May 9, 2012

All documentation converted to LaTeX

All Artisynth documentation that was formerly maintained using *AsciiDoc* has been converted to LaTeX, with HTML output produced by LaTeXML. If you update `$ARTISYNTH_HOME/doc`, you will see that all the `.txt` files have been replaced with `.tex` files. For details about the changes, please see the (revised) Documentation Manual.

The new HTML files have already been uploaded to www.artisynth.org, and the documentation area now provides PDF files as well.

The move from *AsciiDoc* was done for several reasons:

- Our use of *AsciiDoc* required a lot of customization that was becoming very hard to maintain.
- *AsciiDoc* was difficult to install, while LaTeX and LaTeXML, on the other hand, are relatively easy to install and use.
- LaTeX is a fairly stable and standard environment that many people are already familiar with.
- The math support for *AsciiDoc* did not work out as well as expected.

In the end, the emergence of LaTeXML as a relatively reliable LaTeX to HTML converter prompted the change.

May 1, 2012

Materials added to Axial Springs

In order to better modularize the behavior of point-based springs and muscles, we are replacing the various force parameters (such as stiffness and damping) with a material object that encapsulates these parameters and can be exchanged with other materials to provide different force/length behaviors. This material will behave analagously to the materials used in FEM models. All materials for point-based springs will be subclasses of `AxialMaterial`, which is in turn a subclass of `MaterialBase`.

This first part of this transition is complete: a `material` property has been added to the base class for point-based springs, and the linear parameters `stiffness`, `damping`, and `restLength` for `AxialSpring` and `MultiPointSpring` have been replaced with a linear material object called `LinearAxialMaterial`.

For the moment, the setters and getters for `stiffness`, `damping`, and `restLength` have been left in place, with these methods accessing an underlying linear material. Also, a convenience method has been added which allows you to set a linear material:

```
setLinearMaterial (stiffness, damping, restLength);
```

The next step in this process will be to implement materials for the various muscle types.

Material package moved

The package `artisynth.core.femmodels.materials` has been moved into `artisynth.core.materials`, which now also contains the new materials for point-based springs.

April 26, 2012

Changes to the model advance framework

There have been some significant changes to the model advance framework:

- The maximum step size of the root model (returned by `RootModel.getMaxStepSize()`) is now used to control the overall simulation advance rate. Models located under the root model will be advanced at this rate, unless they specify a smaller step size using `getMaxStepSize()`.
- `Scheduler.setStepTime()` has been removed. Instead, you can call the root model method `setMaxStepSize()`, or `Main.setMaxStep()`. The command line argument `-singleStepTime` has also been changed to `-maxStep`, and now determines the default maximum step size for root models.
- Models can now leave their maximum step size undefined by having `getMaxStepSize()` return -1. In this case, the model will be advanced using the root model step size.
- The default maximum step size for the root model is 0.01. This can be overriden by specifying a different step size in the root model's constructor, or by using the `-maxStep` command line argument.
- Output probes can now be associated with models located under the root model. Previously, only input provbes could be associated with a model. Output probes that are associated with a model will called immediately after that model is advanced.
- If the update interval for an output probe is undefined (i.e., `getUpdateInterval()` returns -1), then it's `apply()` method is called at the rate determined by its model's effective step size (the minimum of the root model step size, or the value returned by `getMaxStepSize()`, if defined).
- `setDefaultInputs()` has been removed. The small amount of code that was declared for that method has been moved into the `advance()` methods of the respective models.
- `Model.initialize()` is now called whenever the system is reset to a particular time (such as when going to a WayPoint location). Previously, it had been called only when starting simulation at time t = 0.

Full details about model advancement are contained in the (fledgling) ArtiSynth Reference Manual.

Adaptive stepping

Adaptive stepping is now supported. If a model's `advance()` method returns a value s < 1, then this indicates that the current time step is too large and should be reduced. The system will then reduce the step size, restore the model's state, and retry the advance.

The value returned by `advance()` can also recommend how much to reduce the step size by. If s = 0, no recommendation is made, but for 0 < s < 1, it is recommended to reduce the step size by scaling by s. A value of s >= 1 indicates that the advance has succeeded, with s > 1 recommending to increase the step size by scaling by s.

Once an advance succeeds, the system will try to incrementally increase the step size back to its nominal value.

Adaptive step sizing is enabled or disabled using the `adaptiveStepping` property of `RootModel`. It is enabled by default.

`MechModel` has been adjusted to request adaptive stepping when collision distances exceed a theshhold defined by the `collisionLimit` property, and `FemModel3d` has been adjusted to request adaptive stepping when it encounters inverted elements.

As an example of the latter, you can run the `FemMuscleTongue` demo with probes enabled and `FemModel3d.abortOnInvertedElems` set to `false` (i.e., omit the command line argument `-abortOnInvertedElems`). The model experiences some slight instability around t = 1, but completes the simulation without element inversion.

Full details on adaptive stepping are given in the reference manual.

State for Probes, Monitors and Controllers

Probes, monitors, and controllers can now contain state. Details are given in the reference manual.

April 21, 2012

Mesh Bodies

Mesh bodies (type `MeshBody`) have been added to `MechModel`. A mesh body is an object consisting primarily of a polygonal mesh (type `PolygonalMesh`). It is an abstract class for which two concrete subclasses are currently implemented: `FixedMesh`, which defines a fixed mesh shape, and `SkinMesh`, which defines a mesh whose shape is defined by the motion of a set of rigid bodies through a skinning algorithm. Both classes are defined in `artisynth.core.mechmodels`.

At present, `SkinMesh` implements a simple linear skinning. More sophisticated algorithims may be supported later. Once created, it is necessary to set the weights for a `SkinMesh`. If the weights are known, they can be set with the `setWeights()` method. Otherwise, they can be computed automatically with the method `computeWeights()`, which determines the weights from the current mesh and body positions, based on the nearest distance of each vertex to each body.

The demo `artisynth.models.mechdemos.SkinDemo` shows a simple `SkinMesh`.

> Note that `MeshObject` and its subclasses are likely to undergo significant changes since this is still a fairly experimental component.

April 10, 2012

Time type changed from long (nanoseconds) to double (seconds)

The basic type used to denote time in Artisynth has been changed from a `long` giving the time in nanoseconds, to a `double` giving the time in seconds.

In particular, the following methods now receive or return time as a double value in seconds:

```
Model.initialize (t0)
Model.setDefaultInputs (t0, t1)
Model.advance (t0, t1, flags)
Model.getMaxStepSize()

Probe.apply (t)
Controller.apply (t0, t1)
Monitor.apply (t0, t1)
```

The orginal reason for representing time using an integer nanosecond quantity was to make it easy to determine precisely the sequence of timeline events without worrying about round-off error. However, in practice, this approach was cumbersome, difficult to read, and required most methods to convert nanoseconds into a double quantity internally.

Instead, to handle round-off issues, `TimeBase` now provides the following methods to compare and manipulate time quantities within a fixed tolerance (currently set to one picosecond, or 1e-12):

```
TimeBase.equals (t0, t1)
TimeBase.compare (t0, t1)
TimeBase.modulo (t0, t1)
TimeBase.round (t)
```

These methods are used by the scheduler to sequence timing events in a precise way.

Time quantities in probe files and probe data files are now written in seconds instead of nanoseconds. All `.art` files that are currently checked in have been converted. For backward compatibility, integer time quantities (i.e., those not containing a decimal point) are still read as nanoseconds and converted to seconds.

Extra Toolbar

A toolbar has been added to the top of the main Artisynth frame, under the menu bar, and is used to contain icons that were previously contained in the menu bar.

April 4, 2012

A large number of changes have been made as part of a general refactoring of the physics solver. Many of the changes are "under the hood", but the following are visible to developers:

Changes to Model.advance

The signature of `Model.advance()` has been modified in prepartion for adaptive step sizing. It now returns a double value that will be used to indicate desired changes in step size. If no step size change is desired the method should return 1. All declarations of `advance` have been altered to ensure that 1 is presently returned.

A `flags` argument has also been added, although this is not expected to be used much except internally.

Removal of effective mass and inertia

The effective mass and spatial inertia fields of particles and rigid bodies have been removed. These had been used to store the "effective" masses and inertias that resulted from attaching particles to these objects. The effective mass calculation is now done within the solver itself.

Rigid body velocites and forces now integrated in world coordinates

The solver now integrates rigid body velocities and forces in world-rotated coordinates (i.e., a coordinate frame coincident with the body frame but with an orientation aligned with world coordinates). Before, velocities and forces were integrated strictly in body coordinates. This change was made for several reasons:

- The coriolis force terms are less complex when using world-rotated coordinates, which allows for more accurate integration.
- It makes the velocity seen by the solver indentical to the internal rigid body velocity state (which also uses world-rotated coordinates).
- Velocities in body coordinates are not independent of body position, which makes it difficult to save and restore velocity state exactly.
- World-rotated coordinates are easier for a user to conceptualize.

The only disadvantage to using world-rotated coordinates is that the spatial inertia matrix is no longer constant. However, this is not a major issue since the inertia matrix is easy to compute and invert, particularly since it differs from the constant body-centric inertia by only a rotational similarity transform.

> **Note:**
> The `velocity` property of a rigid body is not affected by this change, since this was always presented in world-rotated coordinates.

Save and restore model state now properly implemented

Save and restore state for MechModels and FemModels is now properly implemented, and in particular properly handles collision and viscoelastic state. This means that backtracking to a waypoint and then advancing should yield results identical to when the waypoint was first traversed.

Changes to the model and probe file formats

Some of the `scan` and `write` methods for saving and loading models from a file have been refactored to simplify the code.

The file format for AxialSpring and Muscle objects has been changed: the old format whereby parameter values are specified as a simple untagged list of numeric values is no longer supported.

There has also been a change in the file format for probes: the field `element`, which identifies the model associated with the probe, has been renamed to `model`, so that entries such as

```
element=models/xxx
```

now appear as

```
model=models/xxx
```

All the `.art` files which are currently checked in have been patched to reflect this change.

Removal of local position correction

The position correction code, used to stablize bilateral and unilateral constraints, has been refactored. All position correction is now done globally by the solver itself. Local position correction, which was done using ad-hoc model-specific methods, and was formerly available using the command line option

```
-posCorrection Local
```

has been removed.

Time display

A time display box has been added to the main frame.

January 4, 2012

New release, ArtiSynth 2.8

A new release has been put on the website. Main changes include:

- moving the inverse simulation code to `artisynth.core.inverse`.
- creating a set of inverse demos in `artisynth.models.inversedemos` (thanks to Ian for this).
- removal of old shared libraries.
- updating the installation documentation and making the Eclipse installation easier.

October 13, 2011

Automatic creation of CompositePropertyPanels

A `CompositePropertyPanel` for a widget is now created automatically for composite properties that export the static method `getSubClasses()` (which returns a list of the various instances of said composite property that can be instantiated by the panel). This replaces the need to create property-specific composite panels such as `MaterialPanel`, `MuscleMaterialPanel`, etc.

Specifying expandibility for property widgets

The `PropertyInfo` structure, which provides information about properties, has been augmented with the method `getWidgetExpandState()`, which returns a code describing if the property's widget should be able to expand or contract in order to save GUI space, and if so whether it should be initially expanded or contracted. The settings for this can be specified using the flags `XE` (initially expanded) `CE` (intially contracted) in the property declaration options string.

Creating FemModels from surface meshes

Functionality has been added to `FemFactory` allowing FEM models to be created directly from a surface mesh, using Tetgen called via the `TetgenTessellator` class. The relevant method is

```
FemFactory createFromMesh (model, surface, quality)
```

This functionality has also been added to the FemModel editing panel, allowing to you specify a surface mesh in addition to other options.

October 5, 2011

Passive sections added to MultiPointSpring

It is now possible to specify certain sections of a `MultiPointSpring` to be *passive*, meaning that they will not contribute to the total length used to determine the spring's force. The relevant methods within `MultiPointSpring` are:

```
setSegmentPassive (idx, passive)
isSegmentPassive (idx)
clearPassiveSegments()
```

where `idx` is an index identifying the segment between points `idx` and `idx+1`.

September 10, 2011

Updates to PolygonalMesh

`PolygonalMesh` has been updated to include an `addMesh()` method that allows meshes to be combined. This is based on some code that Ian recently wrote. Also, the `triangulate()` method has been rewritten so that texture and normal information (if present) will be preserved.

September 8, 2011

Articulation constraints preserved when manipulating bodies

A feature has been added that allows articulated body constraints to remain enforced when rigid bodies are moved using the translation and rotation manipulators. To enable this feature, you can specify `-useArticulatedTransforms` on the artisynth command line. The feature can also be enabled using the articulation icon located below the manipulator icons on the left side of the main viewer.

Properties now validated using a getRange method

The method for validating property values has been changed. Previously, if property `xxx` had restricted values, the application could define a `validateXxx()` method to validate proposed values for that property. Now, instead, the application should define a `getXxxRange()` method that returns a `Range` object for the property. The result from this method will then be returned through the `getRange()` method of the Property handle itself. The `Range` object contains an `isValid()` method that can be used to validate values.

For numeric properties, a user can still define a default numeric interval range of the form "[lo,hi]" in the options string of the property's definition. If no `getXxxRange()` is defined, then the property's `getRange()` method will return this interval instead. The default numeric range is also used to determine bounds on slider widgets attached to the property, in cases where the upper or lower limits returned by any `getXxxRange()` method are unbounded.

The main reason for reformulating property validation to use `Range` objects is that ranges can be combined using their `intersect()` method. Then if a widget is controlling the same property on several objects, it is possible to determine a range that is acceptable to all objects.

NumericRange, DoubleRange and IntRange have been renamed

The classes `NumericRange`, `DoubleRange` and `IntRange` have been renamed to `NumericInterval`, `DoubleInterval` and `IntegerInterval`, and have been moved from `maspack.property` to `maspack.util`, along with `Range`.

> **Important:**
> This has caused a change in the file format, since `.art` files sometimes make direct reference to these class names. The format has been corrected for any `.art` files that were checked in.

Angle coordinates added to revolute and spherical joints

Revolute and spherical joints have now been augmented with angle variables that represent generalized coordinates associated with the degrees of freedom allowed by these joints. In particular, the class `RevoluteJoint` is associated with an angle `theta`, and a new class of spherical joint, called `SphericalRpyJoint`, has been defined that allows its orientation to be controlled using `roll`, `pitch`, and `yaw` angles. Demos of both can be found in `RevoluteJointDemo` and `SphericalJointDemo`, both in `artisynth.models.mechdemos`.

The joint angles are exposed as properties, which can be used to get or set their values in degrees. Setting the properties will cause the joint to move by changing the position of one of the rigid bodies to which it is attached. In determining which body to move, the system tries to identify one which is "free", i.e., has no connections to ground, either directly or indirectly via other bodies in the articulation chain. Moreover, when moving such a free body, all other bodies connected to it are moved in unison.

Joint angles are not bounded by the usual range of ± 180 degrees, and their values will grow indefinitely as a joint continues to "wind". This is acheived by placing a state variable in the constraint that keeps track of the most recent joint value. Joint angles can also be given range limits, which themselves are controlled by properties such as `getThetaRange`, `getRollRange`, etc. By default, the angles have no limits, corresponding to a range of [-inf,inf]. Note that limits with a range greater than 360 degrees, such as [-400,400], are perfectly feasible, and often occur in mechanical systems as an artifact of gearing.

> The spherical joint represented by `SphericalRpyJoint` models a gimble system in which rotation is achieved by a `roll` rotation about the z axis, followed by a `pitch` rotation about the new y axis, followed by a `yaw` rotation about the final x axis. Such gimble systems experience restricted motion and instability when the pitch angle is close to ± 90 degrees.

Connector scaling fixed

Problems associated with scaling certain kinds of connectors (such as `SegmentedPlanarConnector`), and models containing them, have been fixed.

Frame information added to FemElement3d

It is now possible to add frame information to a `FemElement3d`, using the methods

```
FemElement3d.setFrame (Matrix3dBase M)
Matrix3d FemElement3d.getFrame()
```

The frame information can be specified using any `Matrix3d` type (such as `RotationMatrix3d`) and is mainly intended for use in computing anisotropic material behaviors. At present, the frame information is stored repeatedly at each of the element's integration points, within the point's `IntegrationData3d` object. The ability to store individual frame information at each integration point may be added in the future.

Frame information maybe accessed by the `computeStress` and `computeTangent` methods of `Material`, which now are now supplied with the integration point's `IntegrationData3d` structure as an additional parameter:

```
public void computeTangent (
   Matrix6d D, IntegrationPoint3d pt, IntegrationData3d dt);

public void computeStress (
   SymmetricMatrix3d sigma, IntegrationPoint3d pt, IntegrationData3d dt);
```

and the frame information itself can be obtained using

```
dt.getFrame()
```

Slider ranges and out-of-range values

The procedures for automatically determining ranges for slider widgets have been reworked. In addition, in some cases (particularly involving the joint angle properties described above) the actually value for a slider widget may lie outside the slider's range. When this happens, the slider background is changed to a dark gray color, indicating that subsequent adjustments to the slider may produce a jump in the property's value.

August 9, 2011

Ability to duplicate FEM models

It is now possible to duplicate a `FemModel3d`. Simply select the model, choose Duplicate from the context menu, and click in the viewer where you want to model to appear.

In code, one can do

```
FemModel3d femCopy = (FemModel3d)fem.copy (0, null);
```

Ability to merge FEM models

A method has been added to `FemFactory` which allows you to add a copy of a FEM model to an existing FEM model:

```
addFem (fem0, fem1, nodeMergeDist)
```

This creates copies of the nodes, elements, markers, and attachments of `fem1` and adds them to `fem0`. It will also merge nodes that are within a certain distance of each other: if `nodeMergeDist` >= 0, then if a node in `fem1` has a nearest node in `fem0` within a distance of `nodeMergeDist`, then the `fem0` node will be used instead of copying the `fem1` node.

For a demo of this, see the new demo HexFrame.

August 3, 2011

Attaching FemNodes to other FEM models

Support has now been added to allow you to attach an `FemNode` directly to an element of another FEM model (in the same way that a FemMarker can be attached to an element).

For the demo, run `AttachDemo`, select one of the FEMs, and then choose Attach particles from the context menu. A dialog will then appear which allows you to select the nodes (and other particles) to attach. By default, the particles will be attached either to their containing element, or projected onto the nearest surface element. If you select project points onto surface, then the particles will always be projected onto the nearest surface element, which can be a useful option if the particles are inside the FEM you want to connect to. The FEMs must be contained within a MechModel.

I've made some attempt to ensure reasonable behavior if you move either an attached point (or one of the nodes to which it is attached) using the dragger fixtures. However, the element will not (yet) change with these operations and so you may end up with an attachment that lies outside the element. This is not necessarily a bad thing, and in fact I noticed that you can place attachments outside an element and still get reasonable behavior.

For the API, the main methods are:

```
MechModel.attachPoint (Point p1, FemModel3d fem, double reduceTol)
MechModel.attachPoint (Point p1, FemElement3d elem)
```

The former finds the element to attach to, while the latter assumes you know the element already. See the Javadocs for more info.

To locate an element within an FEM, you can use

```
FemModel3d.findContainingElement (Point3d pnt)
FemModel3d.findNearestSurfaceElement (Point3d loc, Point3d pnt)
FemModel3d.findNearestElement (Point3d loc, Point3d pnt)
```

Again, see the Javadocs for more info.

June 26, 2011

FEM muscles integrated with linear materials

FEM-based muscle forces (implemented in `FemMuscleModel` using different kinds of `MuscleMaterial`) now work with linear base materials. In addition, stress and strain plotting now also works with linear materials.

One caveat is that all currently implemented MuscleMaterials are quite non-linear, so it is not clear how useful this will be. One could implement a companion linear-type MuscleMaterial. Alternatively, if you use `GenericMuscle` with `expStressCoef` and `fibreModulus` set to 0, you will get a very basic behavior that simply applies a uniform, activation-proportional stress along the muscle direction.

June 2, 2011

Controllers added

`Controller` objects have just been added to Artisynth. They are the complement of `Monitors`.

Like a Monitor, a Controller contains a single function

```
apply (t0, t1)
```

that is called before the `advance` routine of an associated model. Times `t0` and `t1` denote the start and end times associated with the time step. You can add/remove controllers from the `RootModel` using

```
addController (controller)
addController (controller, model)
removeController (controller)
```

The first method adds a "free" controller that is called before all the advance methods. The second method adds a controller that is called before the advance method of a particular model (this probably won't be used much).

> **Important:**
> The `apply` method for a Monitor now takes two time arguments as well (it used to take only a `t0` argument). Also, Monitors are now called *after* the `advance` method, so you might want to convert any previous Monitors that you were using to Controllers.

April 21, 2011

Jython console updated

The Jython console has been fixed to properly handle loops within scripts. Previously, all the output from within a loop was printed only when the loop finished. Output is now correctly routed to the console as it is generated. Also, scripts can now be nested. Typing '`^C`' in the console window will also about cause a script to abort, although only after the return of any blocking call.

Jython has also been updated to 2.5.2. Since the jython jar file is bundled with ArtiSynth, I don't *think* this will require anyone to explicitly upgrade to Jython 2.5.2 on their system, although that might not be a bad idea.

I still notice an occasional crash when loading models in a script. This seems to occur inside GUI code associated with the model creation, and may be related to the fact the construction and handling of GUI components should in theory be done only within the GUI thread. If this starts causes anyone trouble, I'll try to investigate further.

April 14, 2011

Target positions and velocities

Both the `Point` and `Frame` components have been augmented with properties to describe target positions and velocities. For a Point, we have

targetPosition Target position for the point.

targetVelocity Target velocity for the point.

while for a Frame (which includes `RigidBody`), we have

targetPosition Target position for the frame's origin.

targetOrientation Target orientation for the frame (specified as an `AxisAngle`).

targetPose Complete target pose for the frame (specified as a `RigidTransform3d`).

targetVelocity Target translation and angular velocity for the frame.

Another property, called `targetActivity`, is supplied to control which of the position and velocity targets are actually active. This allows the user of the target data (e.g., the solver) to know whether it should interpolate position data, velocity data, or both. The settings for `targetActivity` are:

Position The position target is active, while the velocity target is inactive and tracks the current velocity.

Velocity The velocity target is active, while the position target is inactive and tracks the current position.

PositionVelocity Both the position and velocity targets are active.

None Both the position and velocity targets are inactive.

Auto Both the position and velocity targets are initially inactive, but will become active when their values are set. This is the default setting.

> **Note:**
> `Position` and `Velocity` target activity refer to *generalized* positions and velocities. In particular, for Frames, `Position` activity refers to `targetPosition`, `targetOrientation`, and `targetPose`. One way to resolve this ambiguity might be to rename the `position` property of a Frame to something like `translation`.

Specifying a target position and/or velocity is now the preferred way to control the motion of one of these components parametrically: If the component is set to be non-dynamic, then the target position and/or velocity is used by the simulator to control the component's motion, and its actual position and/or velocity will be matched to the target over the coarse of the next time step.

Current plans also call for targets to be used to specify the desired motions of dynamic and attached components (such as markers) for tracking by a controller (e.g., inverse actuator control).

Proper interpolation for rotations

As part of implementing target orientations for Frames, it was necessary to construct a proper interpolation methods for rotations. These are now available for probes which control the orientation of a Frame, through either `orientation` or `pose` properties. The interpolation methods include

SphericalLinear Interpolates between two orientations by finding the axis-angle that separates them and then uniformly interpolating the angle about this axis (this is the *slerp* method that was described by Ken Shoemake at SIGGRAPH 1985).

SphericalCubic Smoothly interpolates between orientations by taking into account estimated angular velocities. The method used is described in "A general construction scheme for unit quaternion curves ...", by Kim, Kim and Shin at SIGGRAPH 1995.

> The above interpolations are actually enabled for numeric data probes with vector sizes of 4 and 16. For the former, the data is assumed to be an orientation in AxisAngle format. For the latter, the data is assumed to be the 4x4 matrix associated with a RigidTransform3d, with the rotation interpolated as described above and the translation interpolated using standard linear or cubic methods.

Displacement properties added to FemNode3d

`FemNode3d` now has two additional properties:

displacement A read-only property giving the displacement of the node from the rest position.

targetDisplacement An alternate way of specifying a target position relative to the rest position.

March 2, 2011

Constrained motions have been added to draggers. If you press SHIFT while moving a dragger, then rotations are constrained to multiples of 5 degrees, and translations are constrained to multiples of a step size determined as follows (this may be improved):

- If the viewer grid is visible, then the step is the size of the smallest grid cell.
- Otherwise, the step is 1/10 of the size of the dragger.

Feb 3, 2011

Changes to RenderProps

A new style for rendering lines, `SOLID_ARROW`, has been added. Also, the following render properties have been renamed:

cylinderRadius Renamed to lineRadius

cylinderSlices Renamed to lineSlices

sphereRadius Renamed to pointRadius

sphereSlices Renamed to pointSlices

Finally, two new properties, edgeWidth and edgeColor, have been added, but are currently only used for rendering contact information, as described below.

Rendering contact normals and contours

Support has been added for rendering the intersection contours and contact normals associated with collisions. This rendering is controlled by the render properties associated with `MechModel.collisionHandlers()`.

By default, contact and contour rendering is disabled. To enable it, one can use the following code fragment:

```
RenderProps.setVisible (mechModel.collisionHandlers(), true);
```

The following render properties are used:

lineStyle Style of the line used for rendering the contact normals

lineWidth Width (in pixels) of the contact normal if the `Line` line style is used

lineRadius Radius of the contact normal if a solid line style is used

lineSlices Number of slices in the contact normal for a solid line style

lineColor Color of the contact normal

edgeWidth Width (in pixels) of the line used to render the contour

edgeColor Color of the contour

> **Note:**
> Contours will only be rendered in Andrew Larkin's collision code is enabled, i.e., `-useAjlCollision`.

These properties can be set in the same way as the visibility, e.g.,

```
Renderable collisions = mechModel.collisionHandlers();
RenderProps.setEdgeWidth (col, 2);
RenderProps.setEdgeColor (col, Color.Red);
```

To access them on a read-only basis, one can do

```
RenderProps props = mechModel.collisionHandlers().getRenderProps();
```

Finally, to set the length of the rendered contact normals, set the `contactNormalLen` property in `MechModel`. Since contact normals have no preferred direction, it may be necessary to use a negative length value in order to visualize them properly.

> For a demo, run the model DentalCasts (artisynth.models.articulator.DentalDemo), and set the top cast invisible to see the contact interactions.

> **Note:**
> The artisynth command line option `-renderCollisionContours` has been removed.

Jan 31, 2011

User interface guide completed

The ArtiSynth UI Guide is now complete, and contains detailed descriptions of most of the interactions and editing operations available through the GUI. In particular, all of the editing panels are now documented. The user interface guide can be obtained from the website, or directly through

`http://www.artisynth.org/doc/html/uiguide/uiguide.html`

New editing features for FemMuscleModel

A new `MuscleElementAgent` allows elements to be added to `MuscleBundles` contained within a `FemMuscleModel`. Other menu-based features allow you to automatically set the direction vectors in the elements, or add elements that are a certain distance from the fibres. For details, see the UI Guide, under "Editing Muscle Bundles".

Exclusive open mechanism for editing panels

A lock mechanism has been introduced to enable some editing panels to function on an "exclusive open" basis, whereby only one can be open at a time. This is useful for panels that modify the GUI state, and for which simultaneous panels could lead to unexpected side effects. Edit operations that cannot open because of the exclusivity lock will still appear in the context menu, but disabled.

If it turns out that the exclusivity locks are too restrictive, we can try to relax them on an as-needed basis.

Revised interface for specifying collisions

The API and GUI interface for specifying collision behavior has been revised. In particular, the functions

```
setCollisions (a,b,behavior)
getCollisions (a,b)
```

have been replaced by

```
setCollisionBehavior (a,b,behavior)
getCollisionBehavior (a,b)
setDefaultCollisionBehavior (a,b,behavior)
getDefaultCollisionBehavior (a,b)
```

and related convenience methods. See the Javadocs for `MechModel`.

In the GUI, you can either set the default behaviors for a `MechModel`, or set specific collision behaviors by selecting a set of bodies and choosing Set collisions ... from the context menu. Self collision behavior can be set from the context menu for a single deformable body. See "Collision handling" in the UI Guide.

Updated editing panels

As part of the process of finishing the UI Guide, many of the editing panels have been revamped to make them clearer and easier to use. To simplify their initial presentation, many of the default property panels are now expandable.

Also, the label alignment mechanism for `LabeledComponent` has been generalized to allow it to take account of component borders. This means that labels align propertly even for panels within panels.

Jan 19, 2011

The reduced tongue model is working again, and has been renamed to `models.reducedFem.ReducedTongue`.

`MuscleTissue`, `MuscleFibreTissue` and their associated classes have been removed, and replaced with `FemMuscleModel` (which is the renamed version of `MuscleElementTissue`). In addition, `MuscleElementBundle` has been renamed to `MuscleBundle`.

Jan 17, 2011

The JNI interface for Pardiso 4.1 has been compiled for MacOS (Snow Leopard), Windows, and both 32 and 64 bit Linux systems.

If you specify `-usePardiso4` to artisynth (or if this is set in your `.artisynthInit` file), then Pardiso 4.1 will be used. Otherwise, Pardiso 3 will be used.

To use Pardiso 4.1, you will need to obtain a new licence from `http://www.pardiso-project.org` if you haven't already. The pardiso.lic file that you create from this will **not** be compatible with Pardiso 3, so if you switch between versions you will also need to switch the licence files. Be sure to save your old licence file if you do this because you can no longer get Pardiso 3 licences.

There appears to be a bug in the Intel OpemMP library provided for the MacOS Pardiso version, which causes spordic crashes occur if Pardiso is called from more than one Java thread. I seem to have been able to work around this by making the Scheduler "play" thread persistent, so that we simply create one play thread at startup and use it for all subsequent play actions.

Jan 2, 2011

It is now possible to load a model by specifying its `RootModel` class directly. Select Load from class in the File menu.

Dec 8, 2010

Control panels:

Control panels are now scrollable by default (previously, scrollablity had to be enabled via the `scrollable` property).

Tetgen and DelaunayInterpolator:

As part of the support for `MuscleElementTissue`, we have added a JNI interface for `tetgen`, called

```
maspack.geometry.TetgenTessellator
```

The native interface has been compiled for Linux, Windows, and MacOS.

Building on top of this, we have created a method called

```
maspack.geometry.DelaunayInterpolator
```

which can be used for sparse unstructured 3D interpolation. Suppose you have a set of 3D points which contain data values that you wish to interpolate. You can create a Delaunay interpolation and pass it the points via the `setPoints` method, which will create a Delaunay tessellation of these points. The method `getInterpolation` can then identify which of these data points should be used for interpolating at an arbitrary point, along with the weights needed for the interpolation.

Dec 7, 2010

A new class, `MuscleElementTissue`, has been completed in which muscle activation is effected by means of `MuscleElementBundles`. A MuscleElementBundle may contain both point-to-point actuators (of type `Muscle`, as in the current `MuscleBundle` used by `MuscleTissue`), as well as sets of FEM elements whose material behaviour may be augmented using a `MuscleMaterial` that acts in a specific direction.

Each element associated with a MuscleElementBundle is described by a `FemElementDesc` that identifies the element and specifies the direction (relative to the element's rest position) along which the anisotropic behaviour should be exercised. This direction acts in concert with a `MuscleMaterial` to produce a material behavior that is superimposed on top of the element's default isotropic material behaviour. A default MuscleMaterial is specified for the MuscleElementTissue. This may be overridden by specifying non-null MuscleMaterial materials for specific MuscleElementBundles or for individual FemElementDescs.

By default, the point-to-point actuators (called *fibres*) in a MuscleElementBundle are inactive and are used only for visualization and determining the activation directions within individual elements (see below). However, the actuators can be activated via the `fibresActive` property. Likewise, the directional material behavior associated with the elements may be deactivated by specifying `NullMuscle` as the muscle material.

Two methods are currently provided to assist in determining the elements and activation directions for a MuscleElementBundle:

`computeElementDirections()` Computes directions for each element by performing a Delaunay interpolation based on the center position of the element and the centers of the nearest point-to-point actuators (with "nearest" being determined using a Delaunay tessellation). All calculations are done with respect to rest coordinates.

`addElementsNearFibres(dist)` Adds all elements that are within `dist` units of a fibre center, and sets their directions to that of the closest fibre.

A demo of MuscleElementTissue is provided by

```
artisynth.models.femdemos.MuscleElementDemo
```

which contains three muscle bundles: "top", "mid", and "bot". The following variables within the code can be used to control the muscle elements and directions associated with the middle ("mid") bundle:

`defaultMidElements` Describes which elements should be initially added (either all, none, or the middle elements).

`addMidElementsWithin` Adds all elements within a prescribed distance of the middle bundle fibres.

`autoComputeMidDirections` Automatically compute element directions from the middle bundle fibres using the Delauany interpolation described above.

Nov 18, 2010

After much delay, gravity has now been made an inherited property in both `FemModel` and `MechModel`. Also, gravity is now set using a full 3-vector. So instead of

```
model.setGravity (9.8);
```

you should do either

```
model.setGravity (0, 0, -9.8);
```

or

```
model.setGravity (new Vector3d (0, 0, -9.8));
```

Don't forget the minus sign! Likewise, `getGravity()` now returns a 3-vector. All the current code has been updated to reflect this.

Nov 17, 2010

A constraint `ParticlePlaneConstraint` has been added to MechModel which allows particles to be constrained to a fixed plane. The principal methods are

```
ParticlePlaneConstraint c =
   new ParticlePlaneConstraint(particle, plane);

model.addParticleConstraint (c);
model.removeParticleConstraint (c)
model.clearParticleConstraints();
```

For a demo, see

```
artisynth.models.femdemos.PlaneConstrainedFem
```

`ParticlePlaneConstraint` is an instance of a more general constraint class. It should now be relatively easy to add more complex constraints involving particles.

Nov 14, 2010

Problems have been fixed in the panel for editing the mesh geometry and inertia of a `RigidBody`. To edit these, select a rigid body, and choose

```
Edit geometry and inertia
```

from the context menu.

The methods in `RigidBody` for setting inertia have also been rationalized. First, there are two new methods:

`setInertiaMethod (InertiaMethod m)` specifies the method by which inertia is determined

`getInertiaMethod()` returns the current inertia method

along with a corresponding property `inertiaMethod`, which has three settings:

Explicit Inertia is specified explicitly

Density Inertia is calculated from the mesh using a specified density

Mass Inertia is calculated from the mesh using a specified mass.

Both the *Density* and *Mass* methods cause the inertia to be recomputed whenever the mesh, mass, or density is changed. Density is now defined simply as mass divided by mesh volume, and so setting either will cause the other to be updated to reflect this. There are also three main support methods:

`setInertia (SpatialInertia M)` explicitly sets the inertia and sets the inertia method to *Explicit*

`setInertiaByDensity (double density)` sets the inertia from a given density and sets the inertia method to *Density*

`setInertiaByMass (double mass)` sets the inertia from a given mass and sets the inertia method to *Mass*.

As before, there are a bevy of methods for explicitly setting the inertia in special ways. Note also that `set/getSpatialInertia` have been renamed to `set/getInertia`, and -1 is no longer a valid value for the density.

Nov 10, 2010

A `FullPlanarJoint` constraint has been implemented to restrict the motion of a RigidBody to a plane. This constraint is similar to RevoluteJoint, except that it allows translation in the plane perpendicular to the joint axis. For a demo, see

```
artisynth.models.mechdemos.PlaneConstrainedJaw
```

Note that when this joint is attached to a rigid body, care must be taken that other joints attached to the body do not over-constrain it. In particular, you can't attach both a RevoluteJoint and FullPlanarJoint to a single body (although if the z axes of the two joints are parallel, you won't need to, since RevoluteJoint restricts the body to a plane as part of it's normal operation). While there exist techniques that allow for the resolution of redundant constraints, these are not currently implemented in ArtiSynth.

The main motivation for FullPlanarJoint is to allow implementation of a reduced-complexity symmetric models.

Nov 10, 2010

Self-collision handling for deformable bodies is now implemented using sub-surfaces. This should be considered a temporary measure until proper self-intersection detection is implemented for meshes.

A deformable body will now handle self-collisions if

- Collisions are enabled between the body and itself, e.g.,

```
mechModel.setCollisions (femModel, femModel, true);
```

- The model contains two or more sub-surfaces (described below).

For a demo, see

```
artisynth.models.femdemos.SelfCollision
```

A sub-surface is a closed, manifold mesh that enclosed a portion of the FEM model. Each vertex of a sub-surface must correspond to a node of the FEM. Self-collision within the model is implemented by enforcing collision handling between all the sub-surface pairs. Note that this is not a complete solution, since collision handling will be restricted to sub-surface interactions. However, this may be desirable in some cases.

`FemModel3d` contains the following methods for managing sub-surfaces:

```
numSubSurfaces()
getSubSurface(int)
addSubSurface(PolygonalMesh)
removeSubSurface(PolygonalMesh)
clearSubSurfaces()
```

Rendering of sub-surfaces can be enabled via the `subSurfaceRendering` property.

A sub-surface can be created by reading it in from a file. `FemModel3d` contains the following methods to support this:

```
scanMesh(String fileName)
scanMesh(ReaderTokenizer rtok)
scanSurfaceMesh(ReaderTokenizer rtok)
scanSurfaceMesh(String fileName)
writeMesh(PrintWriter pw, PolygonalMesh mesh)
writeSurfaceMesh(PrintWriter pw)
writeSurfaceMesh(String fileName)
```

The file format contains a list of faces, whose vertices are described by a (counter-clockwise) list of their corresponding node numbers.

One way to create a sub-surface is to select the elements that should be used to form the sub-surface, and then choose

```
  Build surface mesh for selected elements
```

in the context menu. The resulting surface mesh can then be saved to a file using the Jython console and the `write` methods listed above.

Nov 9, 2010

I have added a couple of new flags to the `artisynth` command:

`-useAjlCollision` Enables Andrew Larkin's collision detection

`-showJythonConsole` Create the Jython console on start-up

May 13, 2010

A trapezoidal integrator has been added. This is a second-order Newmark method which does a fully constrained solve in the manner of ConstrainedBackwardEuler and should provide greater accuracy. To select it in code, you can do

```
model.setIntegrator (MechSystemSolver.Integrator.Trapezoidal);
```

Otherwise, you can set the model's integrator property through a widget.

Mar 9, 2010

I have created a general `CompositePropertyPanel` class which can be used for setting and selecting `CompositeProperties` within a larger panel, in the same style as `MaterialPanel`. The latter is now an instance of the former.

In particular, CompositePropertyPanel (and hence MaterialPanel) should work properly when directed at multiple components.

Another small change: property and render property panels now have names based on the set of components they are controlling.

Jan 26, 2010

I have added support for different kinds of position stabilization, through the artisynth option `-posCorrection`, which can be specified either on the command line or in your `.artisynthInit` file. This option accepts one of the following string arguments:

`Local` applies a local (Gauss-Seidel type) stabilization which we have been using until now.

`GlobalMass` applies a global position correction using impulses computed with the system mass matrix.

`GlobalStiffness` applies a global position correction using impulses computed with the complete system stiffness matrix.

`Default` applies the default position correction.

At the moment, I have set the default behavior to use `Local` stabilization for explicit integrators and `GlobalMass` stabilization for implicit ones, since `GlobalMass` stabilization doesn't seem to incur much compute penalty. `GlobalStiffness` stabilization, on the other hand, while a bit more robust, can (at present) almost double the computation time.

For implicit integrators, I do apply a one-time `GlobalStiffness` correction at the start of the first time step.

Jan 15, 2010

The code has been refactored to correctly implement point-based attachments, and some minor bugs involving deformable body contact have also been fixed.

Also, the rendering of individual finite elements now includes an optional widget in the center of the element that can be used for selection. The widget shows the shape of the element in miniature, with its proportionate size controlled by the property `elementWidgetSize`, which appears in both `FemModel3d` and `FemElement3d`. Element rendering has also been improved so that the edges of selected elements appear fully illuminated.

Oct 22, 2009

Improvements have been made to the Jython console. These include:

- Built in functions (see below)
- Initialization files
- Scripting support
- Line wrapping now works correctly, and the console is embedded in a scroll pane

Built-in functions

A number of built-in functions have been added, allowing you to do certain things easily without having to locate the appropriate java object and in particular without having to access `main`. For example, to add a break point and run the current model, you can now do

```
>>> addBreakPoint (10)
>>> run()
```

The current set of built-ins include:

`run()` run the simulation

`run(tinc)` run for a certain time

`pause()` pause the simulation

`waitForStop()` wait for the simulation to stop

`reset()` reset the simulation

`step()` single step the simulation

`addWayPoint(t)` add a waypoint at time t

`addBreakPoint(t)` add a breakpoint at time t

`removeWayPoint(t)` remove a waypoint or breakpoint at time t

`clearWayPoints()` clear all waypoints and breakpoints

`root()` get the current root model

`script(fileName)` run a script (see below)

`loadModel(name)` load a model by it's demo name

`find(name)` find a component by a name relative to the root model

It is expected that the set of built-ins will expand greatly and will be subject to modification.

Initialization files

The built-ins are defined in the initialization file `.artisynthJythonInit.py`, located in the ArtiSynth home directory. This is a Jython script that is executed once when the console starts up. It can be modified to add additional built-ins, by either defining them directly using `def`, or by adding a java method directly to the interpreter's dictionary using a statement of the form

```
_interpreter_.set ("waitForStop", main.waitForStop)
```

where the symbol `_interpreter_` references the interpreter itself.

Users can also define their own `.artisynthJythonInit.py` initialization files, in any directory inside the `ARTISYNTH_PATH`. Multiple files can be defined, with evaluation proceeding from last to first along the path.

Scripting

The built-in `script()` executes a script file within the console. This is similar to the standard built-in `execfile()`, except that the script is run in a separate thread and echos its commands to the console. This allows GUI interaction and rendering to proceed concurrently with the script execution. A script can be aborted by typing `^C`.

As an example, try running

```
>>> script ("testscript.py")
```

in the ArtiSynth home directory. This loads and runs some demos with a variety of integrators and logs the resulting state vectors into a file.

Oct 16, 2009

Materials have been made to properly implement `scaleDistance` and `scaleMass`. The numeric format string for a text widget has been made into a property, so that it can now be set by selecting the widget and choosing set properties from the context menu.

Some minor bugs have been fixed, and a number of internal changes have been made, mostly in preparation for fixing the interaction problem between attached particles and other constraints.

Sept 22, 2009

For anyone installing documentation on the ArtiSynth web server:

The `Makefiles` in the documentation directories now contain the command

```
> make install_html
```

that will create html documentation and then copy it onto the server. This assumes you have an account on the server, and that you have set the environment variable `ARTISYNTH_WEB_ACCOUNT` to the name of said account. Unfortunately you'll be asked for your account password twice: once to copy the files, and once again to set the permissions so other people can modify them.

Permission setting is done by a revised script called `setMagicPerms`, located on the server.

For more details, see the documentation document.

Sept 20, 2009

Lagrange multiplier-based incompressibility has been added for Hex elements. You can now select `incompressible` for an FEM model consisting either entirely of Hex elements or Tet elements (although unfortunately not for mixed element models because the formulations aren't compatible). The results can be very good - try it with the HexTongue demo using the Linear material.

Hex element incompressibility should work with nonlinear materials as well. For incompressible materials, it should simply complement the incompressible penalty force added by the material.

However, as can be seen with the HexTongue and HexBeam3d demos, incompressibility with nonlinear materials also seems to go unstable at higher compressions. The most likely culprit is that our semi-implicit integrators are no longer sufficient, and we need to use a fully implicit integrator instead. That means adding Newton iterations onto the existing semi-implicit steps, which will take a little bit of work. To begin, I'll compute the residuals from the semi-implicit steps - if these start getting large right before the instability, that will suggest the need for fully implicit integration.

Other changes:

The Material widget has been completed to the point where you should now be able to replace widgets controlling a FemModel's `YoungsModulus`, `PoissonsRatio`, and `warping` properties with a single widget controlling the model's `material` property. One remaining issue is that the Material widget will still not work properly with a group of objects (such as a collection of elements). Obviously this needs to be fixed.

If you create a widget for a property whose value is `double`, the widget will now automatically contain a slider. The range of the slider will be determined automatically from the current value of the property. If the current value is zero, then a default range of [0,1] will be assigned. This is not restrictive since slider ranges now readjust on the fly, as described next.

Sliders fields have been modified so that if you enter a number in the text box that exceeds the slider's range, the range will be automatically increased to accommodate this. This was done by giving these components a 'slider range' in addition to their regular range. Slider ranges must still lie within the regular range, but since the regular range is often something like `[0, +inf]` or `[-inf, +inf]`, this is not generally a problem.

Finally, slider widgets have been altered so that the system tries to ensure that they have a track length of 200 pixels. This helps ensure reasonable value increments as long as the slider's range is itself cleanly divisible by 200.

Sept 3, 2009

Support has been added for nonlinear FEM materials. For application programming, a `material` property has been added to both `FemModel` and `FemElement`. This is a composite property whose sub-properties describe the parameters of the material in question. A FemElement's material can be `null`, in which case the material for the FemModel is used instead.

A new type of widget, called a `MaterialPanel`, has been created in `artisynth.core.gui` to support editing of materials and their properties.

Some simple materials are defined in `artisynth.core.femmodels.materials`.

There are still some rough edges being sorted out in the code. Incompressible materials are currently implemented using a penalty method. This has yet to be unified with the constraint-based incompressibility available for tetrahedral elements.

July 31, 2009

The ArtiSynth website now has an update log that you can access from the sidebar heading `Update Log`. All messages posted to the `artisynth-updates` mailing list will appear there in a more readable form.

The update log is written in AsciiDoc and its source is located in `$ARTISYNTH_HOME/doc/updates/updates.txt`. You can make your own changes to the log if your system is configured to compile ArtiSynth documentation (see Writing Documentation for ArtiSynth) and you have an account on the ArtiSynth web server machine.

Running the command

```
> make post
```

from within the `updates` directory will compile `updates.txt` into *html* and copy it to the website. You must have the environment variable `$ARTISYNTH_WEB_ACCOUNT` set to the name of your account on the web server.

Other things: all Asciidoc documentation on the website is now nested within the main frame (thanks to Byron for this), and `artdoc` (the interface to AsciiDoc that generates documentation) has two new options:

–no-contents do not create a table of contents

–section-number-depth *depth* set the section number depth (where 0 disables numbering)

‘

July 27, 2009

Probes has been modified so that start times, stop times, and update intervals are now specified in seconds instead of ticks. This removes the need to call `TimeBase.secondsToTicks()` when accessing these quantities. All committed code has been reformatted, so you shouldn't need to do anything. All testing comes up clean, but let Dr. Lloyd know if you see anything suspicious. A good number of files were touched so you should do a general update. Some internal ArtiSynth code still uses ticks, so the convenience methods `getStartTimeTicks()` and `getStopTimeTicks()` have been provided. Also, start and stop times are still written to files using ticks; this is to prevent breaking existing files and will be changed when all the probe data files are converted.

July 26, 2009

The LegendDisplay code that controls the plotting of lines in NumericProbeBase displays has been reimplemented.

The main changes are:

1. The legend now contains more informative labeling which is based, if possible, on the properties associated with the probe. 2. Labels can be set by the user: right click at the bottom of the panel and select "Enable label editing". 3. Legend information is saved and restored with the probe.

In terms of implementation, the `LegendDisplay` is now actually owned by its probe, which may not be ideal but solves a lot of problems and is consistent with the fact that the displays themselves are owned by the probe. All Legend code that was in ProbeInfo has been removed. Also, the `LegendDisplay` is now a subclass of PropertyPanel, which greatly simplifies the code.

July 20, 2009

The last major updates for the collision code have been checked in. Here are the highlights:

Collision API in MechModel

- The collision behavior between all Collidable bodies is specified in a MechModel using `setCollisions (a, b, enabled, friction)` where 'a', 'b' specifies a pair of Collidables, '`enable`' enables or disables collisions, and '`friction`' gives the coefficient of friction.
- You can specify collisions between individual collidables, or use `Collidable.RigidBody` or `Collidable.Deformable` to specify default collision behaviors for
  - RigidBody-RigidBody
  - RigidBody-Deformable
  - Deformable-Deformable
- The convenience method `setDefaultCollisions (enabled, friction)` specifies all three of the above.
- Default and specific collidables cannot currently be mixed; e.g., you cannot do

```
RigidBody box = createBox();
setCollisions (box, Collidable.RigidBody);
```

- The collision behavior for any pair of Collidables can be queried using `getCollisions (a, b)`;. If the pair is contained in one or more sub-models, then explicitly set behaviors in higher level models take priority. For example, `setCollisions(a, b, true, 1);` has a higher priority than `subModel.setCollisions(a, b, false, 0);`. If there is no explicitly set behavior for the pair, then the default behavior in the lowest level sub-model containing `(a,b)` is used. The returned behavior will be determined by
  - any explicitly set behavior for `(a,b)`, or
  - the default behavior for the given pair type.

Graphically Editing Collisions

There are several ways to graphically edit collisions.

- Select a MechModel, followed by "Edit Collisions" from the context menu. This will bring up a panel that shows you the default settings for the model, plus all explicitly specified collision pairs, in the current model and any sub-model. The latter are presented using a two-level expandable tree. To set new behaviors, select the desired defaults and/or explicit pairs, set the desired enabled and the friction settings in the fields below the JTree, and click "Set". To remove explicitly set behaviors from the current model, select said behaviors and click "Unset".
- Select a set of Collidables, followed by "Set Collisions" in the context menu. This will bring up a dialog which lets you collectively set the collision behavior between all the selected collidables. This is done by adding explicit behaviors in the lowest level MechModel containing all the collidables (or in the MechModel associated with the most recently opened "Edit Collisions" panel).
- Select a set of Collidables, followed by "Unset Collisions" in the context menu. This will delete any explicitly set collision behaviors between the selected collidables in the lowest level MechModel containing them all (or in the MechModel associated with the most recently opened "Edit Collisions" panel).

Creating Basic RigidBodies

Some factory methods for creating RigidBodies have also been added. These automatically create the required mesh and set the inertia:

```
RigidBody.createBox (name, wx, wy, wz, density);
RigidBody.createSphere (name, r, density, nslices);
RigidBody.createEllipsoid (name, a, b, c, density, nslices);
RigidBody.createCylinder (name, r, h, density, nsides);
RigidBody.createFromMesh (name, mesh, density, scale);
RigidBody.createFromMesh (name, meshFilePath, density, scale);
```

July 3, 2009

A collection of updates has been checked into CVS. The bulk of these involve reformatting code in several packages, so a lot of files were touched, albeit without much change in functionality. The main changes are:

- Explicit integrators now use body coordinates by default. This shouldn't cause any problem, but if it does, you can revert by setting

  ```
  private static boolean useBodyCoordsForExplicit = false;
  ```

  MechSystemSolver.java.
- There is improved functionality for adding waypoints. Selecting "Add WayPoint(s) ..." on the timeline model track now provides you with a "repeat" field that lets you add a whole bunch of waypoints in one go.
- Another option, "Delete Waypoints", lets you delete all waypoints (except for the first one) in one go.

June 29, 2009

Work on converting property paths from the old format to the new one has been checked in. A number of .java, .art, and .probe files were touched, so updating is a good idea.

June 24, 2009

Modifications to code have been checked in, including:

- New editing functionality that allows attaching points to other points or to rigid bodies, or to remove these attachments.
- Reformating the code in artisynth.core.gui.editorManager; this is a start at reformatting all the code as we have been discussing, in order to make it more compatible with standard practice and hopefully easier for most people to write and understand.
- An eclipse settings file for the new code format can be found in

  `\$ARTISYNTH\_HOME/support/eclipse/artisynthCodeFormat.xml`

June 16, 2009

New property paths are now in effect. This affects both Java code and the .art files containing model and probe information. Please do an update on the entire distribution. The property part of the path is now separated from the component path by a semi-colon, as in `models/mechmodel/particles/0:mass`. Previously, a '/' was used, so that the above would have appeared as `models/mechmodel/particles/0/mass`. As many of the easily found old-style paths have been updated, in both the Java code and .art files, but some may have been missed. Qsubst may be helpful in fixing any .art files you have that are not checked in. The following invocations may be useful, and can be used independently:

```
> qsubst '(property="[^\s]*)/([^/:"]*")' '\1:\2' -re -find '*.art'

> qsubst '/excitation' ':excitation' -find '*.art'
```

ArtiSynth is still forgiving if it encounters an old-style path name, and it will print a warning message like this:

> **Warning:**
> Old style property path models/jawmodel/frameMarkers/lowerincisor/displacement should be replaced with models/jaw-model/frameMarkers/lowerincisor:displacement

which should be taken as a strong hint to fix old-style path.

June 15, 2009

Most of these involve improving the editing of RigidBody geometry and inertia, which in turn required some changes and additions to the widget code. The `artisynth` script has been reworked. The main changes are:

- `artisynth -help` now works properly
- the log file is placed in `$ARTISYNTH_HOME/tmp`, instead of `$ARTISYNTH_HOME`
- The `-v` option has been removed. Output is sent to the console (as well as the log file) by default. If you *don't* want console output, use the `-s` option.

Note that the log file is a bit of a hack and may change/disappear later.

May 29, 2009

- Unstable behavior is now detected properly and you get an appropriate exception indicating such, rather than some side effect like `Bad Cholesky factorization`.
- Dragger positions are now kept current with the bounded box for selected objects (or the coordinate frame if a single Frame or RigidBody is selected).
- Button masks for things like the context menu are now stored in `artisynth.core.gui.ButtonMask` (the context menu mask used to be stored in `artisynth.core.gui.selectionManager.SelectionManager`).
- All GUI components that create context popups now use `ButtonMask.getContextMenuMask()` and so should work properly on the MacBook.
- To add Frame markers, you now use the add FrameMarkers option with a MechModel selected, and you can click on **any** rigid body owned by that MechModel.
- Some extra material and figures has been added to doc/uiguide

May 26, 2009

Component names can no longer contain a colon ':', because that character is used in component/property path names. It has been illegal for a number of weeks, but it has just been removed from names in existing model files, and replaced with an underscore '_'. This mostly affects tongue data files, where muscle groups were often given names like `"f12:3"`. Those names now look like `"f12_3"`. If you have model or probe files that are not part of the checked-in code base, then you can convert ':' to '_' yourself using the qsubst command:

```
> qsubst 'name="([^:]*):([^:]*)"' 'name="\1_\2"' -re <files> ...
```

Also, `this` is no longer used in component path names. Instead, the '`.`' character is used, in complete analogy with Unix path names. For example, "`=this`" has been replaced with "`=.`" in model and probe files; please do the same for files that are not part of the code base.

May 21, 2009

A new command called qsubst has been added to `$ARTISYNTH_HOME/bin`. It's a python script that allows you to do interactive string replacement in a set of files. You specify a string expression, its replacement, and one or more files, and it goes through each file, prints all the matches with some surrounding context lines, and you hit a key indicating whether or not to do the replacement. Hitting ' ' means replace, '`n`' means don't replace. For example,

```
> qsubst double float Vector.java Matrix.java
```

will let you interactively replace '`double`' with '`float`' in `Vector.java` and `Matrix.java`. There are additional key commands as well as some command line options;

```
> qsubst -help
```

provides a synopsis. In particular, if you specify the -re option, then the expression is a Python regular expression, and the replacement string can contain group names. Fairly powerful stuff. qsubst will probably come in handy for modifying model and probe files. No guarantees are made for Windows; that depends on how well the `curses` package is supported.

May 14, 2009

Some fairly major ArtiSynth changes have been checked in. The visible changes are not that large, but there was some significant code refactoring and about 200 files were modified. Users should do a `cvs update -dP` from the artisynth root directory. These changes include:

- Adding documentation in the doc directory.
- Refactoring of the widget and viewer interface code.
- Changes in the look-and-feel of the probe editors.
- The grid and the clip planes now have properties, which allow you to set the grid spacing, color, line width, etc. To edit properties for the grid, right click on the grid resolution widget (which appears at the right of the menu bar when the grid is enabled). To edit properties for the clip planes, right click on the appropriate clip plane icon.
- The Viewer now has some properties too. To edit them, right click in the viewer when nothing else is selected. More properties will be exposed in the future.