by default. The cylinder radius and length are specified by the line render property `lineRadius` and the revolute joint property `axisLength`, which are set explicitly in the code.

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 3.7. Running the model (Section 1.5.3) will cause `bodyA` to fall and swing under gravity.

Commonly used joints

Some of the joints commonly used by ArtiSynth are shown in Figure 3.8. Each illustration shows the allowed joint motion relative to the base coordinate frame D. Clockwise from the top-left, these joints are:

Revolute joint

A one DOF joint which allows rotation by an angle θ about the z axis.

Roll-pitch joint

A two DOF joint, similar to the revolute joint, which allows the rotation about z to be followed by an additional rotation ϕ about the (new) y axis.

Spherical joint

A three DOF joint in which the origin remains fixed but any orientation may be assumed.

Planar connector

A five DOF joint which connects a point on a single rigid body to a plane in space. The point may slide freely in the x-y plane, and the body may assume any orientation about the point.

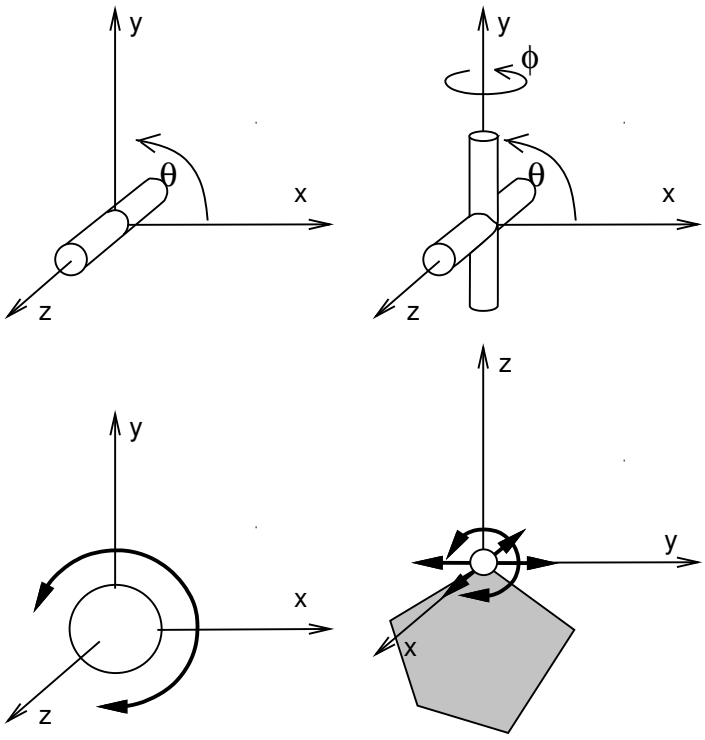


Figure 3.8: Commonly used joints. Clockwise from top left: revolute, roll-pitch, spherical, planer connector.

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.

Frame spring coordinate frames

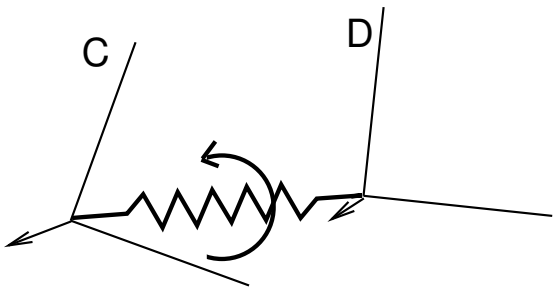


Figure 3.9: A frame spring connecting two coordinate frames D and C.

The basic idea of a frame spring is shown in Figure 3.9. 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

$$\begin{aligned} \mathbf{f}_C &= \mathbf{f}_k(\mathbf{T}_{DC}) + \mathbf{f}_d(\hat{\mathbf{v}}_{DC}) \\ \tau_C &= \tau_k(\mathbf{T}_{DC}) + \tau_d(\hat{\mathbf{v}}_{DC}). \end{aligned} \tag{3.15}$$

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

$$\begin{aligned} \mathbf{f}_D &= -\mathbf{f}_C, \\ \tau_D &= -\tau_C. \end{aligned} \tag{3.16}$$

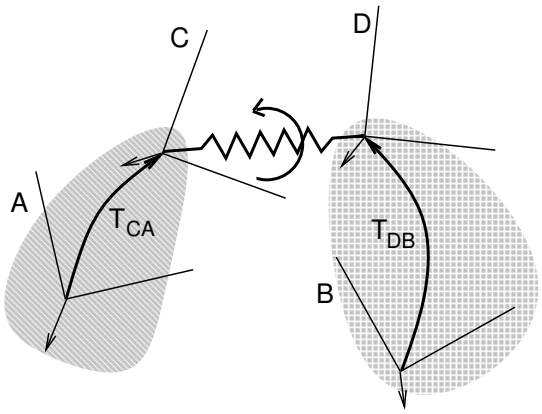


Figure 3.10: A frame spring connecting two rigid bodies A and B.

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 3.10).

Frame materials

The restoring forces (3.15) 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 behaviour may be defined in `artisynth.core.materials`.

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 3.3.2), 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.

Example: Two bodies connected by a frame spring

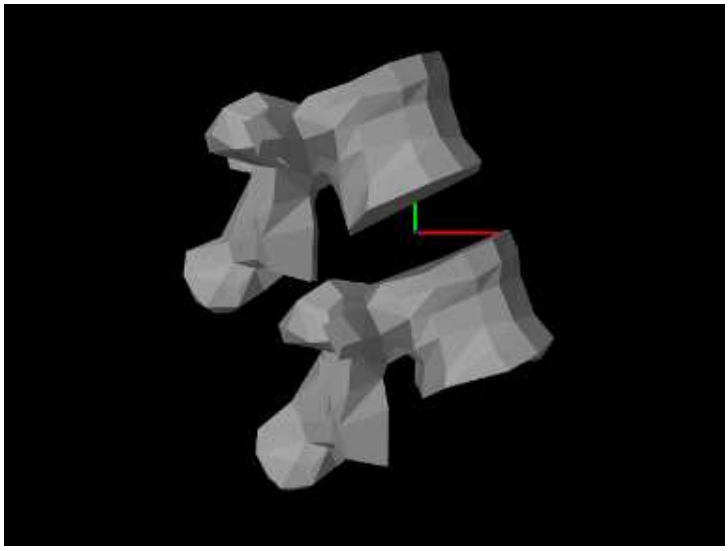


Figure 3.11: LumbarFrameSpring model loaded into ArtiSynth.

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.util.*;
10 import artisynth.core.workspace.RootModel;
11 import maspack.matrix.*;
12 import maspack.geometry.*;
```

```
13 import maspack.render.*;
14
15 /**
16  * Demo of two rigid bodies connected by a 6 DOF frame spring
17  */
18 public class LumbarFrameSpring extends RootModel {
19
20    double density = 1500;
21
22    // path from which meshes will be read
23    private String geometryDir = ArtisynthPath.getSrcRelativePath (
24       LumbarFrameSpring.class, "../mech/geometry/");
25
26    // create and add a rigid body from a mesh
27    public RigidBody addBone (MechModel mech, String name) throws IOException {
28       PolygonalMesh mesh = new PolygonalMesh (new File (geometryDir+name+".obj"));
29       RigidBody rb = RigidBody.createFromMesh (name, mesh, density, /*scale=*/1);
30       mech.addRigidBody (rb);
31       return rb;
32    }
33
34    public void build (String[] args) throws IOException {
35
36       // create mech model and set it's properties
37       MechModel mech = new MechModel ("mech");
38       mech.setGravity (0, 0, -1.0);
39       mech.setFrameDamping (0.10);
40       mech.setRotaryDamping (0.001);
41       addModel (mech);
42
43       // create two rigid bodies and second one to be fixed
44       RigidBody lumbar1 = addBone (mech, "lumbar1");
45       RigidBody lumbar2 = addBone (mech, "lumbar2");
46       lumbar1.setPose (new RigidTransform3d (-0.016, 0.039, 0));
47       lumbar2.setDynamic (false);
48
49       // flip entire mech model around
50       mech.transformGeometry (
51          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    }
66 }
```

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. ArtisynthPath.getSrcRelativePath() is used to find a proper path to this directory 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).

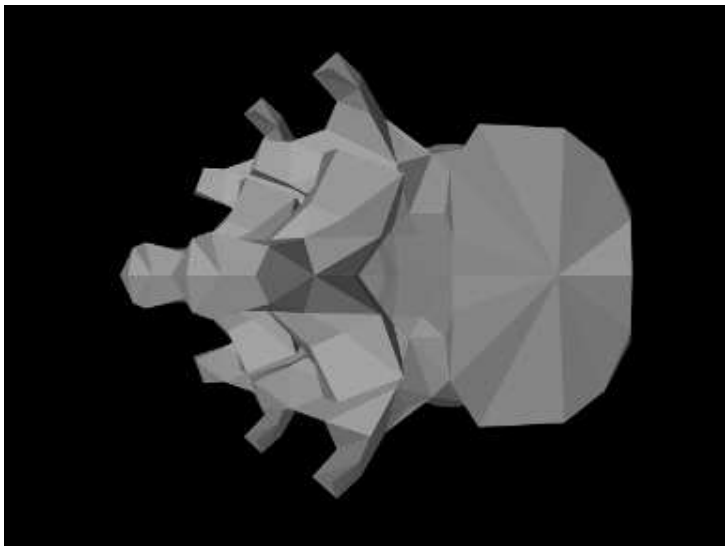


Figure 3.12: LumbarFrameSpring model as it would appear if not rotated about the x axis.

At this point in the construction, if the model were to be loaded, it would appear as in Figure 3.12. To change the viewpoint to that seen in Figure 3.11, 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 4.7.

The frame spring is created and added at lines 54-59, using the methods described in Section 3.4.3, 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 3.11. 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 manipulator (see the section "Pull Manipulator" in the ArtiSynth User Interface Guide).

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.

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 at 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);
```

Example: model with particle attachments

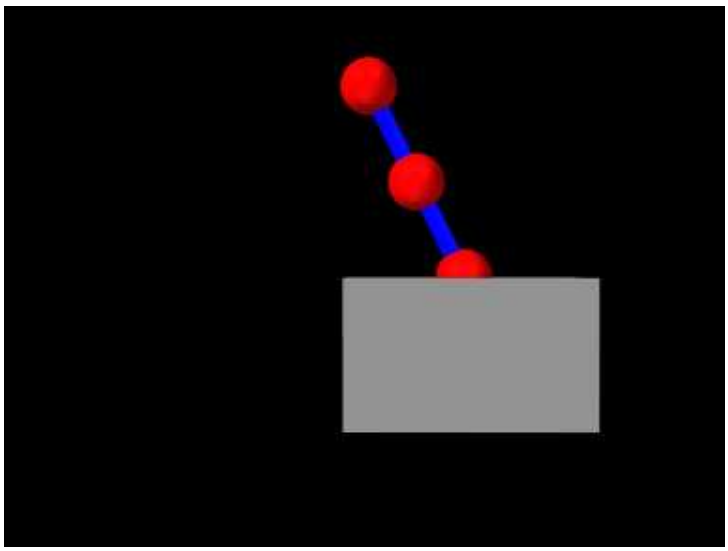


Figure 3.13: ParticleAttachment model loaded into ArtiSynth.

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);
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 3.13. Running the model (Section 1.5.3) will cause the box to fall and swing under gravity.

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.

Example: model with frame attachments

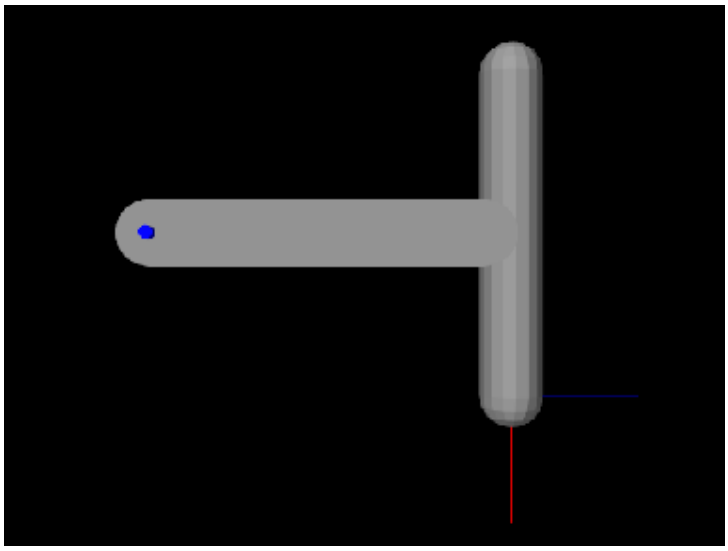


Figure 3.14: FrameBodyAttachment model loaded into ArtiSynth.

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 3.3.3, 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);
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 3.13. 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.

Simulation control properties

Both RootModel and MechModel contain properties that control the simulation behavior. 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();
```

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.

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).

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.

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.

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)

```
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.

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

```
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.

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.

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.

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.

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:

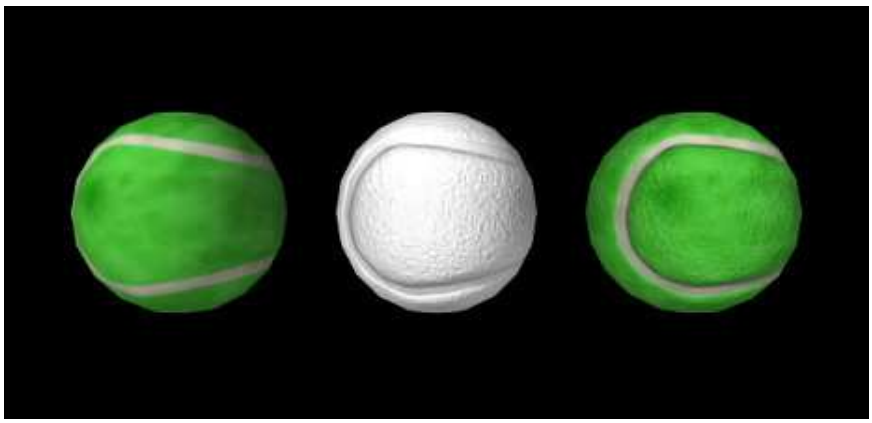


Figure 4.1: Color and bump mapping applied to spherical meshes. Left: color mapping only. Middle: bump mapping only. Right: combined color and bump mapping.

```
1 package artisynth.demos.tutorial;
2
3 import java.awt.Color;
4
5 import maspack.geometry.*;
```

```
import maspack.matrix.RigidTransform3d;
import maspack.render.*;
import maspack.render.Renderer.ColorMixing;
import maspack.render.Renderer.Shading;
import maspack.util.PathFinder;
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);
      RenderProps.setFaceColor (mech, Color.WHITE);
      // create and add the texture maps (provided free courtesy of
      // www.robinwood.com).
      String dataFolder = PathFinder.expand (
         "${srcdir SphericalTextureMapping}/data");

      ColorMapProps cprops = new ColorMapProps();
      cprops.setEnabled (true);
      // no specular coloring since ball should be matt
      cprops.setSpecularColoring (false);
      cprops.setFileName (dataFolder + "/TennisBallColorMap.jpg");

      BumpMapProps bprops = new BumpMapProps();
      bprops.setEnabled (true);
      bprops.setScaling ((float)radius/10);
      bprops.setFileName (dataFolder + "/TennisBallBumpMap.jpg");

      // apply color map to balls 0 and 2. Can do this by setting color map
      // properties in the MechModel, so that properties are controlled in one
      // place - but we must then also explicitly enable color mapping in balls
      // 0 and 2.
```

```
71        RenderProps.setColorMap (mech, cprops);
72        RenderProps.setColorMapEnabled (ball0, true);
73        RenderProps.setColorMapEnabled (ball2, true);
74
75        // apply bump map to balls 1 and 2.
76        RenderProps.setBumpMap (ball1, bprops);
77        RenderProps.setBumpMap (ball2, bprops);
78     }
79  }
```

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.

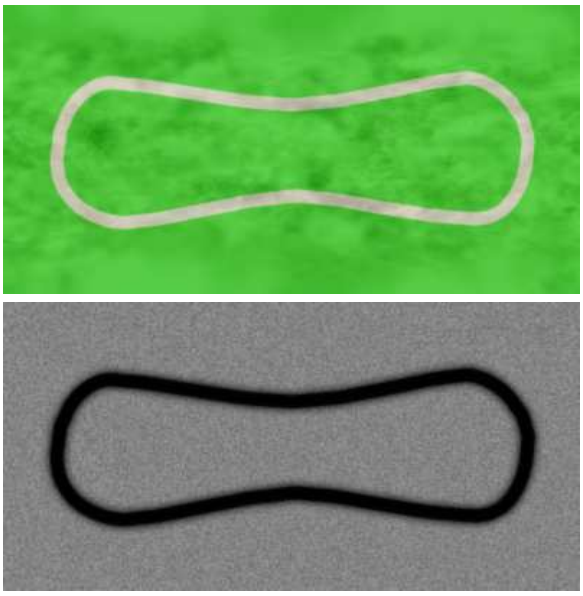


Figure 4.2: Color and bump map images used in the texture mapping example. These map to spherical coordinates on the mesh.

Color mapping is applied to balls 0 and 2, and bump mapping to balls 1 and 2. To illustrate setting mapping properties in an ancestor component, the color mapping is arranged by setting the colorMap render property of the `MechModel`, and then, as described above, enabling color mapping within the individual balls.

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.

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();
```

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 non-linear passive terms and active terms that depend on the muscle length l. Some of those available in ArtiSynth include ConstantAxialMuscle, BlemkerAxialMuscle, PaiAxialMuscle, and PeckAxialMuscle.

Example: Muscle attached to a rigid body

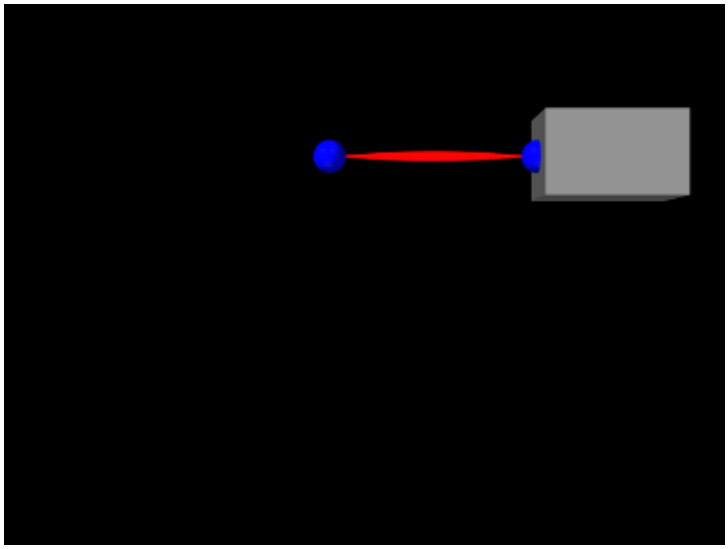


Figure 4.3: SimpleMuscle model loaded into ArtiSynth.

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.

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 interactioons 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.

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 two methods:

```
setDefaultCollisionBehavior (enabled, mu);
setDefaultCollisionBehavior (behavior);
```

In the first method, `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`. In the second method, `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.

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. For example, 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 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.

Example: Collision with a plane

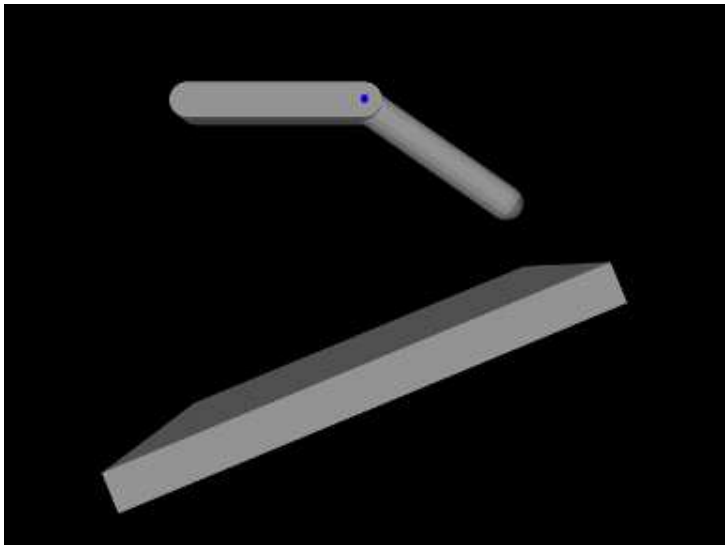


Figure 4.4: JointedCollide model loaded into ArtiSynth.

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.

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 4.6.3). 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).

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.

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`.

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 4.6.4).

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`.

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.

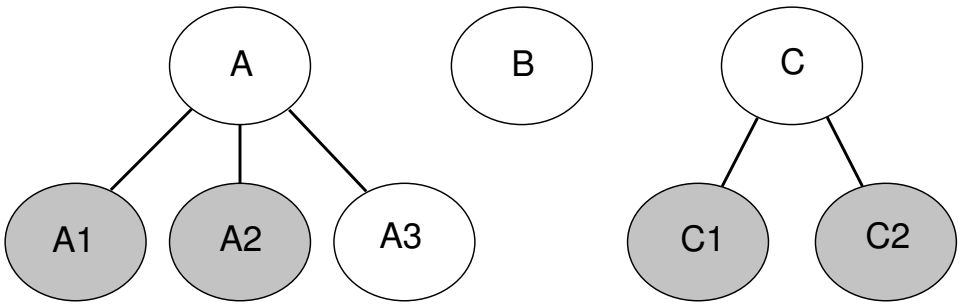


Figure 4.5: A collection of collidable components, where A possesses sub-collidables A1, A2, and A3, B is solitary, and C possesses sub-collidables C1 and C2. Internal collisions are enabled among those sub-collidables which are shaded gray.

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.
```

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.

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 impulses 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 contact impulses acting on all the deformable
// bodies associated with the first or second collidable, as specified by
// cidx=0 or cidx=1.
Map<Vertex3d,Vector3d> getContactImpulses(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 coule 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.getContactImpulses (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.getContactImpulses (0);
```

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`.

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);
```

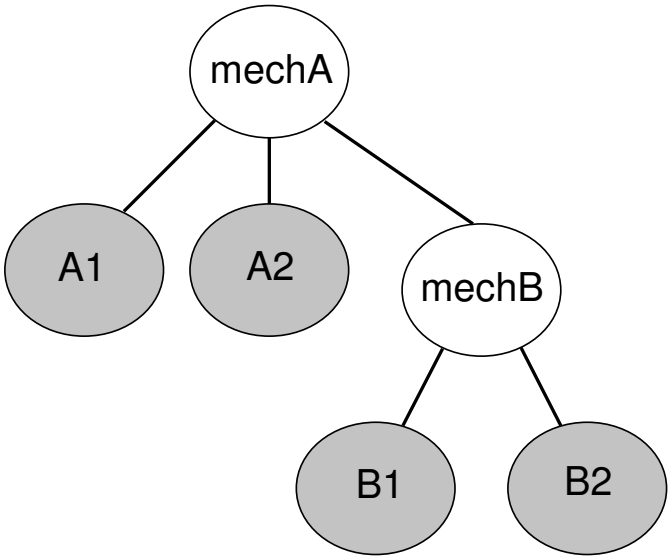


Figure 4.6: A `MechModel` containing collidables `B1` and `B2`, nested within another containing collidables `A1` and `A2`.

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();

SignedDistanceGrid getDistanceGrid();
```

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 getDistanceGrid() and is used by the collider type `SIGNED_DISTANCE` (Section 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.

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 supprted:

AJL_CONTOUR

A bounding-box hierachy 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 hierachy 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 hierachy. This latter step requires interating 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 getDistanceGrid(). 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 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.

Configuring and rendering signed distance grids

As mentioned above, bodies which maintain signed distance grids expose these using the hasDistanceGrid() and getDistanceGrid() methods. By default, these grids are created automatically on-demand within an axis-aligned bounding volume for the mesh. The cell resolution (number of cells along the x, y, z directions) is also choosen automatically. However, in some cases it may be necessary to control the cell resolution more precisely. If the body supports the interface HasDistanceGrid, this provides a set of methods for controlling the cell resolution:

```
void setDistanceGridRes (Vector3i res);
Vector3i getDistanceGridRes();

void setDistanceGridMaxRes (int max);
int getDistanceGridMaxRes();
```

setDistanceGridRes() explicity sets the grid resolution, which is the number of grid cells along the x, y, z axes. If any of these values are specified as 0, then all values are set to zero and the resolution is determined instead by the maximum cell resolution returned by getDistanceGridMaxRes(), which specifies the number of cells along the *maximum* x, y, z direction, with the cell counts along the other directions adjusted to maintain a uniform cell size.

`HasDistanceGrid` also provides methods to support the rendering of the signed distance grid:

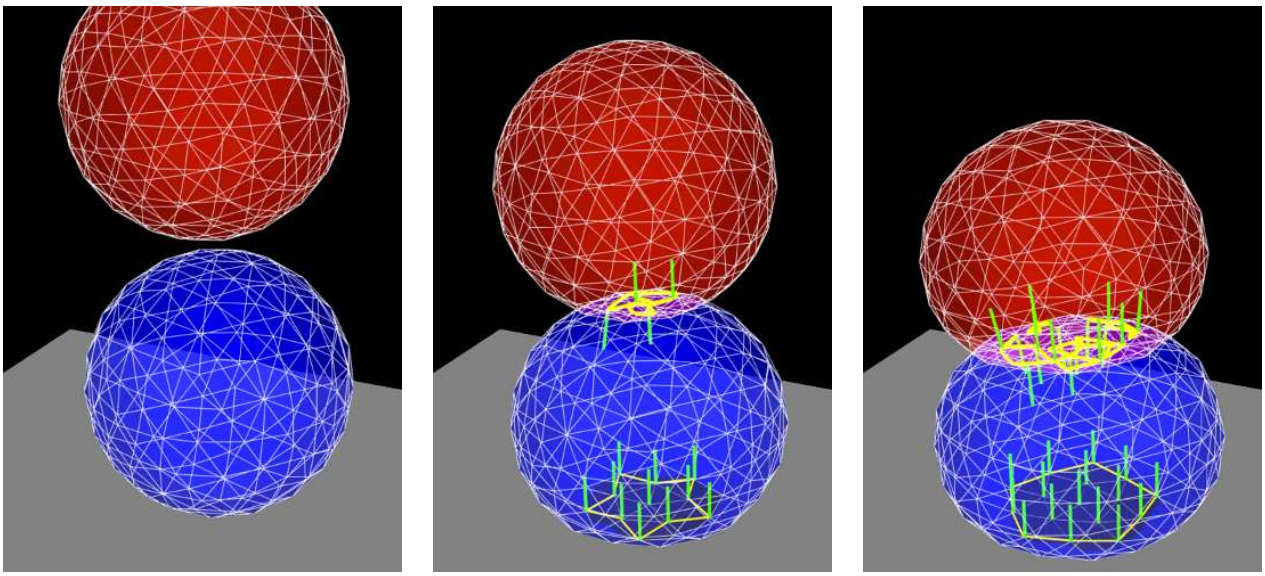


Figure 4.7: Time sequence of contact handling between two deformable models falling under gravity, showing the intersection contours (yellow) and the contact normals (green lines).

```
void setRenderDistanceGrid (boolean enable);
boolean getRenderDistanceGrid();

void setDistanceGridRenderRanges (String ranges);
String getDistanceGridRenderRanges();
```

setRenderDistanceGrid() enables or disables grid rendering. By default, grids are rendered by drawing each cell vertex along with the associated normal (which is the numeric derivative of the distance field), using the point and line render properties for the component. In some cases, it may be desirable to restrict which vertices are drawn. This can be done using setDistanceGridRenderRanges(), which takes a string argument containing vertex render ranges for the x, y, and z axes. A vertex render range may be either `*` (all vertices), `n:m` (vertices in the index range `n` to `m`, inclusive), or `n` (vertices only at index `n`). For example:

```
"* * *"       - all vertices
"* 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 indices 0-3 along x, and 4-5 along y
```

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 [4] to compute dry friction, instead of the polyhedralized friction cones common in the multibody dynamics literature [1, 7]. This allows for a less expensive and more robust computation at the expense of some accuracy.

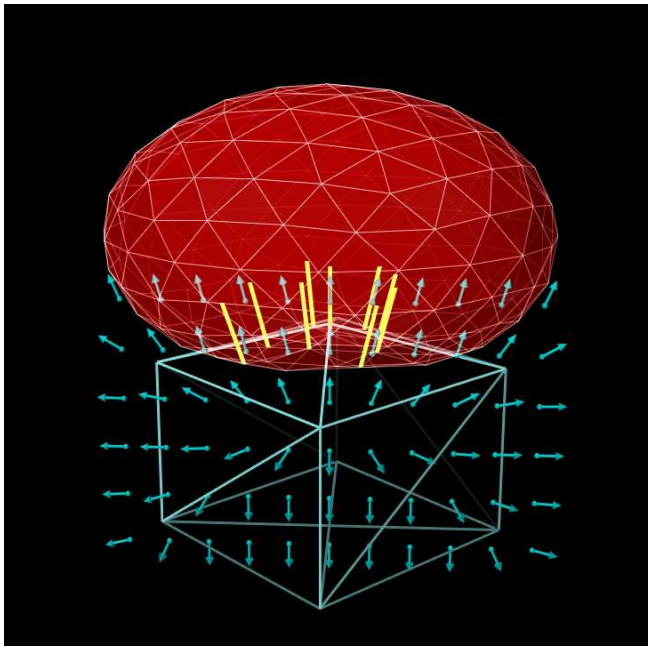


Figure 4.8: Contacts generated by a signed distance field. A deformable ellipsoid (red, top) is colliding with a solid prism (cyan, bottom). A signed distance field for the prism (the grid points and normals of which are shown partially by the dark cyan arrows) is used to locate penetrating vertices of the ellipsoid and hence generate contact constraints (yellow lines).

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 meshes, using the edgeWidth and edgeColor render properties.

drawIntersectionPoints

Boolean value requesting drawing of the interpenetrating vertices on each 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, lineWidth, lineRadius, 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.

drawPenetrationDepth

Integer value requesting drawing of the interpenetration depth of one mesh with respect to the other. The integer value should be either `-1`, `0` or `1`, where `-1` disables this feature and `0` or `1` select depth drawing for either the first or second mesh associated with the collision behavior.

Penetration depth is drawn using a "heat map" type of rendering, where the depth value range is controlled using the penetrationDepthRange property (see next item) and depth values are mapped onto colors using the colorMap property of the collision manager. The depth map itself is drawn as a patch formed from the mesh's penetrating faces, using vertex coloring where the color at each vertex is determined by the penetration depth. An example is given in Section 4.6.6.

penetrationDepthRange

Composite property of the type ScalarRange that controls the range used for depth rendering. Sub properties include interval, which gives the depth range to be used for the depth map, and updating, which specifies how this range should be updated: `FIXED` (no updating), `AUTO_EXPAND` (range is expanded as depth increases), and `AUTO_FIT` (range is reset every step).

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();
```

Example: Rendering normals and contours

A simple model showing contact rendering is defined in

`artisynth.demos.tutorial.BallPlateCollide`

This shows a rigid sphere colliding with a plane, while rendering the resulting contact normals in red and the intersection contours in blue.

The complete source code is shown below:

```
1  package artisynth.demos.tutorial;
2
3  import java.awt.Color;
4  import maspack.matrix.*;
5  import maspack.render.*;
6  import artisynth.core.workspace.*;
7  import artisynth.core.mechmodels.*;
8
9  public class BallPlateCollide extends RootModel {
10
11    public void build (String[] args) {
12
13       // create MechModel and add to RootModel
14       MechModel mech = new MechModel ("mech");
```

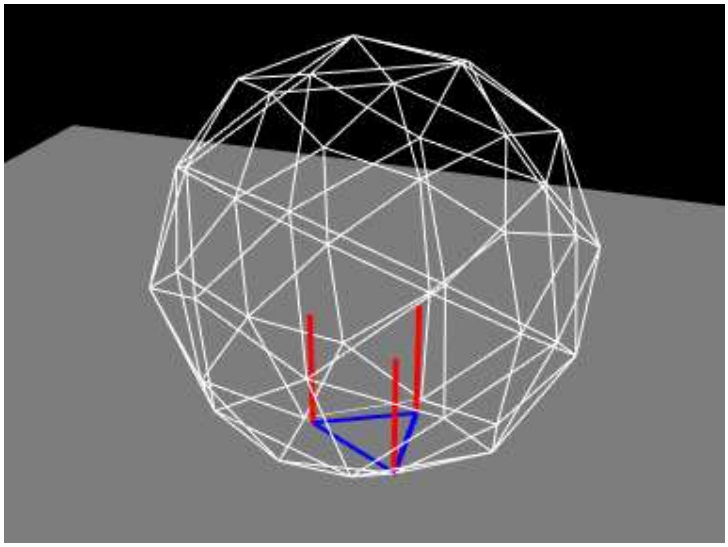


Figure 4.9: BallPlateCollide showing contact normals (red) and collision contour (blue) of the ball colliding with the plate.

```
15        addModel (mech);
16
17        // create and add the ball and plate
18        RigidBody ball = RigidBody.createIcosahedralSphere ("ball", 0.8, 0.1, 1);
19        ball.setPose (new RigidTransform3d (0, 0, 2, 0.4, 0.1, 0.1));
20        mech.addRigidBody (ball);
21        RigidBody plate = RigidBody.createBox ("plate", 5, 5, 0.4, 1);
22        plate.setDynamic (false);
23        mech.addRigidBody (plate);
24
25        // turn on collisions
26        mech.setDefaultCollisionBehavior (true, 0.20);
27
28        // make ball transparent so that contacts can be seen more clearly
29        RenderProps.setFaceStyle (ball, Renderer.FaceStyle.NONE);
30        RenderProps.setShading (ball, Renderer.Shading.NONE);
31        RenderProps.setDrawEdges (ball, true);
32        RenderProps.setEdgeColor (ball, Color.WHITE);
33
34        // enable rendering of contacts normals and contours
35        CollisionManager cm = mech.getCollisionManager();
36        RenderProps.setVisible (cm, true);
37        RenderProps.setLineWidth (cm, 3);
38        RenderProps.setLineColor (cm, Color.RED);
39        RenderProps.setEdgeWidth (cm, 3);
40        RenderProps.setEdgeColor (cm, Color.BLUE);
41        cm.setDrawContactNormals (true);
42        cm.setDrawIntersectionContours (true);
43     }
44  }
```

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.

Example: Rendering penetration depth

As described above, it is possible to use the drawPenetrationDepth property to render the extent to which one mesh interpenetrates another. A simple example of this is defined in

```
artisynth.demos.tutorial.PenetrationRender
```

which displays the penetration depth of one hemispherical mesh with respect to another.

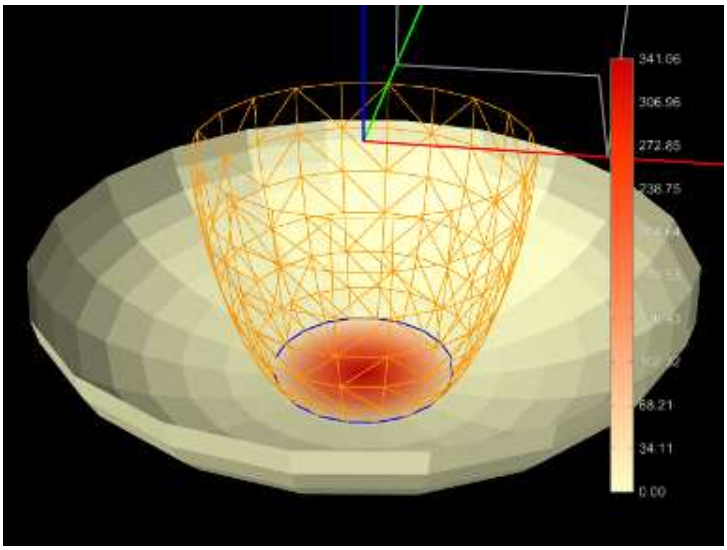


Figure 4.10: PenetrationRender showing the penetration depth of the bottom mesh with respect to the top, with red indicating greater depth. A translation dragger fixture at the top is being used to move the top mesh around, while the penetration range and associated colors are displayed on the color 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.util.ScalarRange;
14 import artisynth.core.workspace.RootModel;
15 import artisynth.core.renderables.ColorBar;
16 import maspack.render.color.JetColorMap;
17
18 public class PenetrationRender extends RootModel {
19
20    // Convenience method for creating colors from [0-255] RGB values
21    private static Color createColor (int r, int g, int b) {
22       return new Color (r/255.0f, g/255.0f, b/255.0f);
23    }
24
25    private static Color CREAM = createColor (255, 255, 200);
26    private static Color GOLD = createColor (255, 150, 0);
27
28    // Creates and returns a rigid body built from a hemispherical mesh.  The
29    // body is centered at the origin, has a radius of 'rad', and the z axis is
30    // scaled by 'zscale'.
31    RigidBody createHemiBody (
32       MechModel mech, String name, double rad, double zscale, boolean flipMesh) {
33
34       PolygonalMesh mesh = MeshFactory.createHemisphere (
```

```
35          rad, /*slices=*/20, /*levels=*/10);
36       mesh.scale (1, 1, zscale); // scale mesh in the z direction
37       if (flipMesh) {
38          // flip upside down is requested
39          mesh.transform (new RigidTransform3d (0, 0, 0, 0, 0, Math.PI));
40       }
41       RigidBody body = RigidBody.createFromMesh (
42          name, mesh, /*density=*/1000, /*scale=*/1.0);
43       mech.addRigidBody (body);
44       body.setDynamic (false);  // body is only parametrically controlled
45       return body;
46    }
47
48    // Creates and returns a ColorBar renderable object
49    public ColorBar createColorBar() {
50       ColorBar cbar = new ColorBar();
51       cbar.setName("colorBar");
52       cbar.setNumberFormat("%.2f");       // 2 decimal places
53       cbar.populateLabels(0.0, 1.0, 10); // Start with range [0,1], 10 ticks
54       cbar.setLocation(-100, 0.1, 20, 0.8);
55       cbar.setTextColor (Color.WHITE);
56       addRenderable(cbar);                // add to root model's renderables
57       return cbar;
58    }
59
60    public void build (String[] args) {
61       MechModel mech = new MechModel ("mech");
62       addModel (mech);
63
64       // create first body and set its rendering properties
65       RigidBody body0 = createHemiBody (mech, "body0", 2, -0.5, false);
66       RenderProps.setFaceStyle (body0, FaceStyle.FRONT_AND_BACK);
67       RenderProps.setFaceColor (body0, CREAM);
68
69       // create second body and set its pose and rendering properties
70       RigidBody body1 = createHemiBody (mech, "body1", 1, 2.0, true);
71       body1.setPose (new RigidTransform3d (0, 0, 0.75));
72       RenderProps.setFaceStyle (body1, FaceStyle.NONE); // set up
73       RenderProps.setShading (body1, Shading.NONE);     // wireframe
74       RenderProps.setDrawEdges (body1, true);           // rendering
75       RenderProps.setEdgeColor (body1, GOLD);
76
77       // create and set a collision behavior between body0 and body1, and make
78       // collisions INACTIVE since we only care about graphical display
79       CollisionBehavior behav = new CollisionBehavior (true, 0);
80       behav.setMethod (CollisionBehavior.Method.INACTIVE);
81       behav.setDrawPenetrationDepth (0); // show penetration of mesh 0
82       behav.getPenetrationDepthRange().setUpdating (
83          ScalarRange.Updating.AUTO_FIT);
84       mech.setCollisionBehavior (body0, body1, behav);
85
86       CollisionManager cm = mech.getCollisionManager();
87       // penetration rendering only works with contour-based collisions
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
107       // create a separate color bar to show depth values associated with the
108       // color map
109       ColorBar cbar = createColorBar();
110       cbar.setColorMap (map);
111    }
112
113    public void prerender(RenderList list) {
114       // In prerender, we update the color bar labels based on the updated
115       // penetration range stored in the collision behavior.
116       //
117       // Object references are obtained by name using 'findComponent'. This is
118       // more robust than using class member variables, since the latter will
119       // be lost if we save and restore this model from a file.
120       ColorBar cbar = (ColorBar)(renderables().get("colorBar"));
121       MechModel mech = (MechModel)findComponent ("models/mech");
122       RigidBody body0 = (RigidBody)mech.findComponent ("rigidBodies/body0");
123       RigidBody body1 = (RigidBody)mech.findComponent ("rigidBodies/body1");
124
125       CollisionBehavior behav = mech.getCollisionBehavior(body0, body1);
126       ScalarRange range = behav.getPenetrationDepthRange();
127       cbar.updateLabels(0, 1000*range.getUpperBound());
128       super.prerender(list); // call the regular prerender method
129    }
130 }
```

To improve visibility, the example uses two rigid bodies, each created from an open hemispherical mesh using the method `createHemiBody()` (lines 31-46). 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 "Geometric Manipulation" in the ArtiSynth User Interface Guide). Rendering properties for each body are set at lines 66-67 and 72-75, with the top body being rendered as a wireframe to improve visibility.

Lines 79-84 create and set a collision behavior between the two bodies with penetration depth rendering enabled. 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. The updating of the penetration range is also set to `AUTO_FIT` so that it will be recomputed at every time step. At line 82, the collider type used by the collision manager is set to ColliderType.AJL_CONTOUR, since this is needed for depth rendering. Other rendering properties are set for the collision manager at lines 90-105, including a custom color map that varies between `CREAM` (the color of the mesh) for no penetration and dark red for maximum penetration.

At line 109, a color bar is created and added to the scene, using the method `createColorBar()` (lines 49-58), 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 113-130), where we obtain the collision behavior between the two bodies, and use its depth range to update the color bar labels. 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.

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

```
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 inertiaDensity property is set to Density. Figure 4.11 shows a simplified illustration of both rigid and affine transformations being applied to a model.

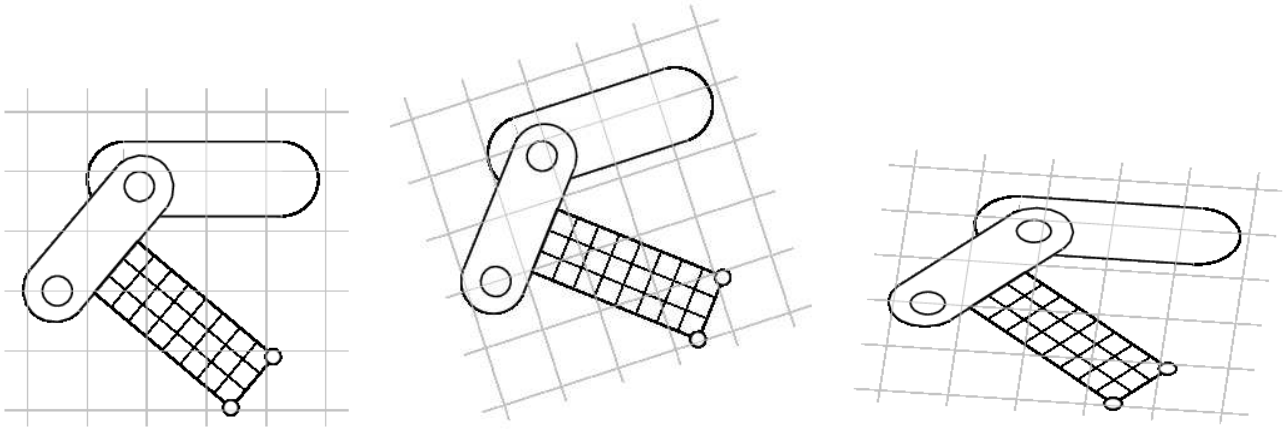


Figure 4.11: Simple illustration of a model (left) undergoing a rigid transformation (middle) and an affine transformation (right).

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 ...

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.transformGeomerty (T);
```

Nonlinear transformations

The TransformableGeometry interface also supports general, non-linear 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

$$\mathbf{F} \equiv \frac{\partial f}{\partial \mathbf{x}} \tag{4.2}$$

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 4.7.2, and by Figure 4.13. 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.

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.

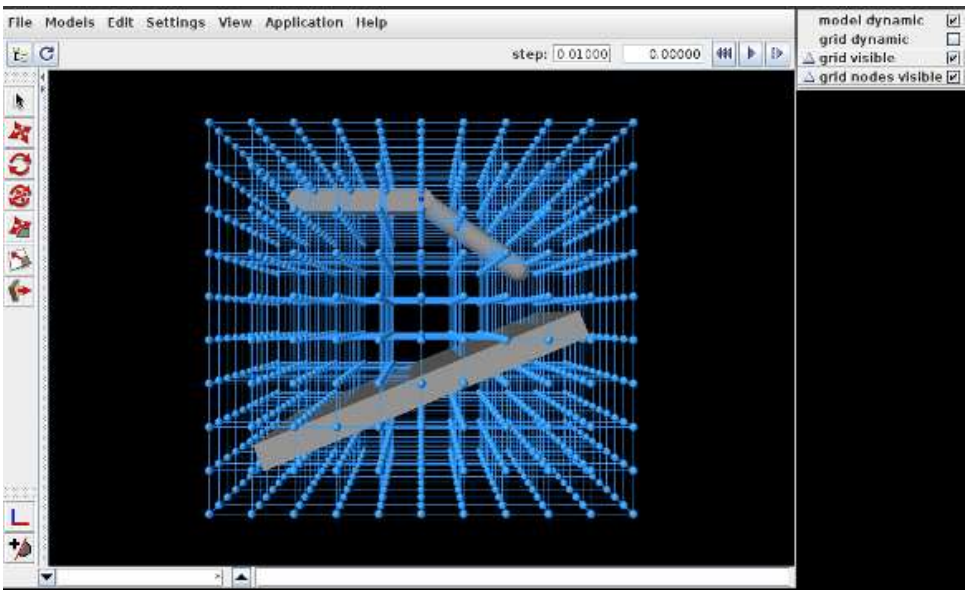


Figure 4.12: The DeformedJointedCollide example initially loaded into ArtiSynth.

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 4.12, 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 "Manipulation 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.

The result of a possible deformation is shown in Figure 4.13.

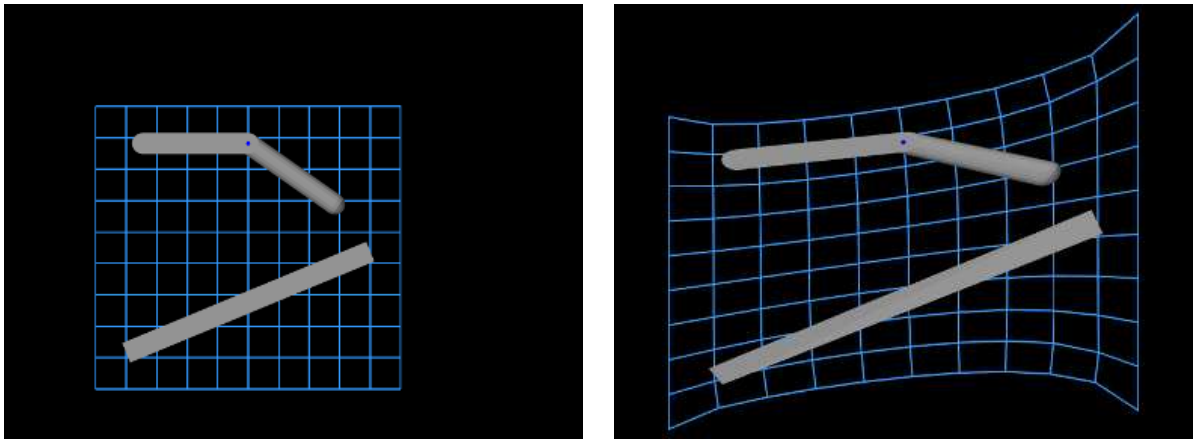


Figure 4.13: Deformation achieved in DeformedJointedCollide, showing both the model and grid (using an orthographic view) before and after the deformation.

> Note: `FemModelDeformer` is not intended to provide a general purpose solution to non-linear geometric transformations. Rather, it is mainly intended to illustrate the capabilities of GeometryTransformer and the TransformableGeometry interface.

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.

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.

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.

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.

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`

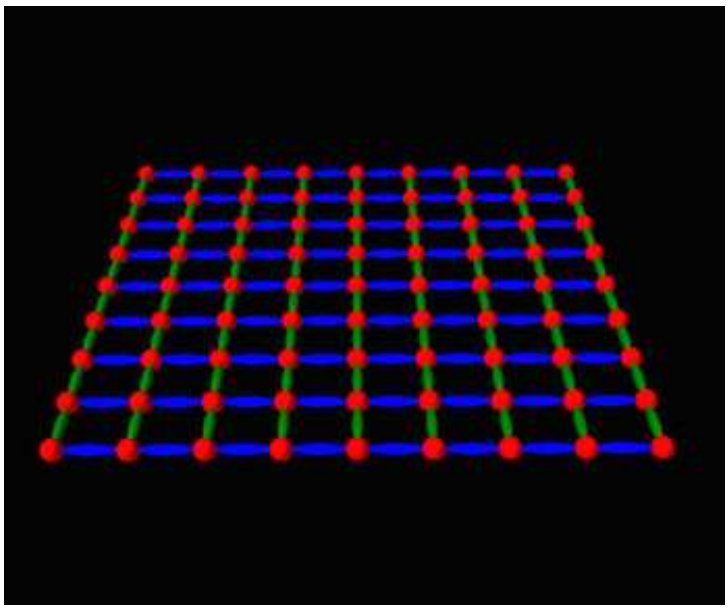


Figure 4.14: NetDemo model loaded into ArtiSynth.

The `build()` method and some of the supporting definitions for this example are shown below.

```
   protected double stiffness = 1000.0;    // spring stiffness
   protected double damping = 10.0;        // spring damping
   protected double maxForce = 5000.0;     // max force with excitation = 1
   protected double mass = 1.0;            // mass of each ball
   protected double widthx = 20.0;         // width of the net along x
   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));
```

```
23        return spr;
24     }
25
26     public void build (String[] args) {
27
28        // create MechModel and add to RootModel
29        mech = new MechModel ("mech");
30        mech.setGravity (0, 0, -980.0);
31        mech.setPointDamping (1.0);
32        addModel (mech);
33
34        int nump = (numx+1)*(numy+1); // nump = total number of balls
35
36        // create custom containers:
37        balls = new PointList<Particle> (Particle.class, "balls");
38        springs = new ComponentList<ModelComponent>(ModelComponent.class,"springs");
39        greenSprings = new RenderableComponentList<AxialSpring> (
40           AxialSpring.class, "greenSprings");
41        blueSprings = new RenderableComponentList<AxialSpring> (
42           AxialSpring.class, "blueSprings");
43
44        // 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 4.14. 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 4.15. While the standard `MechModel` containers are still present, they are not displayed by default because they are empty.

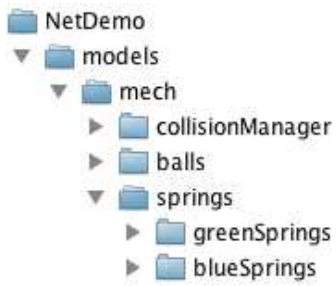


Figure 4.15: NetDemo components displayed in the ArtiSynth navigation panel.

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.

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.

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.

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 1.4.1, 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.

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:

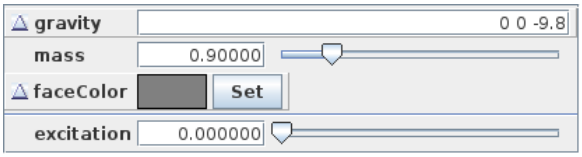


Figure 5.1: Control panel created by the model `SimpleMuscleWithPanel`.

```
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, rigod 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.

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.

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.

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.

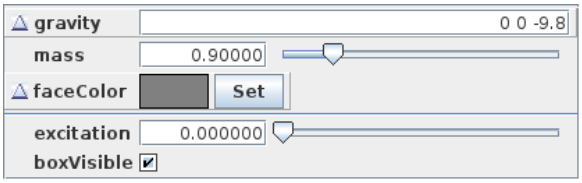


Figure 5.2: Control panel created by the model `SimpleMuscleWithProperties`, showing the newly defined property `boxVisible`.

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.

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
```

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.

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).

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.

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()`.

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
6 import artisynth.core.modelbase.*;
7 import artisynth.core.mechmodels.*;
```

```
8  import artisynth.core.probes.*;
9  import artisynth.core.util.*;
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             ArtisynthPath.getSrcRelativePath (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             ArtisynthPath.getSrcRelativePath (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 ArtisynthPath.getSrcRelativePath() is used to locate the file relative to the source directory for the application model. 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.

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:

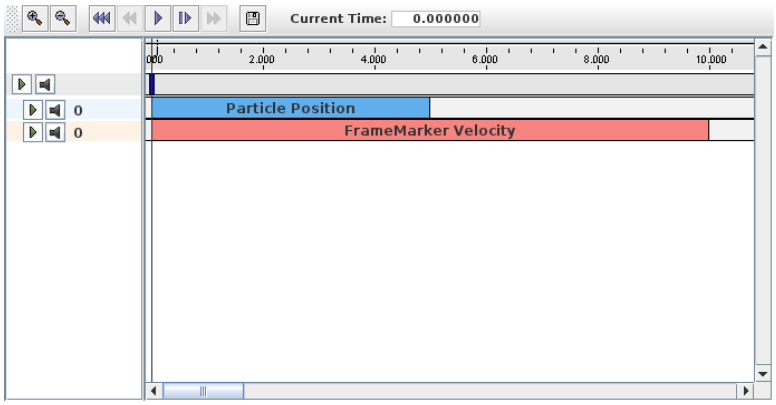


Figure 5.3: Timeline view of the probes created by SimpleMuscleWithProbes.

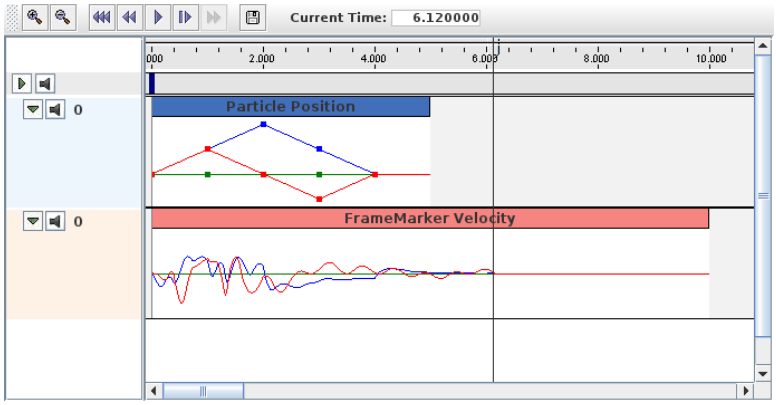


Figure 5.4: Expanded view of the probes after `SimpleMuscleWithProbes` has run for about 6 seconds, showing the data accumulated in the output probe "`FrameMarker Velocity`".

```
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
```

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);
```

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.

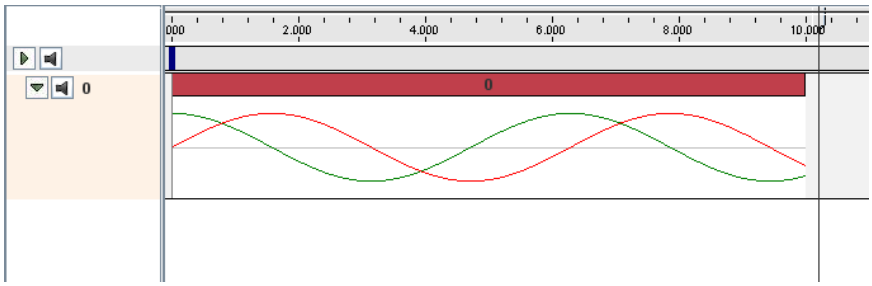


Figure 5.5: Output from a `NumericMonitorProbe` which generates sine and cosine waves.

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.

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.

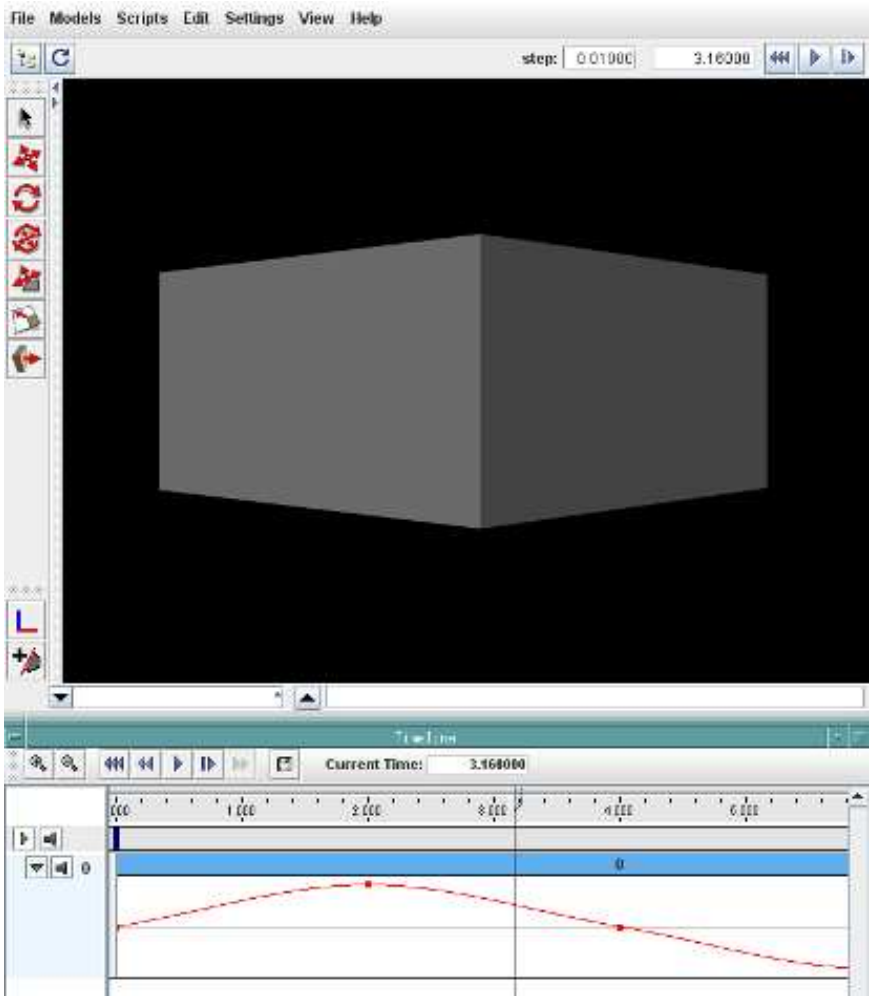


Figure 5.6: Screen shot of the `SpinControlDemo`, showing the numeric data in the timeline.

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).

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`.

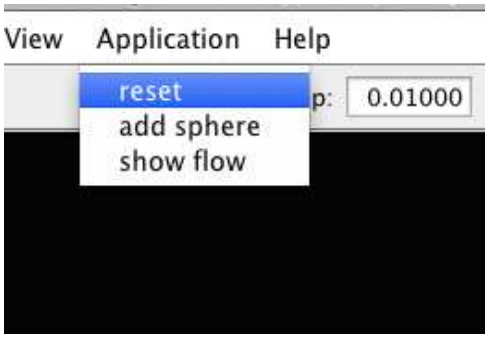


Figure 5.7: Application-defined menu items appearing under the ArtiSynth menu bar.

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 section 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 6.8. An example FEM model of the masseter, coupled to a rigid jaw and maxilla, is shown in Figure 6.1.

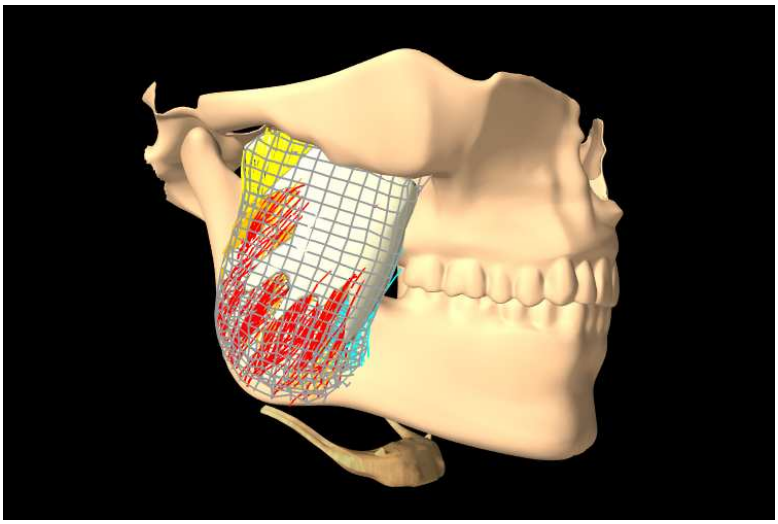


Figure 6.1: Finite element model of the masseter, coupled to the jaw and maxilla.

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 section 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].

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, a mass can later be specified for each node individually.

The FEM's `material` is a delegate object used to compute stress and stiffness within individual elements. It handles the *constitutive* component of the model. Materials will be discussed in more detail in Section 6.1.3.

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".

Component Structure

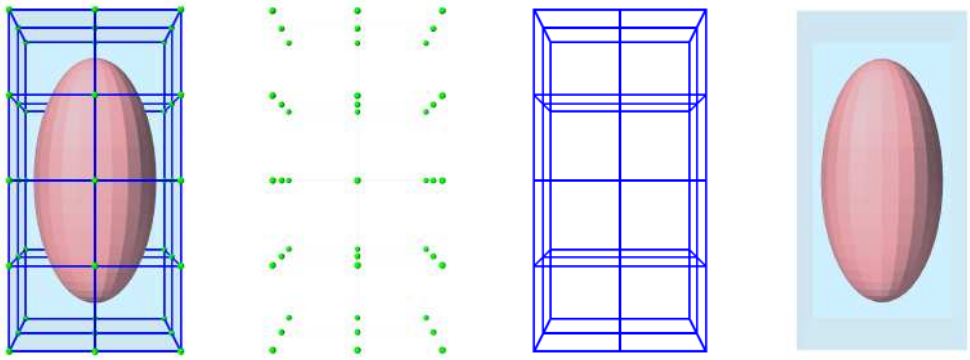
Each FemModel3d contains three lists of sub-components:

`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 spatial building blocks of the model. These define the 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.

An example showing each of these components is shown in Figure 6.2.



(a) FEM model (b) Nodes (c) Elements (d) Geometry

Figure 6.2: Sub-components of FemModel3d.

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.

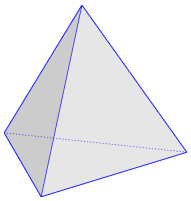
The FEM node's `dynamic` property specifies whether or not the node is considered when computing the dynamics of the system. If not, it is treated as being parametrically controlled. This has implications when setting boundary conditions (Section 6.1.4).

Elements

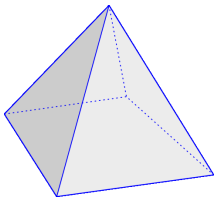
Elements are the spatial building blocks of the domain. Within each element, the displacement (or strain) field is interpolated from displacements at nodes:

$$\mathbf{u}(\mathbf{x}) = \sum_{i=1}^{N} \phi_i(\mathbf{x})\mathbf{u}_i, \tag{6.2}$$

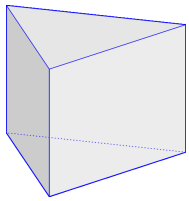
where $\mathbf{u}_i$ is the displacement of the ith node that is adjacent to the element, and $\phi_i(\cdot)$ is referred to as the *shape function* (or *basis function*) associated with that node. Elements are classified by their shape, number of nodes, and shape function order (Table 6.1). ArtiSynth supports the following element types:



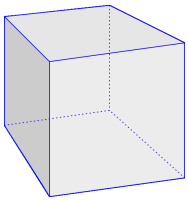
TetElement, QuadtetElement



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(); // return the list of elements
```

All objects of type FemModel3d 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.

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();   // return the list of meshes in a FEM
```

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:

```
public void computeStress (...)  // computes the symmetric stress tensor
public void computeTangent (...) // computes the local tangent stiffness matrix
```

Inputs include a deformation gradient, pressure, and a coordinate frame that specifies potential directions of anisotropy. 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 [6]. To enable or disable this corotational formulation, use LinearMaterial.setCorotated(boolean).

All material models, including linear and non-linear, are available in the package `artisynth.core.materials`. A list of common materials is provided in Table 6.2. Those that are subclasses of IncompressibleMaterial allow for incompressibility.

Table 6.2: 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

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 a FEM can be constrained by enabling collisions with other components. This will be covered in Section 6.9.

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.

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);
```

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 6.3. 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 6.3: 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 = ArtisynthPath.getSrcRelativePath(this,
                         "data/tongue.node");
   String elemFileName = ArtisynthPath.getSrcRelativePath(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 ArtisynthPath.getSrcRelativePath() is used to find a path within the ArtiSynth source tree that is relative to the current model's source file. Note the try-catch block. Most of these readers throw an `IOException` if the read fails.

Generating from surfaces

There are two ways a 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 a FEM from a surface, there is a convenience routine within FemFactory that handles both mesh generation and constructing a `FemModel3d`:

```
// Create a 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 a 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.

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 FemElement3d.getNodeCoords(). This returns the concatenated coordinate list for an "ideal" element of the given type.

Example: a simple beam model

A complete application model that implements a simple FEM beam is given below.

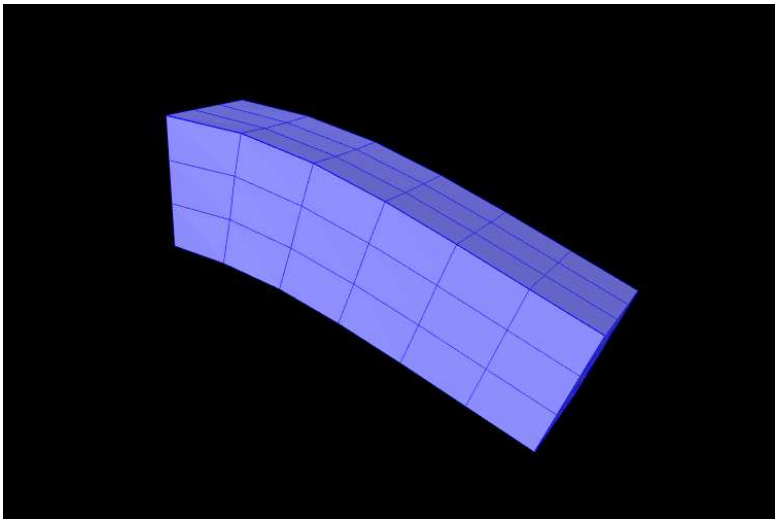


Figure 6.3: FemBeam model loaded into ArtiSynth.

```
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);
      fem.setParticleDamping (0.1);
      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 6.10.

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.

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.

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.

Example: a beam with an embedded sphere

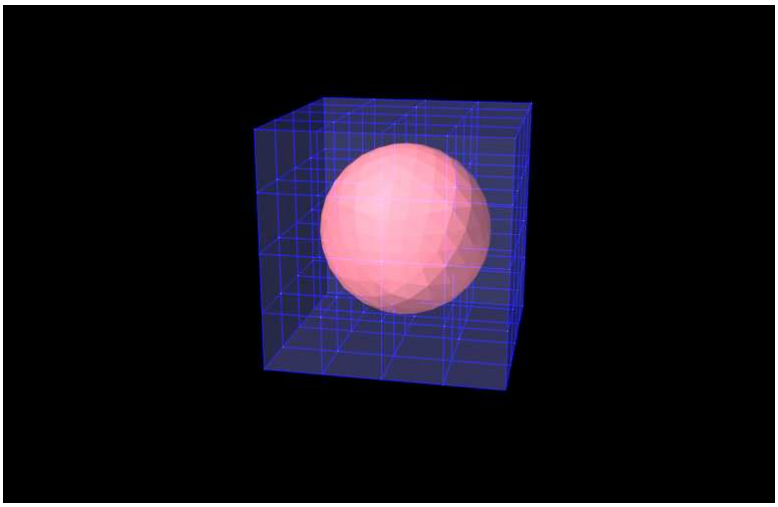


Figure 6.4: FemEmbeddedSphere model loaded into ArtiSynth.

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 < -0.49) {
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 6.4.

Node attachments

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 3.5. Common point-based attachment classes are listed in Table 6.4.

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
```

Table 6.4: 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

There are also convenience routines inside `MechModel` that will create the appropriate attachments automatically (see Section 3.5.1).

Connecting nodes to rigid bodies or particles

Since FemNode3d is a subclass of Particle, the same methods described in Section 3.5.1 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`.

Example: connecting a beam to a block

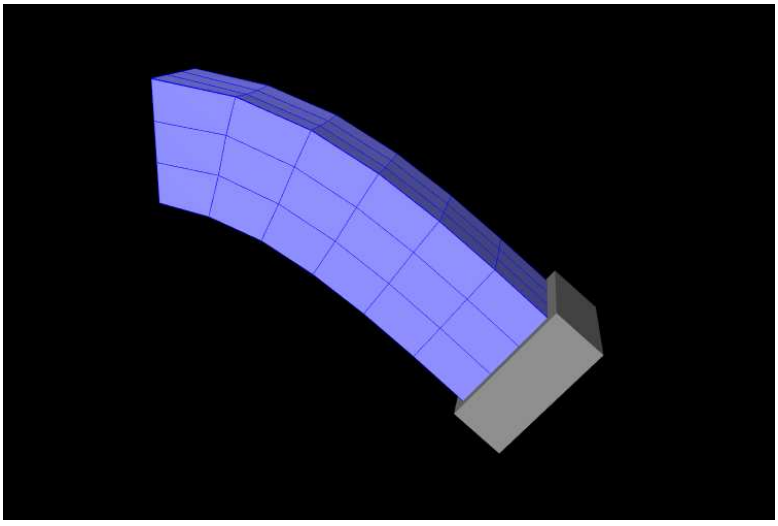


Figure 6.5: FemBeamWithBlock model loaded into artisynth.

The following model demonstrates attaching a 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);
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
```

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 a 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 $\mathbf{p}_k$ and velocities $\mathbf{u}_k$ of selected fem nodes:

$$\mathbf{p} = \sum w_k \mathbf{p}_k \tag{6.4}$$

Any force $\mathbf{f}$ acting on the attached point is then propagated back to the nodes, according to the relation

$$\mathbf{f}_k = w_k \mathbf{f} \tag{6.5}$$

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 $\mathbf{p}_k$ are simply the element nodes and the w_i are corresponding shape function values; this is known as an *element-based* attachment. On the other hand, as desribed 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 6.4.5).

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.

Example: connecting two FEMs together

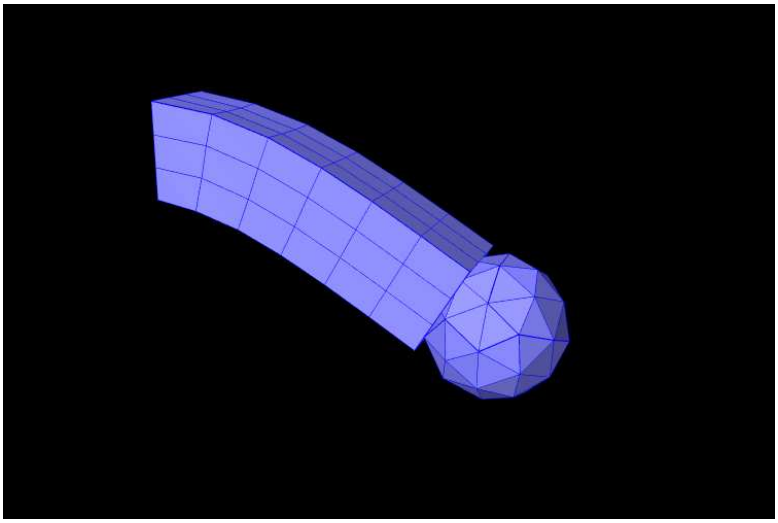


Figure 6.6: FemBeamWithFemSphere model loaded into ArtiSynth.

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 artisynth.core.util.ArtisynthPath;
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          ArtisynthPath.getSrcRelativePath(FemModel3d.class, "meshes/sphere2.1.node"),
23          ArtisynthPath.getSrcRelativePath(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).

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 w_k is initially computed as

$$w_k = \frac{d_{\max}}{d_k + d_{max}} \tag{6.6}$$

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`.

Example: element vs. nodal-based attachments

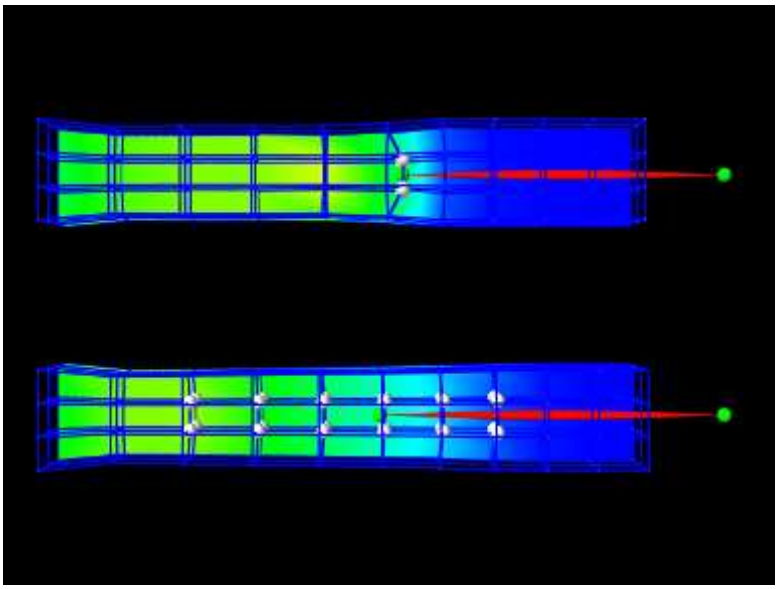


Figure 6.7: PointFemAttachment loaded into ArtiSynth and run until stable. The top and bottom models are connected to their springs using element and nodal-based attachments, respectively. The nodes associated with each attachment are rendered as white spheres.

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 6.7 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.

```
   // Filter to select only elements for which the nodes are entirely on the
   // positive side of the x-z plane.
   private class MyFilter extends ElementFilter {
      public boolean elementIsValid (FemElement e) {
         for (FemNode n : e.getNodes()) {
            if (n.getPosition().y < 0) {
               return false;
            }
         }
         return true;
      }
   }

   // Collect and return all the nodes of a 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.
      Particle p1 = addParticle (mech,  0.9, 0.0,  0.25);
      Particle p2 = addParticle (mech,  0.9, 0.0, -0.25);
      Particle m1 = addParticle (mech,  0.0, 0.0,  0.25);
      Particle m2 = addParticle (mech,  0.0, 0.0, -0.25);

      // attach spring end-point to fem1 using an element-based marker
      mech.attachPoint (m1, fem1);

      // attach spring end-point to fem2 using a larger number of nodes, formed
      // from the node set for elements 22, 31, 40, 49, and 58. This is done by
      // explicitly creating the attachment and then setting it to use the
      // specified nodes
      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 specufied 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 6.7. 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.

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 6.4.5:

```
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 6.4.5, 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
Vectord 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 nodel weights, as described in Equation (6.5).

Example: attaching a 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;
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 {
```

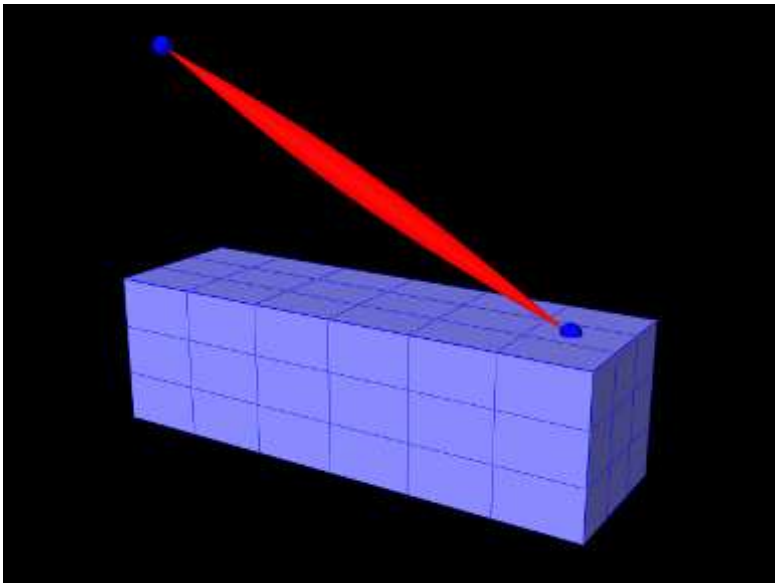


Figure 6.8: FemBeamWithMuscle model loaded into ArtiSynth.

```
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 6.8.

Frame attachments

It is also possible to attach frame components, including rigid bodies, directly to FEM models, using the attachment component FrameFem3dAttachment. Analagously 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 nodel-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 6.4.5): 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.

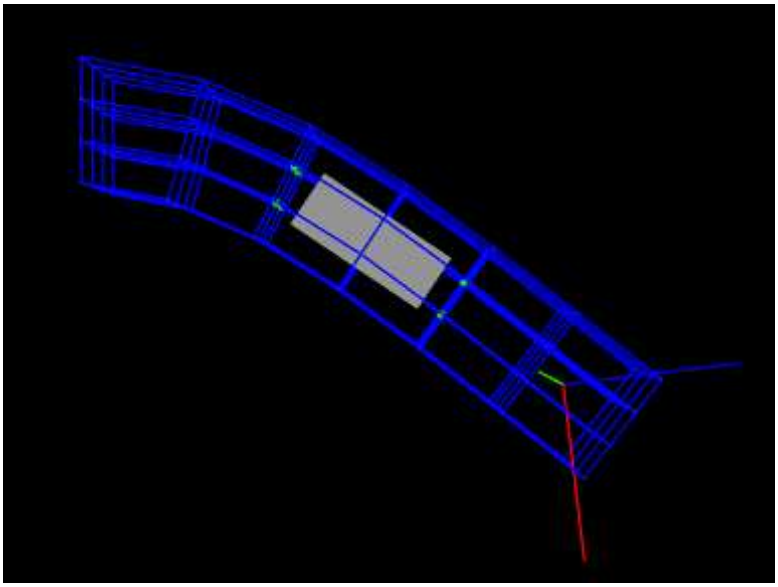


Figure 6.9: FrameFemAttachment loaded into ArtiSynth and run until stable.

Example: attaching frames to a FEM beam

A model illustrating how to connect frames to a FEM model is defined in

```
artisynth.demos.tutorial.FrameFemAttachment
```

It creates a 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 6.9. Forces can interactively be applied to the attached block and frame using pull manipulator, causing the FEM model to deform (see the section "Pull Manipulator" in the ArtiSynth User Interface Guide).

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 3.4) to their respective FEM models.

As indicated in Section 3.3.2, 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 3.4) 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.

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:

```
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);
```

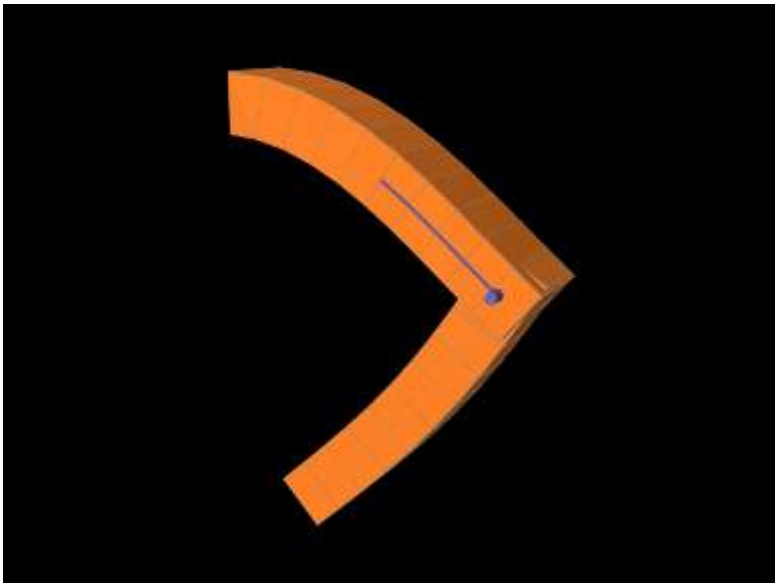


Figure 6.10: JointedFemBeams loaded into ArtiSynth and run until stable.

```
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.setAxisLength (0.8);
25         joint.setMinX (-0.5);
26         joint.setMaxX (0.5);
27         joint.setSlotWidth (0.61);
28         RenderProps.setLineColor (joint, myJointColor);
29         RenderProps.setLineWidth (joint, 3);
30         RenderProps.setLineRadius (joint, 0.04);
31      }
```

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 24-30 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 6.10. Forces can interactively be applied to the beams using pull manipulator (see the section “Pull Manipulator” in the ArtiSynth User Interface Guide).

Incompressiblity

FEM incompressibility within ArtiSynth is enforced by trying to ensure that the volume of a 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 incompressibilty. However, very large bulk modulus values will generally produce stability problems.

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 incomressibility, 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 imcompressibility 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.

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 incompressibilty, 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 situtations 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).

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 a 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).

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 transvere 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).

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 incompressibilty often results in less stable behavior, this is not always the case: in some situations the stronger enforcement afforded by hard incompressibility actually improves stabilty.

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.

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.

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 6.8.2 and 6.8.3, respectively.

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.

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 a FEM muscle.

```
1  // Create a muscle bundle
2  MuscleBundle bundle = new MuscleBundle("fibres");
3  Point3d[] fibrePoints = ...   //create a sequential list of points
4
5  // Add fibres
6  Point pPrev = fem.addMarker(fibrePoints[0]);    // create a FEM marker
7  for (int i=1; i<=fibrePoints.length; i++) {
8     Point pNext = fem.addMarker(fibrePoint[i]);
9
10    // Create fibre material
11    double l0 = pNext.distance(pPrev);          // rest length
12    AxialMuscleMaterial fibreMat =
13          new BlemkerAxialMuscle(
14                1.4*l0, l0, 3000, 0, 0);
15
16    // Add a fibre between pPrev and pNext
17    bundle.addFibre(pPrev, pNext, fibreMat);    // add fibre to bundle
18    pPrev = pNext;
19 }
20
21 // Enable use of fibres (default is disabled)
22 bundle.setFibresActive(true);
23 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.

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);
```

Example: comparison with two beam examples

An example comparing a fibre-based and a material-based muscle is shown in Figure 6.11. The code can be found in `artisynth.demos.tutorial.FemMuscleBeam`. 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 non-linear, 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.

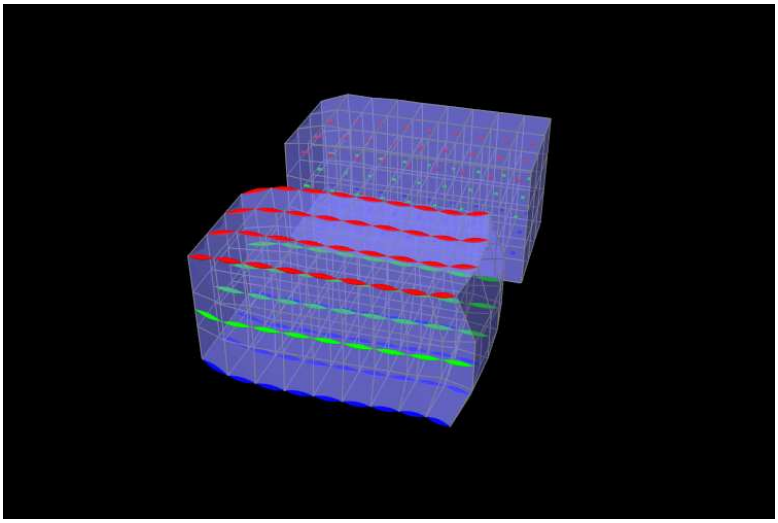


Figure 6.11: FemMuscleBeams model loaded into ArtiSynth.

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 a FEM. Any forces resulting from the collision are then automatically transfered back to the underlying nodes of the model using Equation (6.5).

Example: FEM collisions

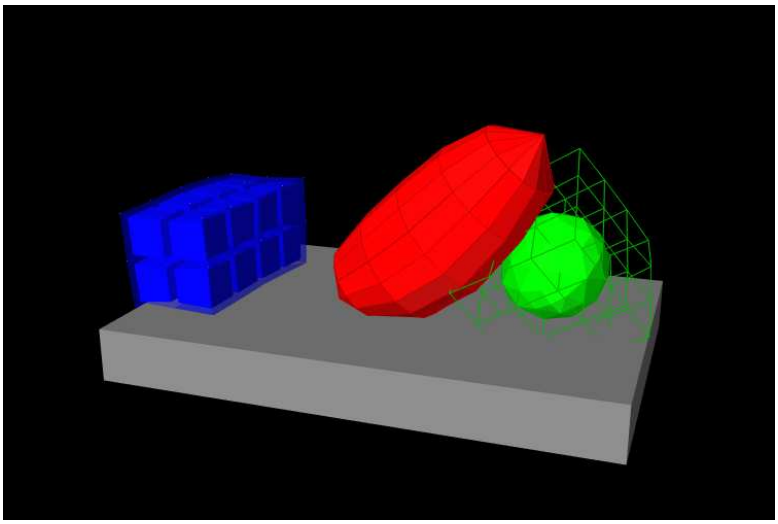


Figure 6.12: FemCollisions model loaded into ArtiSynth.

An example of FEM collisions is shown in Figure 6.12. 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.

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 6.5.

Table 6.5: 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`
`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 6.12 uses a widget size of 0.8, resulting in a mosaic-like pattern.

The next two properties in Table 6.5 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

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.

Example: stress and strain plotting

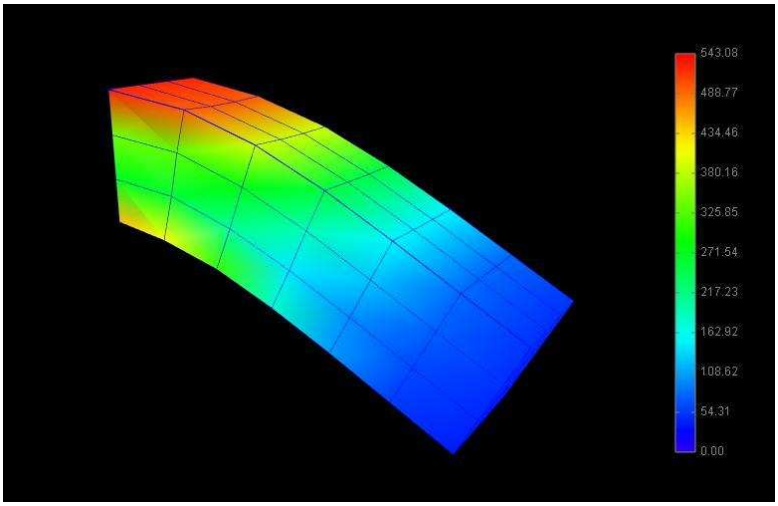


Figure 6.13: FemBeamColored model loaded into ArtiSynth.

The following model extends `FemBeam` to render stress, with an added color bar. The loaded model is shown in Figure 6.13.

```
package artisynth.demos.tutorial;

import java.io.IOException;

import maspack.render.RenderList;
import maspack.util.DoubleInterval;
import artisynth.core.femmodels.FemModel.Ranging;
import artisynth.core.femmodels.FemModel.SurfaceRender;
import artisynth.core.renderables.ColorBar;

public class FemBeamColored extends FemBeam {

   @Override
   public void build(String[] args) throws IOException {
      super.build(args);

      // Show stress on the surface
      fem.setSurfaceRendering(SurfaceRender.Stress);
      fem.setStressPlotRanging(Ranging.Auto);

      // Create a colorbar
      ColorBar cbar = new ColorBar();
      cbar.setName("colorBar");
      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          // Synchronize color bar/values in case they are changed
34          ColorBar cbar = (ColorBar)(renderables().get("colorBar"));
35          cbar.setColorMap(fem.getColorMap());
36          DoubleInterval range = fem.getStressPlotRange();
37          cbar.updateLabels(range.getLowerBound(), range.getUpperBound());
38
39          super.prerender(list);
40      }
41
42  }
```

Chapter 7

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 7.1.

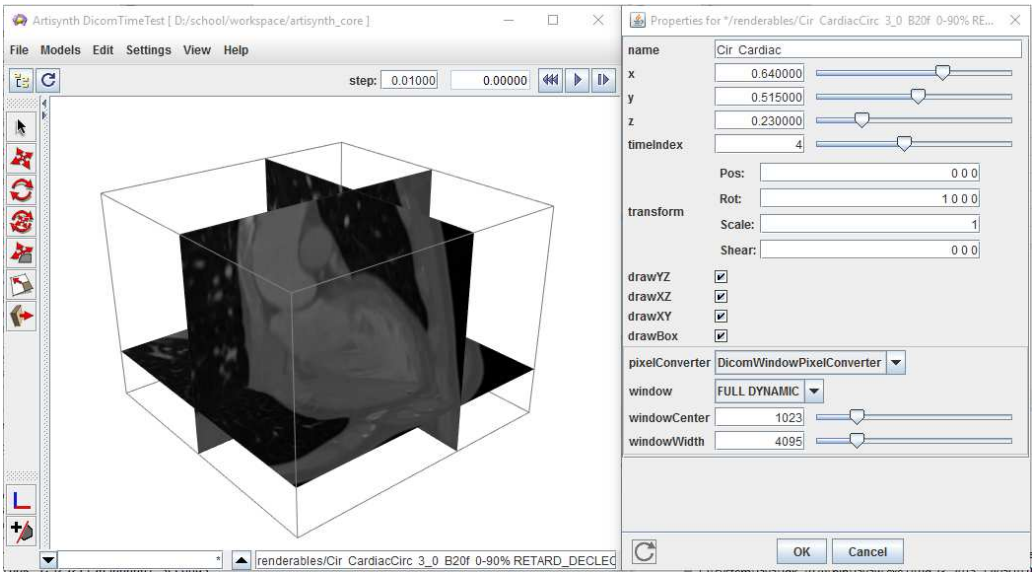


Figure 7.1: DICOM image of the heart, downloaded from http://www.osirix-viewer.com.

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 7.3.

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 7.1: 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 7.1)
- 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 7.2. A list of important attributes are provided in Table 7.2.

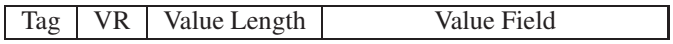


Figure 7.2: DICOM attribute structure

The DICOM classes

Each DicomElement represents a single attribute contained in a DICOM file. The DicomHeader contains the collection of `DicomElement`s 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.

`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 7.2: 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.

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`.

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.

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`.

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 `DicomSlice`s 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.

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.

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.

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 utilies 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.

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 retreived from the viewer using the `getImage()` method.

DICOM example

Examples of DICOM use can be found in the `artisynth.core.demos.dicom` package. These demos automatically download sample DICOM data from http://www.osirix-viewer.com/datasets/. The following listing provides one such example, loading an MR image of the wrist. Note that the image data in this example is encoded with the JPEG 2000 format, so ImageMagick is required to decode the pixels (see Section 7.3.2).

```
1  package artisynth.demos.dicom;
2
3  import java.awt.Color;
4  import java.io.File;
5  import java.io.IOException;
6  import java.util.regex.Pattern;
7
8  import artisynth.core.renderables.DicomViewer;
9  import artisynth.core.util.ArtisynthPath;
10 import artisynth.core.workspace.DriverInterface;
11 import artisynth.core.workspace.RootModel;
12 import maspack.image.dicom.DicomImageDecoderImageMagick;
13 import maspack.fileutil.FileManager;
14
15 /**
16  * Dicom image of the wrist, using ImageMagick to decode
```

```
17  *
18  */
19 public class DicomTestImageMagick extends RootModel {
20
21     String dicom_url = "http://www.osirix-viewer.com/datasets/DATA/WRIX.zip";
22     String dicom_folder = "WRIX/WRIX/WRIST RIGHT/T1 TSE COR RT. - 4";
23
24     public void build(String[] args) throws IOException {
25
26         // grab remote zip file with DICOM data
27         String localDir = ArtisynthPath.getSrcRelativePath(this, "data/WRIX");
28         FileManager FileManager = new FileManager(localDir, "zip:" + dicom_url + "!/");
29         FileManager.setConsoleProgressPrinting(true);
30         FileManager.setOptions(FileManager.DOWNLOAD_ZIP); // download zip file first
31
32         // download dicom image
33         File dicomPath = FileManager.get(dicom_folder);
34
35         // restrict to files ending in .dcm
36         Pattern dcmPattern = Pattern.compile(".*\\.dcm");
37
38         // add DicomViewer
39         DicomViewer dcp = new DicomViewer("Wrist", dicomPath, dcmPattern);
40         addRenderable(dcp);
41
42     }
43 }
```

Lines 27–33 are responsible for downloading and extracting the sample DICOM zip file. In the end, `dicomPath` contains a reference to the desired DICOM directory on the local system. On line 36, we create a regular expression pattern that will only match files ending in `.dcm`. On line 39, we create the viewer, which will automatically parse the desired DICOM files and create a `DicomImage` internally. We then add the viewer to the model for display purposes. This model is displayed in Figure 7.3.

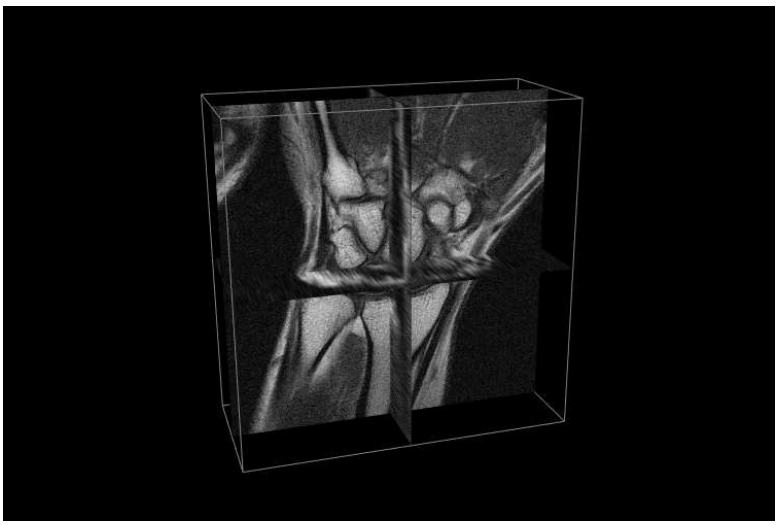


Figure 7.3: DICOM model of the wrist, downloaded from http://www.osirix-viewer.com.

Appendix A

Mathematical Review

This appendix reviews some of the mathematical concepts used in this manual.

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:

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix},$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix},$$

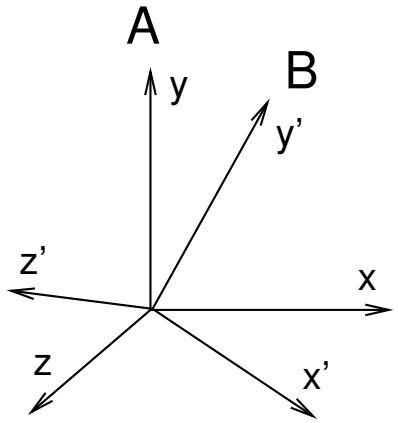


Figure A.1: Two coordinate frames A and B rotated with respect to each other.

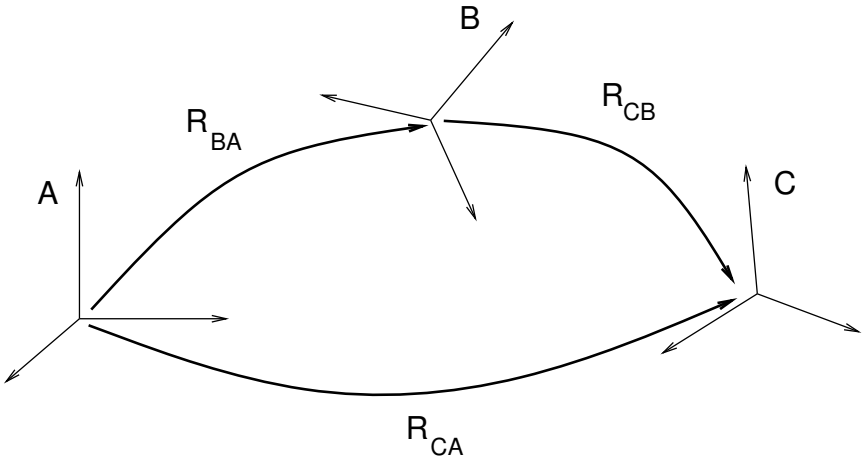


Figure A.2: Schematic illustration of three coordinate frames A, B, and C and the rotational transforms relating them.

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

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: $\mathbf{R}_z(\phi)\,\mathbf{R}_y(\theta)\,\mathbf{R}_z(\psi)$.

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.

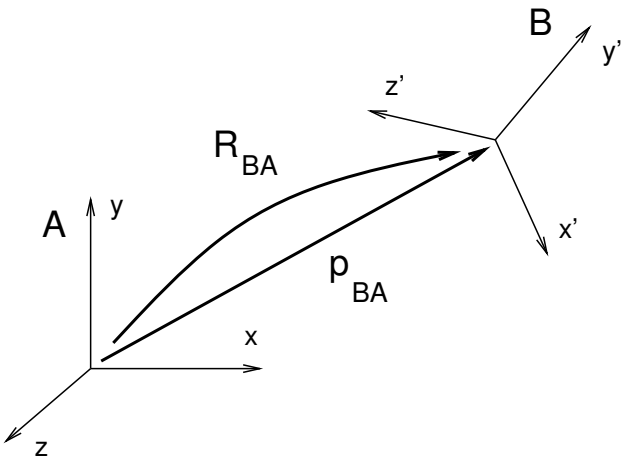


Figure A.3: A position vector $\mathbf{p}_{BA}$ and rotation matrix $\mathbf{R}_{BA}$ describing the position and orientation of frame B with respect to frame A.

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

$$ {}^{A}\mathbf{q}^{*} = \mathbf{T}_{BA}\,{}^{B}\mathbf{q}^{*} \quad \text{and} \quad {}^{B}\mathbf{q}^{*} = \mathbf{T}_{BA}^{-1}\,{}^{A}\mathbf{q}^{*} $$

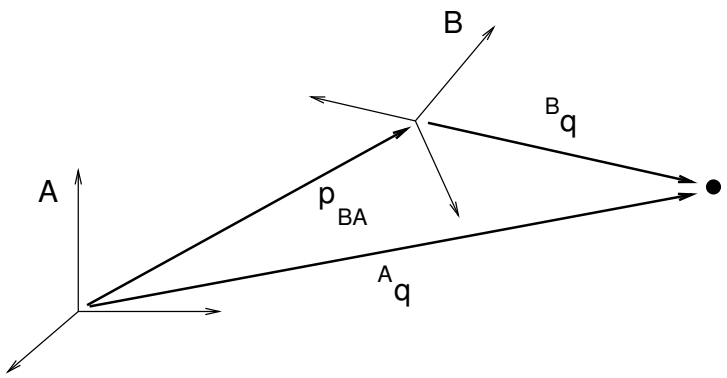


Figure A.4: Point vectors ${}^A\mathbf{q}$ and ${}^B\mathbf{q}$ describing the position of a point $\mathbf{q}$ with respect to frames A and B.

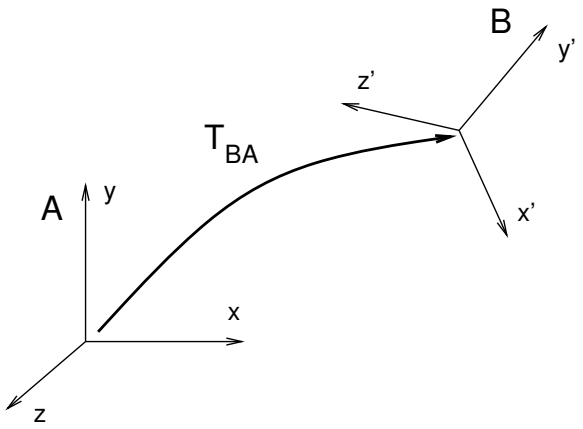


Figure A.5: The transform matrix $\mathbf{T}_{BA}$ from B to A.

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

$$\mathbf{T}_{CB} = \mathbf{T}_{BA}^{-1}\, \mathbf{T}_{CA} = \mathbf{T}_{AB}\, \mathbf{T}_{CA}. \tag{A.17}$$

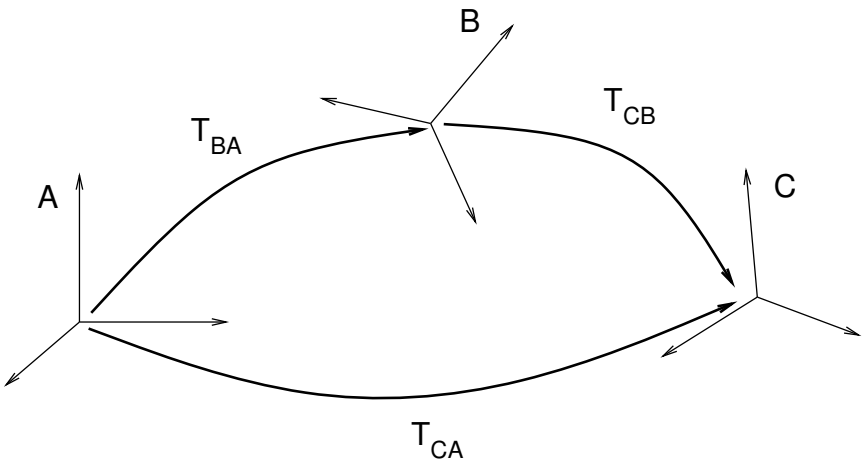


Figure A.6: Three coordinate frames A, B, and C and the transforms relating each one to the other.

Affine transforms

An *affine transform* is a generalization of a rigid transform, in which the rotational component $\mathbf{R}$ is replaced by a general 3×3 matrix $\mathbf{A}$. This means that an affine transform implements a generalized basis transformation combined with an offset of the origin (Figure A.7). As with $\mathbf{R}$ for rigid transforms, the columns of $\mathbf{A}$ still describe the transformed basis vectors $\mathbf{x}'$, $\mathbf{y}'$, and $\mathbf{z}'$, but these are generally no longer orthonormal.

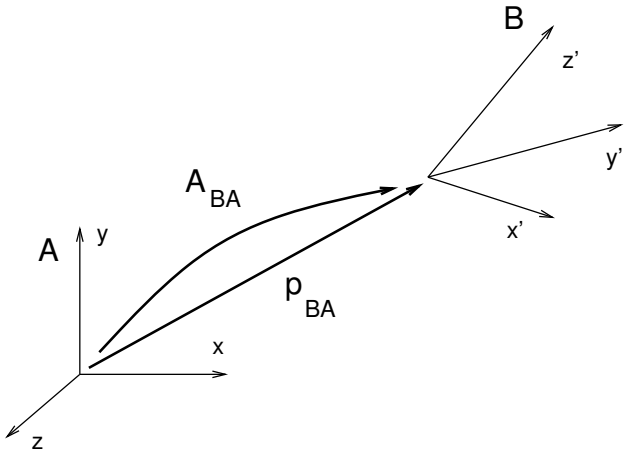


Figure A.7: A position vector $\mathbf{p}_{BA}$ and a general matrix $\mathbf{A}_{BA}$ describing the affine position and basis transform of frame B with respect to frame A.

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 $\mathbf{A}$ is applied and the translation component $\mathbf{p}$ is ignored.

Affine transforms are typically used to effect transformations that require stretching and shearing of a coordinate frame. By the polar decomposition theorem, $\mathbf{A}$ can be factored into a regular rotation $\mathbf{R}$ plus a symmetric shearing/scaling matrix $\mathbf{P}$:

$$\mathbf{A} = \mathbf{R}\mathbf{P} \tag{A.20}$$

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 $\det\mathbf{A} = -1$.

Rotational velocity

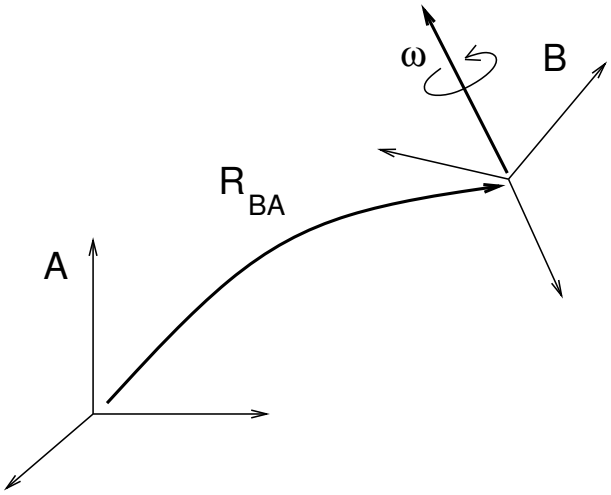


Figure A.8: Frame B rotating with respect to frame A.

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

$$\omega_{AB} = -\omega_{BA}. \tag{A.23}$$

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

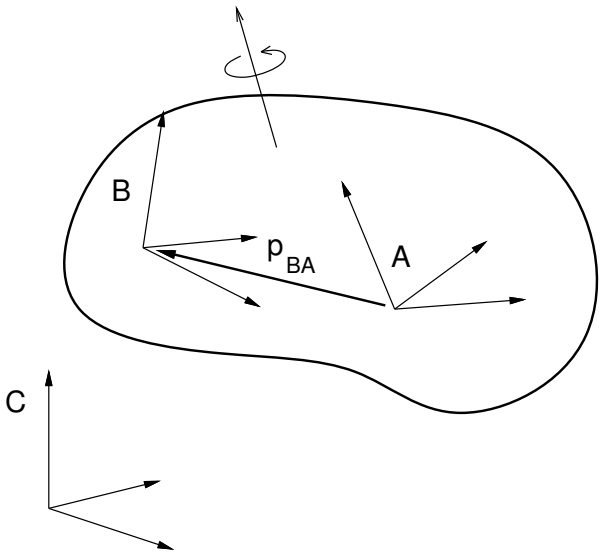


Figure A.9: Two frames A and B rigidly connected within a rigid body and moving with respect to a third frame C.

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

$$\mathbf{X}^{*}_{BA} \equiv \begin{pmatrix} \mathbf{R}_{BA} & 0 \\ [\mathbf{p}_{BA}]\mathbf{R}_{BA} & \mathbf{R}_{BA} \end{pmatrix}. \tag{A.29}$$

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

Bibliography

[1] Mihai Anitescu and Florian A. Potra. A time-stepping method for stiff multibody dynamics with contact and friction. *International Journal for Numerical Methods in Engineering*, 55(7):753–784, 2002.

[2] Silvia S Blemker and Scott L Delp. Three-dimensional representation of complex muscle architectures and geometries. *Annals of biomedical engineering*, 33(5):661–673, 2005.

[3] J. Bonet and R. D. Wood. *Nonlinear continuum mechanics for finite element analysis*. Cambridge University Press, 2000.

[4] 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.

[5] 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*, pages 355–394. Springer, 2012.

[6] 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)*, Feb. 2008. poster.

[7] Florian A. Potra, Mihai Anitescu, Bogdan Gavrea, and Jeff Trinkle. A linearly implicit trapezoidal method for integrating stiff multibody dynamics with contact, joints, and friction. *International Journal for Numerical Methods in Engineering*, 66(7):1079–1124, 2006.